状压dp

概念

状态压缩DP，也称集合状态DP，英文称为Bitmask DP，是指有一些问题需要维护集合状态，为方便表示集合，把每个元素对应于一个整数的各个二进制位(bit)。某一位如果是0，表示该元素不在集合中，如果是1，表示该元素在集合中。例如，考虑有5个元素的集合$A = \{0, 1, 2, 3, 4\}$，元素从左向右依次编号为$0 \sim 4$，则可以用长度为5的二进制数01101表示子集$\{0, 2, 3\}$。利用位运算，可以方便、高效把元素加入、移出集合，检查元素的存在性等。

这种把集合转化为整数记录在DP状态中，用整数的位运算进行状态转移的一类算法，称为状态压缩DP。

使用二进制数表示状态不仅缩小了数据存储空间，还能利用二进制数的位运算很方便地进行状态转移。

位运算常用技巧

技巧	示例	代码实现
去掉最后一位	101101 -> 10110	x >> 1
在末尾加0	101101 -> 1011010	x << 1
在末尾加1	101101 -> 1011011	(x << 1) + 1
将最后一位置0	101101 -> 101100	(x | 1) - 1
将最后一位置1	101100 -> 101101	x | 1
将最后一位取反	101101 -> 101100	x ^ 1
将右数第k位置0	101101 -> 101001 (k = 3)	x & ~(1 << (k - 1))
将右数第k位置1	101001 -> 101101 (k = 3)	x | (1 << (k - 1))
右数第k位取反	101101 -> 101001 (k = 3)	x ^ (1 << (k - 1))
取末k位	101101 -> 101 (k = 3)	x & ((1 << k) - 1)
取右数第k位	101101 -> 1 (k = 3)	(x >> (k - 1)) & 1
把末k位置1	101101 -> 101111 (k = 3)	x | ((1 << k) - 1)
末k位取反	101101 -> 101010 (k = 3)	x ^ ((1 << k) - 1)
把右起第一个0变成1	1010111 -> 1011111	x | (x + 1)
把右起第一个1变成0	1011000 -> 1010000	x & (x - 1)
把右边连续的0变成1	1011000 -> 1011111	x | (x - 1)
把右边连续的1变成0	1010111 -> 1011000	x & (x + 1)
取右边连续的1	1010111 -> 111	(x ^ (x + 1)) >> 1

技巧	示例	代码实现
获得1的数量	1010111 -> 5	__popcount(x)

4023 「SCOI2005」互不侵犯

题目描述

在 $N \times N$ 的棋盘里面放 K 个国王，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子。

输入格式

只有一行，包含两个数 $N(1 \leq N \leq 9), K(0 \leq K \leq N \times N)$。

输出格式

所得的方案数

样例输入

```
3 2
```

样例输出

```
16
```

Solution

首先观察题面，我们发现N非常小，于是考虑对N，即棋盘的每一行进行状压

对于题目中的限制而言，我们只需要保证每排之内不存在冲突，以及每排和**前一排**不存在冲突即可

对于排内不存在冲突，我们只需要保证所有合法的状态k，其所有二进制位为1的位置，左右两边均不为1，因此，我们可以很简单地利用如下表达式进行判断：

$$((k >> 1)\&k) \text{ or } ((k << 1)\&k)$$

我们令$\mathrm{dp}[i][j][k]$为前i排，一共用了j个棋子，第i排的棋子的状态是k

那么，我们需要枚举**前一排**所有可能的状态$\mathrm{dp}[i-1][j-\mathrm{popcount}(k)][k']$，如果状态合法，则将其直接加到$\mathrm{dp}[i][j][k]$上

而对于合法性的判断，我们只需要判断k和k'是否有某个位置均为1，或是某个为1的位置，在另一侧左右一位为1，即：

$$((k >> 1)\&k') \text{ or } ((k << 1)\&k') \text{ or } (k\&k')$$

Code

```
void solve(){
    int n, k; cin >> n >> k;
    vector dp(k + 1, vector(n, vector(1 << n, 0ll)));

    for (int i = 0; i < (1 << n); i++) {
        if ((i & (i << 1)) || (i & (i >> 1))) continue;
        if (__popcount(i) <= k) dp[__popcount(i)][0][i] = 1;
    }

    for (int i = 1; i < n; i++) for (int j = 0; j <= k; j++)
    for (int now = 0; now < (1 << n); now++){
        if (j - __popcount(now) < 0) continue;
        if ((now & (now << 1)) || (now & (now >> 1))) continue;
        for (int pre = 0; pre < (1 << n); pre++) {
            if ((now & pre) || (now & (pre << 1)) || (now & (pre >> 1)))
continue;
            dp[j][i][now] += dp[j - __popcount(now)][i - 1][pre];
        }
    }

    cout << accumulate(all(dp[k][n - 1]), 0ll) << endl;
}
```

16073 「BZOJ2073 POI2004」PRZ

题目背景

一只队伍在爬山时碰到了雪崩，他们在逃跑时遇到了一座桥，他们要尽快的过桥。

题目描述

桥已经很旧了，所以它不能承受太重的东西。任何时候队伍在桥上的人都不能超过一定的限制。 所以这只队伍过桥时只能分批过，当一组全部过去时，下一组才能接着过。队伍里每个人过桥都需要特定的时间，当一批队员过桥时时间应该算走得最慢的那一个，每个人也有特定的重量，我们想知道如何分批过桥能使总时间最少。

输入格式

第一行两个数: $W(100 \leq W \leq 400)$ 表示桥能承受的最大重量和 $n(1 \leq n \leq 16)$ 表示队员总数。

接下来 n 行：每行两个数: $t(1 \leq t \leq 50)$ 表示该队员过桥所需时间和 $w(10 \leq w \leq 100)$ 表示该队员的重量。

输出格式

输出一个数表示最少的过桥时间。

样例输入

```
100 3
24 60
10 40
18 50
```

样例输出

```
42
```

Solution

我们假设$\mathrm{dp}[mask]$为我们总共通过队员分布为$mask$时，所需要的最少的时间

我们可以枚举$mask_i \in [1, (1 << n) - 1]$，通过计算$mask_i$中所有队员的总重量和通过的时间，预处理出来所有的，能够**一起通过**桥的$mask_i$的集合

随后，我们从小到大枚举$mask$，因为从小到大就可以保证，某一个$mask$被枚举到的同时，**其所有的子集都已经被枚举过了。**在枚举到某个集合$mask$时，我们在预处理出来的，能够一起通过桥的人员集合m，如果说$mask \mid m = mask$，则代表m集合是$mask$的子集，那么我们就可以有以下转移方程：

$$\mathrm{dp}[mask] = \min(\mathrm{dp}[mask], \mathrm{dp}[mask \oplus m] + time_m)$$

其中，$time_m$指的是，通过人员为m的情况下，通过桥所需要的**时间**，而$mask \oplus m$，则代表着$mask$所代表人员的集合，减去m所代表的人员的集合

Code

```
void solve(){
    int w, n; cin >> w >> n;
    vector a(n, make_pair(0, 0));
    for (auto& [t, w]: a) cin >> t >> w;

    vector cost(0, make_pair(0, 0));

    for (int i = 0; i < 1 << n; i++) {
        int maxt = 0, totw = 0;
        for (int j = 0; j < n; j++)
            if (i & (1 << j)) {
                cmax(maxt, a[j].first);
                totw += a[j].second;
            }
        if (totw <= w) cost.emplace_back(i, maxt);
    }

    vector dp(1 << n, inf);
    dp[0] = 0;
    for (int i = 1; i < 1 << n; i++) for (auto [status, w]: cost) {
        if ((status | i) != i) continue;
        cmin(dp[i], dp[i ^ status] + w);
    }

    cout << dp.back() << endl;
}
```

二进制集合操作

操作	集合表示	位运算语句
交集	$a \cap b$	`a & b`
并集	$a \cup b$	`a | b`
补集	$\bar{a}$	`~a` （全集为二进制都是 1）
差集	$a \setminus b$	`a & (~b)`
对称差	$a \Delta b$	`a ^ b`

模 2 的幂

一个数对2的**非负整数次幂**取模，等价于取二进制下一个数的后若干位，等价于和$mod - 1$进行**与**操作。

```
constexpr int mod = 1 << m;

template<typename T>
void md(T& a) {
    a &= (mod - 1);
}
```

于是可以知道，2的非负整数次幂对它本身取模，结果为0，即如果n是2的非负整数次幂，n和$n - 1$的与操作结果为0。

事实上，对于一个正整数n，$n - 1$会将n的最低位的1置零，并将后续位数全部置1。因此，n和$n - 1$的**与**操作等价于删掉n的最低位的1。

借此可以判断一个数是不是2的非负整数次幂。当且仅当n的二进制表示只有一个1时，n 为2的非负整数次幂。

```
bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

子集遍历

遍历一个二进制数表示的集合的全部子集，等价于枚举二进制数对应掩码的所有子掩码。

掩码是一串二进制码，用于和源码进行与运算，得到**屏蔽源码的若干输入位**（这些位在掩码中通常为0）后的新操作数。

掩码对于源码可以起到遮罩的作用，掩码中的1位意味着源码的相应位得到保留，掩码中的0位意味着源码的相应位进行置0操作。将掩码的若干1位改为0位可以得到掩码的子掩码，掩码本身也是自己的子掩码。

给定一个掩码m，希望有效迭代m的所有子掩码 ，可以考虑基于位运算技巧的实现。

```
// 降序遍历 m 的非空子集
for (int s = m; s; s = (s - 1) & m)
// s 是 m 的一个非空子集
```

接下来证明，上面的代码访问了所有m的子掩码，没有重复，并且按降序排列。

假设有一个当前位掩码s，并且想继续访问下一个位掩码。在掩码s中减去1，等价于删除掩码s中最低位的1，并将其右边的所有位变为1。

为了使$s-1$变为新的子掩码，需要删除掩码m中未包含的所有额外的1位，可以使用位运算$(s-1)\ \&\ m$来进行此移除。

以上两步操作，得到了比s小，且最大的，m的某个子掩码

例如，假设掩码m为1的位从小到大依次为$a_1, a_2, a_3, \cdots a_n$，假设我们在某次操作中，当前的掩码$s$的最低位为$a_i$，那么，我们就将$a_i$置$0$，同时将$a_{i-1}, a_{i-2} \cdots a_1$置$1$了，这样得到的新的掩码$s'$，就是最大的，且比$s$小的子掩码

特殊的，当掩码的子集为**空**时，我们需要特判一下

设我们掩码为1的位置的个数，为$popcount(m)$，那么，我们掩码m的子掩码个数，就是$O(2^{popcpunt(m)})$，也就是我们上述算法的时间复杂度

遍历所有掩码的子掩码

```
for (int m = 0; m < (1 << n); ++m)
  // 降序遍历 m 的非空子集
  for (int s = m; s; s = (s - 1) & m)
// s 是 m 的一个非空子集
```

如果掩码m具有k个1，那么它有2^k个子掩码。对于给定的k，对应有$\binom{n}{k}$个掩码m的子掩码，那么所有掩码的总数为：

$$\sum_{k=0}^{n} \binom{n}{k} \times 2^k$$

该项和二项式定理对$(1+2)^n$的展开相同，因此等于3^n

直观来说，该式即为我们选择$\binom{n}{k}$个1，剩下选择2的乘积求和

48654. 「ABC269C」Submask

题目描述

给定一个非负整数n，按升序打印满足下列条件的所有非负整数x。

对于每一个非负整数k都成立：

如果二进制下的x的k位上的数字是1，则二进制下的n的k位上的数字也是1。

输入格式

第1行：1个正整数$N(0 \leq N \leq 2^{60})$，保证有N在二进制的写法下，1的位数不超过15

输出格式

输出若干行，每行一个整数，表示答案

样例输入

```
576461302059761664
```

样例输出

```
0
524288
549755813888
549756338176
576460752303423488
576460752303947776
576461302059237376
576461302059761664
```

Solution

本题相当于枚举掩码n的所有子掩码，按题设要求，我们还要将空掩码输出，因此我们只需要按照**从大到小**遍历掩码的方式，加上0的特判后，输出时反向输出，或者使用`std::reverse`即可

Code

```
void solve() {
    ll n; cin >> n;
    vector ans(0, 0ll);

    for (ll s = n; ; s = (s - 1) & n) {
        ans.push_back(s);
        if (!s) break;
    }

    reverse(all(ans));
    for (auto v: ans) cout << v << '\n';
}
```

27006 [「abc187F」Close Group]

题目描述

给你一个n个点，m条边的**简单**无向图，删去一些边(可以是0条)，使得图满足以下性质：

- 任意两点a，b，如果a，b连通，那么a，b之间有边。

求在满足条件的情况下，最少的联通块数量。

输入格式

第一行两个整数$n, m(1 \le n \le 18, 0 \le m \le \frac{n(n-1)}{2})$

此后m行，每行两个整数$u, v(1 \le u, v \le n)$，代表无向图的边

输出格式

一行一个整数，表示答案

样例输入

```
10 11
9 10
2 10
8 9
3 4
5 8
1 8
5 6
2 5
3 6
6 9
1 9
```

样例输出

```
5
```

Solution

我们假设$\mathrm{dp}[mask]$为选中点为$mask$时，最少的连通块数量

显然，我们可以先使用$O(n)$的枚举算法，遍历出$mask$内所有存在的点，然后再用$O(n^2)$的枚举算法，判断这些点是否两两相连。如果这些点两两相连，则代表我们可以不移除任何边，则连通块的最小个数是1

此外，我们可以枚举$mask$所有的子集s，将s和$s \oplus mask$（该集合代表s集合在全集为$mask$下的补集）两个集合的最少连通块数量求和，就能够和$\mathrm{dp}[mask]$取最小值，即：

$$\mathrm{dp}[mask] = \min(\mathrm{dp}[mask], \mathrm{dp}[s] + \mathrm{dp}[s \oplus mask])$$

此时的时间复杂度是$O(n^2 2^n + 3^n)$，足以通过此题

Code

```
void solve() {
    int n, m; cin >> n >> m;
    vector e(n, vector(n, 0));
    vector dp(1 << n, inf);
    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v; u--, v--;
        e[u][v] = e[v][u] = 1;
    }
    for (int i = 0; i < n; i++) e[i][i] = 1;
```

```
    for (int i = 0; i < (1 << n); i++) {
        int ok = 1;
        vector in(0, 0);
        for (int j = 0; j < n; j++) if ((i >> j) & 1) in.push_back(j);
        for (auto u: in) for (auto v: in) ok &= e[u][v];
        if (ok) dp[i] = 1;
        for (int s = i; s; s = (s - 1) & i) cmin(dp[i], dp[s] + dp[s ^ i]);
    }

    cout << dp.back() << endl;
}
```

37400 「CF71E」Nuclear Fusion

题目描述

现给定n个初始原子和k个最终状态原子，求解是否能够通过这n个初始原子进行**聚变反应**生成**所有的**最终状态原子

为了简化题目，我们规定聚变反应是由**多**个原子生成**一个**原子的操作，其中，生成原子的**序号**，等同于所有参加聚变反应的原子**序号之和**，例如样例中的反应就如下所示：

$$
{}_{25}\mathrm{Mn} + {}_{6}\mathrm{C} + {}_{19}\mathrm{K} \to {}_{50}\mathrm{Sn}
$$
$$
{}_{27}\mathrm{Co} + {}_{30}\mathrm{Zn} + {}_{21}\mathrm{Sc} \to {}_{78}\mathrm{Pt}
$$
$$
{}_{3}\mathrm{Li} + {}_{12}\mathrm{Mg} + {}_{15}\mathrm{P} + {}_{9}\mathrm{F} \to {}_{39}\mathrm{Y}
$$

输入格式

第一行两个整数$n, k (1 \le k \le n \le 17)$，分别代表初始原子数和最终原子数

第二行有n个由空格隔开的字符串，每个字符串代表一个原子的英文缩写，代表初始原子集合

第三行有k个由空格隔开的字符串，每个字符串代表一个原子的英文缩写，代表最终原子集合

不保证初始原子集合或最终原子集合内所有原子**两两不同**

保证初始原子和最终原子的序号均不超过100

输出格式

如果不能通过使用给定的初始原子，生成**所有的**最终状态原子，输出一行一个字符串`NO`

否则，输出一行一个字符串`YES`，此后输出k行k个字符串，代表每个**最终原子**的聚变方程，聚变方程格式如下：

$$
X + Y + Z + \cdots + A + B \to N_k
$$

其中，$X, Y, Z, \cdots$是用于聚变反应的**初始原子**，右箭头由`->`构成，N_k是某一个最终状态原子

你的输出需要保证每个初始原子**至多只进行一次**聚变反应，且**每个**最终状态原子都要成为聚变反应的**最终产物**

样例输入

```
10 3
Mn Co Li Mg C P F Zn Sc K
Sn Pt Y
```

样例输出

```
YES
Mn+C+K->Sn
Co+Zn+Sc->Pt
Li+Mg+P+F->Y
```

Solution

该题状态压缩和子集枚举部分较为简单，但由于输入输出较为麻烦，因此该题比较考验**细心程度**和逻辑思维连贯程度

我们用$\mathrm{dp}[i][mask]$代表，使用的初始原子为$mask$，并且成功构建了**前**i个最终状态原子的**可行性**，其中，0代表不可行，1代表可行

对于每个i，我们首先枚举所有掩码j，如果说，掩码j代表的集合内所有原子**序号之和**不同于第i个最终状态原子的序号，则我们可以直接略过；在等同的状态下，我们可以使用掩码j所代表的初始原子，通过聚变反应生成**第**i**个最终原子**，此时，我们需要使用**剩下的初始原子**生成前$i-1$个最终原子

因此，我们枚举**剩下的初始原子**的子集，即设s为$full \oplus j$的子集，其中$full=(1<<n)-1$，即全体原子的集合，看集合s能否构建出前$i-1$个最终原子，即$\mathrm{dp}[i-1][s]$是否为1，如果可以，说明我们从集合s中引入集合j，并且构建出前i个最终原子，即$\mathrm{dp}[i][s\oplus j]=1$。

由于要输出所有的**反应方程式**，因此我们使用一个和dp数组等大的pre数组，记录我们每一个状态的**前置状态**，例如上文中的转移$\mathrm{dp}[i][s\oplus j]=1$，我们就有$pre[i][s\oplus j]=s$，只记录第二维的原因是，第一维固定是$i-1$，因此没必要记录

对于一些细节的具体处理和实现，可以参考下列代码

Code

```
vector<string> elements = {
    "", "H", "He", "Li", "Be", "B", "C", "N", "O", "F", "Ne", "Na", "Mg", "Al",
"Si",
    "P", "S", "Cl", "Ar", "K", "Ca", "Sc", "Ti", "V", "Cr", "Mn", "Fe", "Co",
"Ni", "Cu",
    "Zn", "Ga", "Ge", "As", "Se", "Br", "Kr", "Rb", "Sr", "Y", "Zr", "Nb", "Mo",
"Tc",
    "Ru", "Rh", "Pd", "Ag", "Cd", "In", "Sn", "Sb", "Te", "I", "Xe", "Cs", "Ba",
"La",
    "Ce", "Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho", "Er", "Tm",
"Yb", "Lu",
    "Hf", "Ta", "W", "Re", "Os", "Ir", "Pt", "Au", "Hg", "Tl", "Pb", "Bi", "Po",
"At",
    "Rn", "Fr", "Ra", "Ac", "Th", "Pa", "U", "Np", "Pu", "Am", "Cm", "Bk", "Cf",
"Es", "Fm"
};
```

```cpp
unordered_map<string, int> idx;

void init(){
    for (int i = 0; i < 101; i++) idx[elements[i]] = i;
}

void solve() {
    int n, k; cin >> n >> k;
    vector<pair<string, int>> a(n), b(k);
    for (auto& [s, i]: a) {cin >> s; i = idx[s];}
    for (auto& [s, i]: b) {cin >> s; i = idx[s];}

    vector sum(1 << n, 0);

    for (int i = 0; i < (1 << n); i++) {
        int tot = 0;
        for (int j = 0; j < n; j++) if ((1 << j) & i) tot += a[j].second;
        sum[i] = tot;
    }

    vector dp(k, vector(1 << n, 0)), pre = dp;

    for (int i = 0; i < k; i++) for (int j = 0; j < (1 << n); j++) {
        if (sum[j] != b[i].second) continue;
        if (i == 0) {dp[i][j] = 1; continue;}
        auto mask = ((1 << n) - 1) ^ j;
        for (int s = mask; s; s = (s - 1) & mask) {
            if (!dp[i - 1][s]) continue;
            pre[i][j | s] = s; dp[i][j | s] = 1;
        }
    }

    int flag = 0, bg = 0;
    for (int i = 0; i < (1 << n); i++) if (dp[k - 1][i] == 1) {
        flag = 1; bg = i;
    }

    if (!flag) {cout << "NO\n"; return;}

    vector<string> ans;
    for (int i = k - 1; i >= 0; i--) {
        int now = bg ^ pre[i][bg];
        string res;
        for (int j = 0; j < n; j++) if ((now >> j) & 1) {
            if (res.size()) res.push_back('+');
            res += a[j].first;
        }
        res += "->"; res += b[i].first; bg = pre[i][bg];
        ans.emplace_back(res);
    }

    cout << "YES\n";
    for (auto& v: ans) cout << v << '\n';
}
```

26345 糖果

题目描述

一共有m种口味的糖果出售，但并不单独出售散装糖果，而是k_i颗一罐，整罐出售，价值为a_i。糖果包装上注明了每罐中k_i颗糖果的口味，给定n罐糖果，问得到所有口味糖果的最少花费。

输入格式

第一行两个整数$n, m(1 \leq n \leq 1000, 1 \leq m \leq 10)$，分别表示$n$罐和$m$种口味

接下来n组数据，每组数据两行，第一行两个数$a_i(1 \leq a_i \leq 1000)$和$k_i(0 \leq k_i \leq m)$，分别表示该罐糖果的价值和颗数，第二行由$k_i$个数组成，表示该罐中每颗糖的口味。

输出格式

输出含有所有口味糖果的最小花费

样例输入

```
3 3
2 2
0 2
5 3
0 1 2
1 2
1 2
```

样例输出

```
3
```

Solution

我们假设$\mathrm{dp}[s]$为糖果口味集合为s时的最小花费

转移很显然，枚举购买所有罐装糖果，并将其所有口味添加进去即可，即：

$$\mathrm{dp}[s \mid val_i] = \min(\mathrm{dp}[s \mid val_i], \mathrm{dp}[s] + cost_i)$$

其中，$cost_i$为购买第i罐糖果的花费，val_i指的是该罐糖果所有的口味集合

Code

```cpp
void solve() {
    int n, m; cin >> n >> m;
    vector a(n, make_pair(0, 0));
    for (auto& [v, c]: a) {
        int k; cin >> c >> k;
        for (int i = 0; i < k; i++) {
            int u; cin >> u;
            v |= (1 << u);
        }
```

```
    }

    vector dp(1 << m, inf);
    dp[0] = 0;
    for (int i = 0; i < (1 << m); i++) for (auto [v, c]: a)
        cmin(dp[i | v], c + dp[i]);

    cout << dp.back() << endl;
}
```

10157 TSP问题1

题目描述

给定一张n个点的有向图及其**邻接矩阵**，求经过每个点**一次**的最短路径。

输入格式

第1行：1个整数$n(3 \le n \le 20)$，代表有向图点的数量

接下来n行，每行$n(0 \le a_{i,j} \le 10^4)$个整数，表示有向图的邻接矩阵

注意，

输出格式

第1行：1个整数，表示答案

样例输入

```
3
0 1 2
2 0 1
1 2 0
```

样例输出

```
2
```

Solution

该题比较简单，我们用$\mathrm{dp}[mask][u]$代表已经遍历的点集为$mask$，且**当前处于点**u的情况下，最短路径的长度

转移方程比较好想，就是从上述状态向**不在集合**$mask$内的点走一步，不断取$\min$即可

$$\mathrm{dp}[mask \mid (1 << v)][v] = \min(\mathrm{dp}[mask \mid (1 << v)][v], \mathrm{dp}[mask][u] + e[u][v])$$

其中，$e[u][v]$代表$u \to v$的路径长度

初始化也比较简单，由于我们要求**最小值**，因此我们将dp数组初始化成inf，同时将$\mathrm{dp}[1 << i][i]$初始化为0即可（代表从某个点出发，此时的最短路径为0）

最后的答案，我们枚举u，在全集$\mathrm{dp}[(1 << n) - 1][u]$内寻找最小值即可

Bonus：为什么$mask$要在数组第一维？

Code

```
void solve() {
    int n; cin >> n;
    vector e(n, vector(n, 0));
    for (auto& vec: e) for (auto& v: vec) cin >> v;

    vector dp(1 << n, vector(n, inf));
    for (int i = 0; i < n; i++) dp[1 << i][i] = 0;

    for (int s = 0; s < (1 << n) - 1; s++) for (int u = 0; u < n; u++)
    for (int v = 0; v < n; v++) {
        if (!(s >> u) & 1) continue;
        if ((s >> v) & 1) continue;
        cmin(dp[s | (1 << v)][v], dp[s][u] + e[u][v]);
    }

    cout << *min_element(all(dp.back())) << endl;
}
```

10158 TSP问题2

题目描述

给定一张n个点的有向图及其**邻接矩阵**，求从起点s开始，经过每个点**一次**的最短路径。

输入格式

第1行：2个整数$n, s(3 \leq n \leq 20, 0 \leq s < n)$，代表有向图点的数量和起点

接下来n行，每行n个整数，表示有向图的邻接矩阵

输出格式

第1行：1个整数，表示答案

样例输入

```
3
0 1 2
2 0 1
1 2 0
```

样例输出

```
3
```

Solution

由于起点固定，因此我们只需要在初始化时**只初始化起点**，即$\mathrm{dp}[1 << s][s] = 0$即可，这样就能保证数组内所有路径都是由起点出发的

最后统计答案时，我们由于需要**回到起点**，因此，对于每个$\mathrm{dp}[(1 << n) - 1][u]$，我们还需要额外加上$e[u][s]$

Bonus：如果起点不固定，如何进行求解？（提示：回路包含所有点）

Code

```
void solve() {
    int n, bg; cin >> n >> bg;
    vector e(n, vector(n, 0));
    for (auto& vec: e) for (auto& v: vec) cin >> v;

    vector dp(1 << n, vector(n, inf));
    dp[1 << bg][bg] = 0;

    for (int s = 0; s < (1 << n) - 1; s++) for (int u = 0; u < n; u++) {
        if ((s >> u) & 1) for (int v = 0; v < n; v++) {
            if ((s >> v) & 1) continue;
            cmin(dp[s | (1 << v)][v], dp[s][u] + e[u][v]);
        }
    }

    int ans = inf;
    for (int i = 0; i < n; i++) cmin(ans, dp.back()[i] + e[i][bg]);
    cout << ans << endl;
}
```

37724 [「CF11D」A Simple Task]

题目描述

给定一个简单图无向图，有n个顶点m条边。输出图中简单环的数量。简单环是指不重复顶点或边的环。

输入格式

输入的第一行包含两个整数n和$m(1 \le n \le 19, 0 \le m \le \frac{n(n-1)}{2})$ - 分别表示图的顶点数和边数。

接下来的 行每行包含两个整数a和$b(1 \le a, b \le n, a \ne b)$，表示顶点 和顶点 之间有一条无向边相连。

任意两个顶点之间最多只有一条边。

输出格式

输出给定图中环的数量。

样例输入

```
4 6
1 2
1 3
1 4
2 3
2 4
3 4
```

样例输出

```
7
```

Solution

观察到本题n范围较小，因此可以以**环中已经经过的点**作为状态压缩的集合

对于一个环而言，假设其顶点序列为$a_1, a_2, \cdots a_m$，那么，这个环可以按照$\{1, 2, \cdots m\}, \{2, 3, \cdots m, 1\}, \cdots, \{m, m-1, \cdots, 1\}$，这样$2m$种方式遍历，因此，我们需要人为规定起点为**序列中编号最大的点**

同时，为了保证我们知道下一个点该和集合内哪个点**连边**，我们还需要一维用于记录集合目前最后一个点是哪一个

因此，我们定义$\mathrm{dp}[u][mask]$为**最后经过的点是**u，点集是$mask$的情况下的路径数

我们枚举所有u能够**直接**到达的点v，如果v的编号比$mask$里面**编号最大的点**还大，那就不进行处理，否则分三种情况讨论。

第一种，v的编号**等于**$mask$内编号最大的点，代表我们从u走回了起点，相当于形成了一个完整的**环**，此时，我们可以直接有$ans+=\mathrm{dp}[u][mask]$

第二种，v已经在$mask$中出现过，且不属于第一种情况，说明我们访问重复，不需要进行处理

第三种，v没在$mask$中出现过，说明我们找到了一类新的路径，此时我们将$\mathrm{dp}[u][mask]$添加到我们$\mathrm{dp}[v][mask \mid (1<<v)]$里面，即$\mathrm{dp}[v][mask \mid (1<<v)]+=\mathrm{dp}[u][mask]$

初始情况比较好想，$\mathrm{dp}[i][1<<i]=1$，代表从某个点出发的，长度为0的路径只有一条

此时，我们的ans包括了**仅由一条边构成的环**，以及$\{1, 2, \cdots, m\}, \{1, m, m-1, \cdots, 2\}$这种**被计数了两次的环**，因此，最终的答案是$\frac{ans-m}{2}$

时间复杂度也比较好算，我们对于每个$mask$，要遍历所有的点及其相对应的边，相当于对于每个$mask$遍历边集，时间复杂度就是$O(m2^n)$

Code

```
void solve(){
    int n, m; cin >> n >> m;
    vector e(n, vector(0, 0));
    vector dp(n, vector(1 << n, 0ll));
    ll ans = 0;

    auto is_valid = [](int u, int m) {
```

```
        return ((1 << u) < m) && (((1 << u) | m) != m);
    };

    auto get_ans = [](int u, int m) {
        if (((1 << u) | m) != m) return 0;
        if ((1 << (u + 1)) < m) return 0;
        return 1;
    };

    for (int i = 0; i < n; i++) dp[i][1 << i] = 1;

    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v; u--, v--;
        e[u].push_back(v); e[v].push_back(u);
    }

    for (int j = 0; j < (1 << n); j++) for (int i = 0; i < n; i++) {
        if (!(j & (1 << i))) continue;
        for (auto v: e[i]) {
            if (is_valid(v, j)) dp[v][j | (1 << v)] += dp[i][j];
            else if (get_ans(v, j)) ans += dp[i][j];
        }
    }

    cout << (ans - m) / 2 << endl;
}
```

4215 「NOIP2016」愤怒的小鸟

题目描述

Kiana 最近沉迷于一款神奇的游戏无法自拔。

简单来说，这款游戏是在一个平面上进行的。

有一架弹弓位于 $(0,0)$ 处，每次 Kiana 可以用它向第一象限发射一只红色的小鸟，小鸟们的飞行轨迹均为形如 $y=ax^2+bx$ 的曲线，其中 a,b 是 Kiana 指定的参数，且必须满足 $a<0$，a,b 都是实数。

当小鸟落回地面（即 x 轴）时，它就会瞬间消失。

在游戏的某个关卡里，平面的第一象限中有 n 只绿色的小猪，其中第 i 只小猪所在的坐标为 (x_i,y_i)。

如果某只小鸟的飞行轨迹经过了 (x_i,y_i)，那么第 i 只小猪就会被消灭掉，同时小鸟将会沿着原先的轨迹继续飞行；

如果一只小鸟的飞行轨迹没有经过 (x_i,y_i)，那么这只小鸟飞行的全过程就不会对第 i 只小猪产生任何影响。

例如，若两只小猪分别位于 $(1,3)$ 和 $(3,3)$，Kiana 可以选择发射一只飞行轨迹为 $y=-x^2+4x$ 的小鸟，这样两只小猪就会被这只小鸟一起消灭。

而这个游戏的目的，就是通过发射小鸟消灭所有的小猪。

这款神奇游戏的每个关卡对 Kiana 来说都很难，所以 Kiana 还输入了一些神秘的指令，使得自己能更轻松地完成这个游戏。这些指令将在**输入格式**中详述。

假设这款游戏一共有 T 个关卡，现在 Kiana 想知道，对于每一个关卡，至少需要发射多少只小鸟才能消灭所有的小猪。由于她不会算，所以希望由你告诉她。

输入格式

第一行包含一个正整数 T，表示游戏的关卡总数。

下面依次输入这 T 个关卡的信息。每个关卡第一行包含两个非负整数 n, m，分别表示该关卡中的小猪数量和 Kiana 输入的神秘指令类型。接下来的 n 行中，第 i 行包含两个正实数 x_i, y_i，表示第 i 只小猪坐标为 (x_i, y_i)。数据保证同一个关卡中不存在两只坐标完全相同的小猪。

如果 $m = 0$，表示 Kiana 输入了一个没有任何作用的指令。

如果 $m = 1$，则这个关卡将会满足：至多用 $\lceil n/3 + 1 \rceil$ 只小鸟即可消灭所有小猪。

如果 $m = 2$，则这个关卡将会满足：一定存在一种最优解，其中有一只小鸟消灭了至少 $\lfloor n/3 \rfloor$ 只小猪。

保证 $1 \leq n \leq 18$，$0 \leq m \leq 2$，$0 < x_i, y_i < 10$，输入中的实数均保留到小数点后两位。

上文中，符号 $\lceil c \rceil$ 和 $\lfloor c \rfloor$ 分别表示对 c 向上取整和向下取整，例如：
$\lceil 2.1 \rceil = \lceil 2.9 \rceil = \lceil 3.0 \rceil = \lfloor 3.0 \rfloor = \lfloor 3.1 \rfloor = \lfloor 3.9 \rfloor = 3$。

对于$1 \leq n \leq 12$，有$1 \leq T \leq 30$，而对于$13 \leq n \leq 15$，有$1 \leq T \leq 15$，对于$15 \leq n \leq 18$，有$1 \leq T \leq 5$

输出格式

对每个关卡依次输出一行答案。

输出的每一行包含一个正整数，表示相应的关卡中，消灭所有小猪最少需要的小鸟数量。

样例输入

```
1
10 0
7.16 6.28
2.02 0.38
8.33 7.78
7.68 2.09
7.46 7.86
5.77 7.44
8.24 6.72
4.42 5.11
5.42 7.79
8.15 4.99
```

样例输出

```
6
```

Solution

对于本题而言，要想拿到满分，我们没有必要关注m这个变量

状态很明显，$mask$应该代表**已经被消灭**的猪头的集合，那么，我们转移的时候，就需要枚举**这一次**有哪些猪头被消灭了

对于形如$y=ax^2+bx$的曲线而言，由于其有**两个**未知数a,b，我们可以用**两点唯一确定**a,b，即枚举我们某两只猪头的位置，就能确定唯一一根曲线，再检验其它猪头和曲线有无交点即可

对于a,b的求解，设两个猪头的位置分别为$<x_i,y_i>,<x_j,y_j>$，我们可以联立以下两个二元一次方程：

$$y_i=ax_i^2+bx_i$$
$$y_j=ax_j^2+bx_j$$

解得：

$$a=\frac{y_ix_j-y_jx_i}{x_i^2x_j-x_j^2x_i},b=\frac{y_i-ax_i^2}{x_i}$$

由于题目要求$a<0$，因此我们在解方程后还需要判断一下方程的解是否合法；同时，若$x_i=x_j$，方程的解也不合法（因为有除0的过程）

于此同时，我们可以在$[1,n]$范围枚举i，$[i+1,n]$范围枚举j，$[j+1,n]$范围枚举剩下的猪头，这样，我们就可以对于每一个**序号最小的点**，存储经过它的所有抛物线，我们记为$line[i]$数组。特别的，$line[i]$中还应该有一种合法情况为**只经过猪头**i的曲线

状压dp的过程比较简单，我们令$\mathrm{dp}[mask]$为消灭了集合$mask$中的猪头，剩下没消灭的猪头中，我们假设编号最小那个为i，我们将$line[i]$中的所有合法情况遍历一遍，设合法情况为$line_{i,j}$，我们就得到了如下转移方程：

$$\mathrm{dp}[mask\mid line_{i,j}]=\min(\mathrm{dp}[mask\mid line_{i,j}],\mathrm{dp}[mask]+1)$$

此时，我们的状态数是$O(2^n)$，转移方程中，$line[i]$数组的长度是$O(n)$的（因为之多只有$O(n)$条不同的抛物线），因此动态规划的时间复杂度是$O(n2^n)$

由于我们要求的是**最小值**，因此dp数组应该被赋为inf，特殊的，没有猪头被消灭时，代价应该是0，即$\mathrm{dp}[0]=0$

Code

```
void solve() {
    int n, m; cin >> n >> m;
    vector pos(n, make_pair(0.0, 0.0));
    for (auto& [f, s]: pos) cin >> f >> s;

    auto eq = [](const db& a, const db& b) {
        return fabs(a - b) < eps;
    };

    auto calc = [&eq](const pair<db, db>& u, const pair<db, db>& v) -> pair<db,
db> {
        auto& [f, s] = u;
        auto& [x, y] = v;
```

```
        if (eq(f, x)) return {inf, inf};
        auto a = (s * x - y * f) / (f * f * x - x * x * f);
        auto b = (s - a * f * f) / f;
        return {a, b};
    };

    vector line(n, vector(0, 0));

    for (int i = 0; i < n; i++) for (int j = i + 1; j < n; j++) {
        if (eq(pos[i].first, pos[j].first)) continue;
        auto [a, b] = calc(pos[i], pos[j]);
        if (a >= 0) continue;
        auto mask = (1 << i) | (1 << j);
        for (int k = j + 1; k < n; k++)
            if (eq(a * pos[k].first * pos[k].first + b * pos[k].first,
pos[k].second))
                mask |= (1 << k);
        line[i].push_back(mask);
    }

    for (int i = 0; i < n; i++) line[i].push_back(1 << i);

    vector dp(1 << n, inf);
    dp[0] = 0;

    for (int i = 0; i < (1 << n) - 1; i++) {
        int bg = 0;
        for (; bg < n; bg++) if (!((i >> bg) & 1)) break;
        for (auto v: line[bg]) cmin(dp[i | v], dp[i] + 1);
    }

    cout << dp.back() << endl;
}
```

176 「USACO 2006 Nov」牧场的安排

题目描述

农场主 John 新买了一块长方形的新牧场，这块牧场被划分成 M 行 N 列，每一格都是一块正方形的土地。John 打算在牧场上的某几格里种上美味的草，供他的奶牛们享用。

遗憾的是，有些土地相当贫瘠，不能用来种草。并且，奶牛们喜欢独占一块草地的感觉，于是 John 不会选择两块相邻的土地，也就是说，没有哪两块草地有公共边。

John 想知道，如果不考虑草地的总块数，那么，一共有多少种种植方案可供他选择？（当然，把新牧场完全荒废也是一种方案）

输入格式

第一行：两个整数 N 和 $M(1 \leq N, M \leq 12)$，用空格隔开。

第 2 到第 $N+1$ 行：每行包含 N 个用空格隔开的整数，描述了每块土地的状态。第 $i+1$ 行描述了第 i 行的土地，所有整数均为 0 或 1，是 1 的话，表示这块土地足够肥沃，0 则表示这块土地不适合种草。

输出格式

一个整数，即牧场分配总方案数除以 10^8 的余数。

样例输入

```
2 3
1 1 1
0 1 0
```

样例输出

```
9
```

Solution

本题类似于**互不侵犯**，只要每一行之内不冲突，每一行和上一行不冲突即可

行内不冲突，设某行的状态是$mask$，一方面是某一块草地**左右**不能有草地，即$mask \ \& \ (mask << 1)$为0，另一方面就是不能在**贫瘠的地方**种草，我们可以将一行的**土地状态**用mp_i来表示，如果有$mask \mid mp_i = mp_i$，就代表我们没有在贫瘠的地方种草

而行间不冲突呢，假设当行状态为now，上行状态为pre，只需要简单的寻找有没有两块草地处于彼此的上下方，即$now \ \& \ pre$为0，就代表上下行不冲突

转移方程也显而易见：

$$\mathrm{dp}[i][now] = \sum_{pre=0}^{2^m-1} dp[i-1][pre] \times legal(pre, now)$$

其中$legal(pre, now)$函数代表我们pre和now是否合法

状态$O(n2^n)$种，每种状态转移方程$O(2^n)$个，总时间复杂度$O(n2^{2n})$，通过本题比较微妙

我们可以预处理出所有**行内**的合法状态（即没有两块草地相邻的状态），这样可以极大地减少我们的常数，就可以通过本题了

Code

```cpp
void solve() {
    int n, m; cin >> n >> m;
    vector mp(n, 0);
    for (auto& v: mp) for (int i = 0; i < m; i++) {
        int u; cin >> u;
        v <<= 1; v |= u;
    }

    vector status(0, 0);
    vector dp(2, vector(1 << m, 0));

    for (int i = 0; i < (1 << m); i++) if(!(i & (i << 1))) {
        status.push_back(i);
        if ((i | mp[0]) == mp[0]) dp[0][i] = 1;
    }
```

```
    for (int i = 1; i < n; i++) {
        fill(all(dp[i & 1]), 0);
        for (auto now: status) {
            if ((now | mp[i]) != mp[i]) continue;
            for (auto pre: status) {
                if (pre & now) continue;
                add(dp[i & 1][now], dp[(i & 1) ^ 1][pre]);
            }
        }
    }

    cout << accumulate(all(dp[(n - 1) & 1]), 0ll) % mod << endl;
}
```

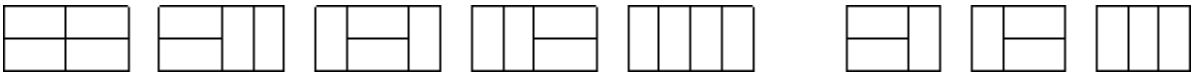
6127 蒙德里安的梦想

题目描述

求把$N \times M$的棋盘分割成若干个1×2的的长方形，有多少种方案。

例如当$N = 2, M = 4$时，共有5种方案。当$N = 2, M = 3$时，共有3种方案。

如下图所示：



输入格式

输入包含多组测试用例。

每组测试用例占一行，包含两个整数N和$M(1 \le N, M \le 11)$。

当输入用例$N = 0, M = 0$时，表示输入终止，且该用例无需处理。

输出格式

每个测试用例输出一个结果，每个结果占一行。

样例输入

```
1 2
1 3
1 4
2 2
2 3
2 4
2 11
4 11
0 0
```

样例输出

```
1
0
1
2
3
5
144
51205
```

Solution

我们令$\mathrm{dp}[i][mask]$为铺了前i行，第i行的状态为$mask$时的方案数

其中，$mask$某一位是1，代表着这个砖块要么是**横置**，要么是和**上一行**一起竖置，而$mask$某一位是0，代表着这个砖块需要和**下一行**一起竖置

我们枚举当行的状态now，和上一行的状态pre，首先就是上一行状态为0的位置，在这行状态必须为1，假设$full = (1 << m) - 1$，即全集，我们就可以通过$(pre \oplus full) \mid now$和$now$是否相等，得到这种情况是否得到满足。其次，除了和上一行竖置的1以外，其它横置的1也必须满足要求，即相邻的1的数量必须是**偶数**，我们可以先预处理出所有的状态是否满足要求，随后利用$pre \oplus full \oplus now$得到删除**竖置**$1$的剩余$mask$，再进行查表观察是否满足结果

如果上述结果都满足，我们就有转移方程如下：

$$\mathrm{dp}[i][now] += \mathrm{dp}[i-1][pre]$$

初始化时，记得只将$\mathrm{dp}[0][full]$初始化成1，因为第一行之前的行**不能**和第一行组合成竖置方块

最后答案就在$\mathrm{dp}[n][full]$得到，因为砖块要铺满

状态数$O(n2^m)$，每个状态的转移方程有$O(2^m)$个，因此时间复杂度为$O(Tn2^{2m})$

Code

```
void solve() {
    int n, m; cin >> n >> m;
    if (!n && !m) exit(0);

    vector dp(2, vector(1 << m, 0ll));
    dp[1].back() = 1;

    vector f(1 << m, 1);
    for (int i = 0; i < (1 << m); i++) for (int j = 0; j < m; j++) {
        if (!((i >> j) & 1)) continue;
        if (!((i >> (j + 1)) & 1)) f[i] = 0;
        j++;
    }

    auto full = (1 << m) - 1;

    for (int i = 0; i < n; i++) {
        fill(all(dp[i & 1]), 0);
```

```
        for (int now = 0; now < (1 << m); now++) for (int pre = 0; pre < (1 <<
m); pre++) {
            if (((pre ^ full) | now) != now) continue;
            if (!f[pre ^ now ^ full]) continue;
            dp[i & 1][now] += dp[(i & 1) ^ 1][pre];
        }
    }

    cout << dp[(n - 1) & 1].back() << endl;
}
```

17312 「BZOJ3312 Usaco2013 Nov」No Change

题目描述

约翰到商场购物，他的钱包里有k个硬币。

约翰想**按顺序**买n个物品，第i个物品需要花费c_i块钱。

在依次进行的购买n个物品的过程中，约翰可以随时停下来付款，每次付款只用一个硬币，支付购买的内容是从上一次支付后开始到现在的这些所有物品（前提是该硬币足以支付这些物品的费用）。不幸的是，商场的收银机坏了，如果约翰支付的硬币面值大于所需的费用，他不会得到任何找零。

请计算出在购买完n个物品后，约翰最多剩下多少钱。如果无法完成购买，输出`-1`

输入格式

第一行两个整数$k, n(1 \leq k \leq 16, 1 \leq n \leq 10^5)$

此后k行，每行一个整数$k_i(1 \leq k_i \leq 10^8)$，代表约翰各个硬币的面值

此后n行，每行一个整数$c_i(1 \leq c_i \leq 10^4)$，代表商品价值

输出格式

一行一个整数，表示约翰能够剩下的最多钱数

样例输入

```
3 6
12
15
10
6
3
3
2
3
7
```

样例输出

```
12
```

Solution

由于商品购买是**按顺序的**，因此我们可以考虑以商品购买的数量作为状态转移时的衡量标准

我们可以用$\mathrm{dp}[s]$代表，我们**剩余硬币集合为**s时，最多能购买的商品的件数

我们枚举集合s中所有**最后使用**的硬币，记其价值为v_j，在使用这个硬币之前的硬币集合可以用$pre =$。那么，我们在使用v_j这个硬币时（包括已经使用过的所有硬币），实际上的购买力等同于

$$v_j + \sum_{i=1}^{\mathrm{dp}[s]} c_i$$

因为已经使用过的硬币相当于已经购买了前$\mathrm{dp}[s]$个物品，且一个都没有剩下

因此，我们可以简单地使用前缀和 + 二分，找到我们最多能够买到哪个商品为止

我们可以在所有商品后加一个价值为INF的商品，这样可以有利于避免数组访问越界

最后，我们遍历整个dp数组，当对于某个集合s，有$\mathrm{dp}[s] \geq n$时，代表我们能购买完所有的物品，此时我们需要计算剩下硬币的面值，并不断与ans（初始化成-1）取最大值即可

Code

```
void solve(){
    int k, n; cin >> k >> n;
    vector a(k, 0ll), b(n + 2, 0ll);
    for (auto& v: a) cin >> v;
    for (int i = 1; i <= n; i++) {
        cin >> b[i]; b[i] += b[i - 1];
    }
    b[n + 1] = INF;

    vector dp(1 << k, 0);

    for (int i = (1 << k) - 1; i >= 0; i--) for (int j = 0; j < k; j++) {
        if ((i >> j) & 1) continue;
        auto pre = i | (1 << j);
        auto tot = b[dp[pre]] + a[j];
        cmax(dp[i], (int)(upper_bound(all(b), tot) - b.begin() - 1));
    }

    int ans = -1;
    for (int i = 0; i < (1 << k); i++) {
        if (dp[i] < n) continue;
        int tot = 0;
        for (int j = 0; j < k; j++)
            tot += ((i >> j) & 1) ? a[j] : 0;
        cmax(ans, tot);
    }

    cout << ans << endl;
```

```
}
```

27750 「dpO」Matching 任务分配

题目描述

有N个人和N项任务，每项任务都将分配给一个人，并且每个人只会分配一项任务。

给出大小为$N \times N$的矩阵A，其中$A[i][j] = 1$表示第i个人可以做第j项任务，反之亦然。现在求不同的分配任务的方案总数。

对于一个分配方案，只要有一个人的分配的任务不同则被视为不同的方案。

输入格式

第1行：1个整数$N(1 \leq N \leq 21)$，表示人和任务的数量

接下来N行，每行N个整数，描述矩阵A

输出格式

第1行：1个整数，表示总共有多少种不同的任务分配方案。答案对$10^9 + 7$取模。

样例输入

```
21
0 0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 1 0 0 1
1 1 1 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 1 1 0
0 0 1 1 1 1 0 1 1 0 0 1 0 0 1 1 0 0 0 1 1
0 1 1 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 1 1 0
1 1 0 0 1 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0
0 1 1 0 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 0 1
0 1 0 0 0 1 0 1 0 0 0 1 1 1 0 0 1 1 0 1 0
0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1
0 0 1 0 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1 1
0 0 0 0 1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 0 1
0 1 1 0 1 1 0 0 1 1 0 0 0 1 1 1 1 0 1 1 0
0 0 1 0 0 1 1 1 1 0 1 1 0 1 1 1 0 0 0 0 1
0 1 1 0 0 1 1 1 1 0 0 0 1 0 1 1 0 1 0 1 1
1 1 1 1 1 0 0 0 0 1 0 0 1 1 0 1 1 1 0 0 1
0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1
1 0 1 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 0 0 1
0 0 0 1 0 0 1 1 0 1 0 1 0 1 1 0 0 1 1 0 1
0 0 0 0 1 1 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0
1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 0
1 0 0 1 1 0 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0
```

样例输出

```
102515160
```

Solution

我们令$\mathrm{dp}[i][status]$表示安排了前i个人，这i个人占用的**任务**是$status$的时候的方案数

初始情况很简单，我们只需要简单的将$\mathrm{dp}[0][i] = mp[0][i]$即可

随后，对于某个状态$\mathrm{dp}[i][status]$，我们只需要枚举我们的第i个人选择了哪样任务，记为j，然后，我们将$\mathrm{dp}[i-1][status \bigoplus (1 << j)]$加到我们的$\mathrm{dp}[i][status]$即可

这样，我们的总时间复杂度为$O(n^2 2^n)$

注意到，我们的$status$里面，1的数量，实际上已经反应了我们安排人员的数量i了，因此，我们只需要**顺序**枚举$status$的**前置状态**pre即可，因为在上文所述的式子$status \oplus (1 << j)$中，如果j是i**能够选择的**，那么一定有$status \oplus (1 << j) < status$，这样就可以省去枚举人数$i$的一维循环，让我们的时间复杂度降为$O(n2^n)$

Code

```
void solve(){
    int n; cin >> n;
    vector dp(n, vector(1 << n, 0));
    vector mp(n, vector(n, 0));
    for (auto& vec: mp) for (auto& v: vec) cin >> v;

    for (int i = 0; i < n; i++) dp[0][1 << i] = mp[0][i];

    for (int pre = 1; pre < (1 << n) - 1; pre++) {
        auto i = __popcount(pre);
        for (int j = 0; j < n; j++) {
            if (mp[i][j] && !((pre >> j) & 1))
                add(dp[i][pre | (1 << j)], dp[i - 1][pre]);
        }
    }
    cout << dp[n - 1].back() << endl;
}
```

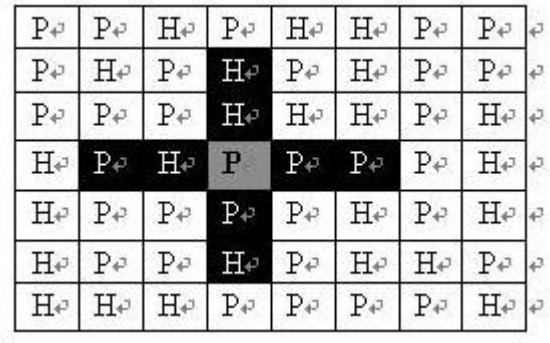
178 [「NOI2001」炮兵阵地]

题目描述

司令部的将军们打算在 $N \times M$ 的网格地图上部署他们的炮兵部队。

一个 $N \times M$ 的地图由 N 行 M 列组成，地图的每一格可能是山地（用 `H` 表示），也可能是平原（用 `P` 表示），如下图。

在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：



如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。

图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

输入格式

第一行包含两个由空格分割开的正整数，分别表示 $N(1 \leq N \leq 100)$ 和 $M(1 \leq M \leq 10)$。

接下来的 N 行，每一行含有连续的 M 个字符，按顺序表示地图中每一行的数据，保证字符仅包含 `P` 与 `H`。

输出格式

一行一个整数，表示最多能摆放的炮兵部队的数量。

样例输入

```
5 4
PHPP
PPHH
PPPP
PHPP
PHHP
```

样例输出

```
6
```

Solution

该题和**互不侵犯**比较类似，唯一的区别在于，该题放置的炮兵能影响**上下两行**

首先，对于山地不能部署炮兵这一限制而言，我们可以将我们的地形状压成$0-1$的形式，其中0代表此处**可以**安放炮兵，反之亦然。当我们枚举到第i行，状态为$status$的时候，如果$status \ \& \ mp_i$不为0，则代表有炮兵安放在了不能安放炮兵的位置

我们使用$\mathrm{dp}[i][now][pre]$代表枚举到第i行，该行的状态为now，**上行**状态为pre的情况下，最多能放置多少个炮兵

其中，now和pre首先得满足不冲突这一关系，而行间冲突，仅在于垂直上下不能同时有炮兵存在，即$now \ \& \ pre$为0即可

随后，我们枚举**上上行**状态$ppre$，在保证$ppre$和now, pre不冲突的前提下，按以下方程进行转移：

$$\mathrm{dp}[i][now][pre] = \max(\mathrm{dp}[i][now][pre], \mathrm{dp}[i-1][pre][ppre] + \mathrm{popcount}(now))$$

最后我们在$\mathrm{dp}[n]$里面寻找最大值作为答案即可

但这样时间复杂度为$O(n2^{3m})$，容易超时

因此，我们可以预处理出所有**单行内合法**的状态，这样就可以通过本题了

此外，注意到对于每个$\mathrm{dp}[i]$转移的时候，我们只用到了$\mathrm{dp}[i-1]$，因此可以使用**滚动数组**优化

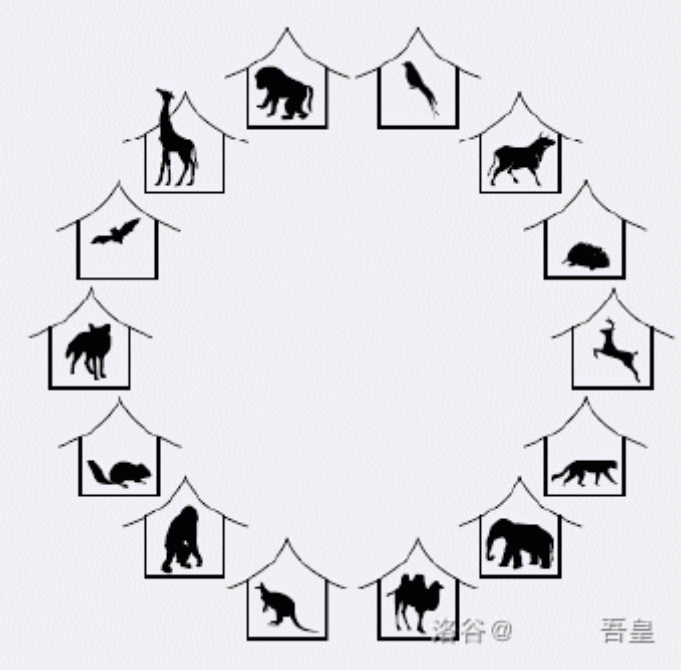
Code

```
void solve(){
    int n, m; cin >> n >> m;
    vector mp(n, 0);
    for (auto &v: mp) for (int i = 0; i < m; i++) {
        char ch; cin >> ch;
        v <<= 1; v += (ch == 'H');
    }

    vector dp(2, vector(1 << m, vector(1 << m, 0)));
    vector status(0, 0);
    for (int i = 0; i < 1 << m; i++)
        if ((i & (i << 1)) || (i & (i >> 2)) || (i & (i >> 1)) || (i & (i >>
2))) continue;
        else status.push_back(i);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < 1 << m; j++) fill(all(dp[i & 1][j]), 0);
        for (auto now: status) for (auto pre: status) {
            if ((now & mp[i]) || (i && (pre & mp[i - 1])) || (now & pre))
continue;
            for (auto ppre: status) {
                if ((now & ppre) || (pre & ppre)) continue;
                cmax(dp[i & 1][now][pre], dp[(i & 1) ^ 1][pre][ppre] +
__popcount(now));
            }
        }
    }
    int ans = 0;
    for (auto &vec: dp[(n - 1) & 1]) cmax(ans, *max_element(all(vec)));
    cout << ans << endl;
}
```

179 「APIO2007」动物园

题目描述

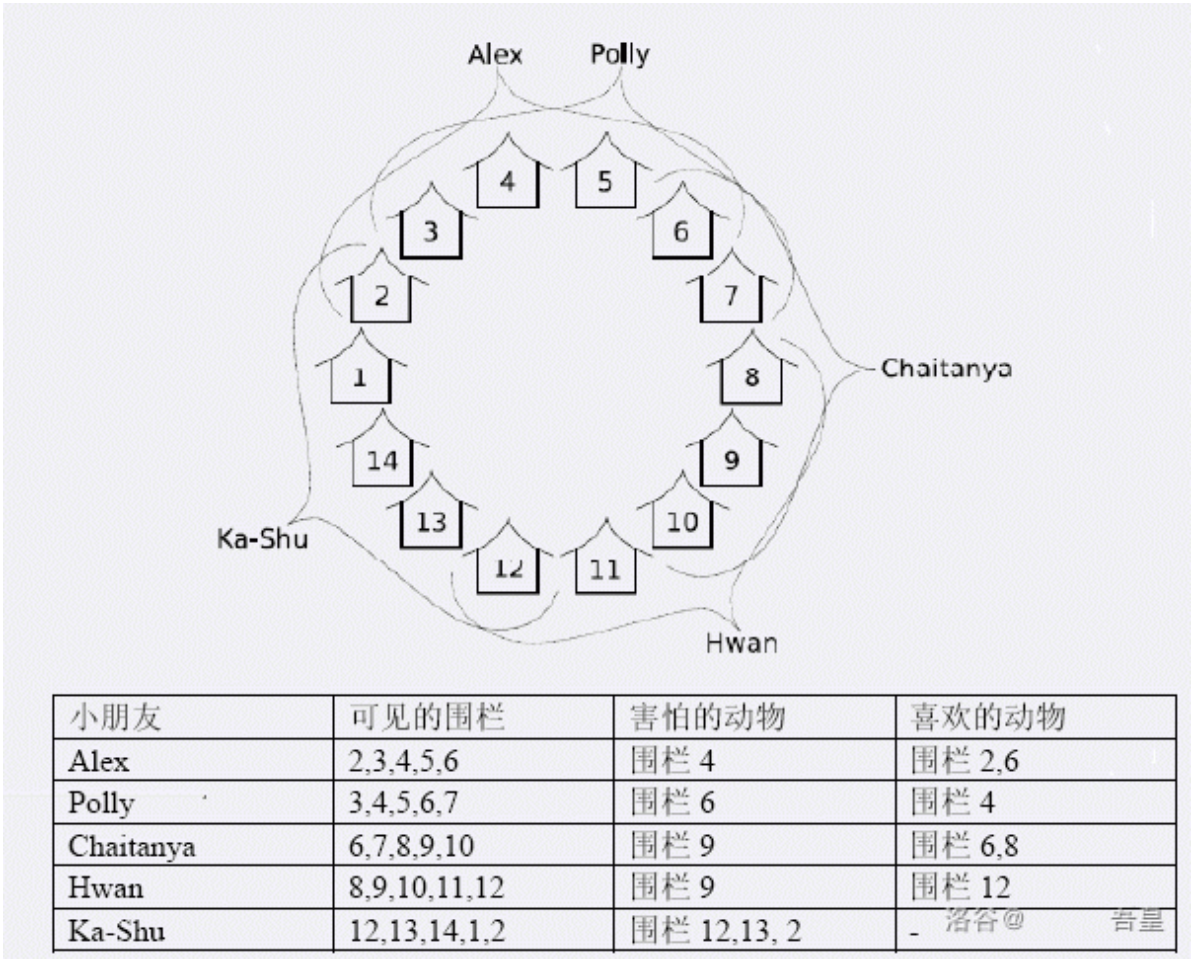
新建的圆形动物园是亚太地区的骄傲。圆形动物园坐落于太平洋的一个小岛上，包含一大圈围栏，每个围栏里有一
种动物。如下图所示：



你是动物园的公共主管。你要做的是，让每个来动物园的人都尽可能高兴。今天有一群小朋友来动物园参观，你希望能让他们在动物园度过一段美好的时光。但这并不是一件容易的事——有的动物有一些小朋友喜欢，有的动物有一些小朋友害怕。如，Alex 喜欢可爱的猴子和考拉，而害怕拥牙齿锋利的狮子。而 Polly 会因狮子有美丽的鬃毛而喜欢它，但害怕有臭味的考拉。你可以选择将一些动物从围栏中移走以使得小朋友不会害怕。但你不能移走太多动物，否则小朋友们就没有动物可看了。每个小朋友站在大围栏圈的外面，可以看到连续的 5 个围栏。你得到了所有小朋友喜欢和害怕的动物信息。当下面两处情况之一发生时，小朋友就会高兴：

- 至少有一个他害怕的动物被移走
- 至少有一个他喜欢的动物没被移走

例如，考虑下图中的小朋友和动物：



小朋友	可见的围栏	害怕的动物	喜欢的动物
Alex	2,3,4,5,6	围栏 4	围栏 2,6
Polly	3,4,5,6,7	围栏 6	围栏 4
Chaitanya	6,7,8,9,10	围栏 9	围栏 6,8
Hwan	8,9,10,11,12	围栏 9	围栏 12
Ka-Shu	12,13,14,1,2	围栏 12,13, 2	-

- 假如你将围栏 4 和 12 的动物移走。Alex 和 Ka-Shu 将很高兴，因为至少有一个他们害怕的动物被移走了。这也会使 Chaitanya 高兴，因为他喜欢的围栏 6 和 8 中的动物都保留了。但是，Polly 和 Hwan 将不高兴，因为他们看不到任何他们喜欢的动物，而他们害怕的动物都还在。这种安排方式使得三个小朋友高兴。
- 现在，换一种方法，如果你将围栏 4 和 6 中的动物移走，Alex 和 Polly 将很高兴，因为他们害怕的动物被移走了。Chaitanya 也会高兴，虽然他喜欢的动物 6 被移走了，他仍可以看到围栏 8 里面他喜欢的动物。同样的 Hwan 也会因可以看到自己喜欢的动物 12 而高兴。唯一不高兴的只有 Ka-Shu。
- 如果你只移走围栏 13 中的动物，Ka-Shu 将高兴，因为有一个他害怕的动物被移走了，Alex, Polly, Chaitanya 和 Hwan 也会高兴，因为他们都可以看到至少一个他们喜欢的动物。所以有 5 个小朋友会高兴。这种方法使得了最多的小朋友高兴。

输入格式

输入的第一行包含两个整数 N，C，用空格分隔。

N 是围栏数（$10 \leq N \leq 10^4$），C 是小朋友的个数（$1 \leq C \leq 5 \times 10^4$）。

围栏按照顺时针的方向编号为 $1, 2, 3, \cdots, N$。

接下来的 C 行，每行描述一个小朋友的信息，以下面的形式给出：
$E, F, L, X_1, X_2, \cdots, X_F, Y_1, Y_2, \cdots, Y_L$。

其中：E 表示这个小朋友可以看到的第一个围栏的编号（$1 \leq E \leq N$），换句话说，该小朋友可以看到的围栏为 E，$E+1$，$E+2$，$E+3$，$E+4$。

注意，如果编号超过 N 将继续从 1 开始算。

如：当 $N = 14$，$E = 13$ 时，这个小朋友可以看到的围栏为 $13, 14, 1, 2$ 和 3。

F 表示该小朋友害怕的动物数。

L 表示该小朋友喜欢的动物数。

围栏 $X_1, X_2, \cdots, X_F$ 中包含该小朋友害怕的动物。

围栏 $Y_1, Y_2, \cdots, Y_L$ 中包含该小朋友喜欢的动物。

$X_1, X_2, \cdots, X_F, Y_1, Y_2, \cdots, Y_L$ 是两两不同的整数，而且所表示的围栏都是该小朋友可以看到的。

小朋友已经按照他们可以看到的第一个围栏的编号从小到大的顺序排好了（这样最小的 E 对应的小朋友排在第一个，最大的 E 对应的小朋友排在最后一个）。

注意可能有多于一个小朋友对应的 E 是相同的。

输出格式

仅输出一个数，表示最多可以让多少个小朋友高兴。

样例输入

```
14 5
2 1 2 4 2 6
3 1 1 6 4
6 1 2 9 6 8
8 1 1 9 12
12 3 0 12 13 2
```

样例输出

```
5
```

Solution

对于本题而言，需要进行状态压缩的**状态**，只能是动物是否移除

然而动物过多，我们没办法一次性全部枚举，观察到我们每个小朋友只会有5**个**相邻的动物，那么，我们考虑用状压表示某个点i，及之后四个动物的状态，即表示区间$[i, i+4]$

注意到，对于某个区间$[i, i+4]$，如果状态固定了，那么处于位置i的所有小朋友是否高兴也随之确定了，那么，我们可以预处理出一个$num[pos][s]$数组，代表处于pos位置的小朋友，在区间$[pos, pos+4]$的状态为s时，是否高兴

对于$num[pos][s]$的预处理较为简单，我们只需要判断位置处于pos小朋友喜欢的动物集合$like$是否没有被全部移除，即$like\ \&\ (\sim i)$，或者说小朋友不喜欢的动物集合fal是否被至少移除一个，即$fal\ \&\ i$，如果两者任意满足一个，那这个小朋友就会高兴，那我们就往$num[pos][s]$里面加上1

对于动态规划的部分，我们能很简单地想出状态，$\mathrm{dp}[i][s]$为处理了前i个位置，位置$[i, i+4]$的状态为s时，前i个位置的高兴小朋友的最大数量

如果我们需要从$\mathrm{dp}[i-1]$这行推导而来，我们就需要在s中删除位置$i+4$的信息，并且补充位置$i-1$的信息，我们假定位置信息从高位到低位按如下方式排布：

$$i+4, i+3, i+2, i+1, i$$

我们使用$s\ \&\ 15$，即可保留区间$[i, i+3]$的信息，而再将其**左移一位**，我们就可以在**最低位枚举**$i-1$的信息了。最后，我们将结果加上$num[i][s]$，即可得到我们$\mathrm{dp}[i][s]$的值，转移方程如下：

$$\mathrm{dp}[i][s] = \max(\mathrm{dp}[i-1][(s\ \&\ 15) << 1], \mathrm{dp}[i-1][((s\ \&\ 15) << 1) + 1]) + num[i][s]$$

但是，对于本题而言，我们的动物结构是**环状**的，也就是说，我们枚举到$\mathrm{dp}[n-3]$这一列时，就需要考虑跨过终点的影响了。我们有一个巧妙的方法解决这个问题：我们手动增加一个**第零位置**，每次将第零位置的**某一个状态**s设为0，其它状态均设为$-inf$，这样就保证了所有后续转移都是基于**第零位置**的；同时，这个第零位置，实际上对应的是区间$[n,1,2,3,4]$，那我们需要最终状态和我们初始状态相等（因为两者在本质上是一个状态），即我们答案会存放在我们的$\mathrm{dp}[n][s]$中。我们枚举第零位置的初始状态s，不断更新答案即可

我们有$32N$个状态，状态转移是$O(1)$的，同时还要进行32次动态规划，因此时间复杂度为$O(2^{10}N)$，足以通过本题

Code

```
void solve(){
    int n, c; cin >> n >> c;
    vector num(n + 1, vector(32, 0));

    for (int i = 0; i < c; i++) {
        int E, F, L; cin >> E >> F >> L;
        int f = 0, l = 0;
        for (int i = 0; i < F; i++) {
            int u; cin >> u; u = (u - E + n) % n;
            f |= (1 << u);
        }
        for (int i = 0; i < L; i++) {
            int u; cin >> u; u = (u - E + n) % n;
            l |= (1 << u);
        }

        for (int i = 0; i < 32; i++) if ((f & i) || ((~i) & l))
            num[E][i]++;
    }

    int ans = 0;
    vector dp(n + 1, vector(32, 0));
    for (int bg = 0; bg < 32; bg++) {
        fill(all(dp[0]), -inf);
        dp[0][bg] = 0;
        for (int i = 1; i <= n; i++) for (int j = 0; j < 32; j++) {
            dp[i][j] = max(dp[i - 1][(j & 15) << 1], dp[i - 1][((j & 15) << 1) |
1]) + num[i][j];
        }
        cmax(ans, dp[n][bg]);
    }

    cout << ans << endl;
}
```