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$\text{dp}[i][j][k]$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$\text{dp}[i-1][j-\text{popcount}(k)][k']$，如果状态合法，则将其直接加到$\text{dp}[i][j][k]$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: $W(100 \le W \le 400)$ 表示桥能承受的最大重量和 $n(1 \le n \le 16)$ 表示队员总数。

接下来 n 行：每行两个数: $t(1 \le t \le 50)$ 表示该队员过桥所需时间和 $w(10 \le w \le 100)$ 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$N(0 \le N \le 2^{60})$，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：

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

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：

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

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$n, m(1 \le n \le 1000, 1 \le m \le 10)$，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$a_i(1 \le a_i \le 1000)$和$k_i(0 \le k_i \le m)$，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
```