第9章 动态规划提高

在本章中，我们将学习区间 DP、树形 DP、状压 DP、计数 DP、数位 DP、概率 DP 和 DP 中常见的优化方法。

9.1 区间 DP

在第 5 章动态规划基础中，我们介绍了动态规划基础和背包等线性动态规划问题，它们一般从初始状态开始，沿着阶段的扩张向某个方向递推，直至计算出目标状态。区间 DP 也是线性动态规划中的一种，它以区间长度作为阶段，使用区间的左右端点描述维度。

例题　石子合并（弱化版）（洛谷 P1755）

设有 n 堆石子排成一排，其编号为 $1,2,\cdots,n$。每堆石子有一定的质量 m_i。现在要将这 n 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻。合并时由于选择的顺序不同，合并的总代价也不相同。试找出一种合理的方法，使总的代价最小，并输出最小代价。

【输入格式】

第一行，一个整数 n。

第二行，n 个整数 m_i。

$1 \le n \le 3 \times 10^2, 1 \le m_i \le 10^3$。

【输出格式】

输出文件仅一个整数，也就是最小代价。

【分析】

若最初的第 l 堆石子和第 r 堆石子被合并成一堆，则说明 $l \sim r$ 之间的每堆石子也已经被合并，这样 l 和 r 才有可能相邻。因此，在任意时刻，任意一堆石子均可以用一个闭区间 $[l,r]$ 来描述，表示这堆石子是由最初的第 $l \sim r$ 堆石子合并而成的，其重量为 $\sum_{i=l}^{r} m_i$。另外，一定存在一个整数 $k\ (l \le k \le r)$，在这堆石子合成之前，先有第 $l \sim k$ 堆石子，第 $k+1 \sim r$ 堆石子被合并成一堆，然后这两堆石子才合并成 $[l,r]$。

对应到动态规划中，就意味着两个长度较小的区间上的信息向一个更长的区间发生了转移，划分点 k 就是转移的决策。自然地，应该把区间长度 len 作为 DP 的阶段。不过，区间长度可以由左端点和右端点表示出，即 $len = r - l + 1$。本着动态规划“选择最小的能覆盖状态空间的维度集合”的思想，可以只用左、右端点表示 DP 的状态。

设状态 $f[l][r]$ 表示把最初的第 l 堆到第 r 堆石子合并成一堆，需要消耗的最少体力。

根据上述分析，可以得到状态转移方程：

$$f[l][r]=\min_{l\le k\le r}(f[l][k]+f[k+1][r])+\sum_{i=l}^{r}m_i$$

初始化为 $\forall l\in[1,n]$，$f[l][l]=0$，其余为负无穷。答案为 $f[1][n]$。

时间复杂度为 $O(n^3)$。

```
template <typename T>
inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}

const int M = int(3e2);
const int inf = 0x3f3f3f3f;

int s[M + 5];
int f[M + 5][M + 5];

void work() {
    int n; scanf("%d", &n);
    memset(f, inf, sizeof f);
    for (int i = 1; i <= n; ++i) scanf("%d", &s[i]), s[i] += s[i - 1], f[i][i] = 0;
    for (int len = 2; len <= n; len++) {
        for (int l = 1; l + len - 1 <= n; l++) {
            int r = l + len - 1;
            for (int k = l; k < r; k++)
                cmin(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s[l - 1]);
        }
    }
    printf("%d\n", f[1][n]);
}
```

例题　[NOI1995] 石子合并（洛谷 P1880）

在一个圆形操场的四周摆放 n 堆石子，现要将石子有次序地合并成一堆，规定每次只能选相邻的 2 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。

试设计出一个算法，计算出将 n 堆石子合并成 1 堆的最小得分和最大得分。

【输入格式】

第一行，一个整数 n。

第二行，n 个整数 a_i。

$1\le n\le 10^2, 0\le a_i\le 20$。

【输出格式】

输出共 2 行，第 1 行为最小得分，第 2 行为最大得分。

【分析】

上一题是在链上进行石子合并，而这题是在环上进行石子合并，问题十分类似，因此我们可以借助上一题的方法对这题进行分析。

面对环上问题，一种十分经典的做法是将环拆成两倍长的链。具体来讲，我们可以根据环构造一个长 $2n$ 的链 $[a_1, a_2, \cdots, a_n, a_1, a_2, \cdots, a_n]$。以最小值为例，设状态 $f[l][r]$ 表示合并了第 $l \sim r$ 堆石子的最小得分，状态转移方程与上一题的相同，那么我们只需要考虑合并区间 $[1, n]$，区间 $[2, n+1]$，…，区间 $[n, 2n-1]$ 的最小得分。在长 $2n$ 的链上进行区间 DP，答案为 $\min\limits_{1 \le i \le n} f[i][i+n-1]$。

```
template <typename T>
inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}

template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(2e2);
const int inf = 0x3f3f3f3f;

int a[M + 5];
int s[M + 5];
int f[M + 5][M + 5];
int g[M + 5][M + 5];

void work() {
    int n; scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]), a[i + n] = a[i];
    memset(f, inf, sizeof f), memset(g, -inf, sizeof g);
    for (int i = 1; i < 2 * n; ++i) s[i] = s[i - 1] + a[i], f[i][i] = g[i][i] = 0;
    for (int len = 1; len <= n; ++len) {
        for (int l = 1; l + len - 1 < 2 * n; l++) {
            int r = l + len - 1;
            for (int k = l; k < r; ++k)
                cmin(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s[l - 1]),
                cmax(g[l][r], g[l][k] + g[k + 1][r] + s[r] - s[l - 1]);
        }
    }
    int mi = inf, mx = -inf;
    for (int i = 1; i <= n; ++i)
        cmin(mi, f[i][i + n - 1]),
        cmax(mx, g[i][i + n - 1]);
    printf("%d\n%d\n", mi, mx);
```

```
33. }
```

例题　　多边形（AcWing283）

“多边形游戏”是一款单人益智游戏。

游戏开始时，给定玩家一个具有 n 个顶点 n 条边（编号 $1 \sim n$）的多边形，如图 所示，其中 $n = 4$。

每个顶点上写有一个整数，每个边上标有一个运算符 +（加号）或运算符 *（乘号）。

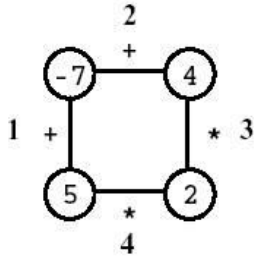


Figure 1. Graphical representation of a polygon

第一步，玩家选择一条边，将它删除。

接下来在进行 $n-1$ 步，在每一步中，玩家选择一条边，把这条边以及该边连接的两个顶点用一个新的顶点代替，新顶点上的整数值等于删去的两个顶点上的数按照删去的边上标有的符号进行计算得到的结果。

下面是进行游戏的全过程。

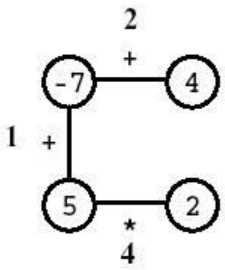


Figure 2. Removing edge **3**

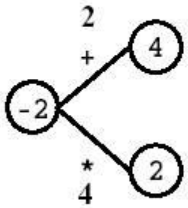


Figure 3. Picking edge **1**

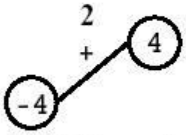


Figure 4. Picking edge **4**

Figure 5. Picking edge **2**

最终，游戏仅剩一个顶点，顶点上的数值就是玩家的得分，上图玩家得分为 0。

请计算对于给定的 n 边形，玩家最高能获得多少分，以及第一步有哪些策略可以使玩家获得最高得分。

【输入格式】

输入包含两行，第一行为整数 n 。 第二行用来描述多边形所有边上的符号以及所有顶

点上的整数，从编号为 1 的边开始，边、点、边…按顺序描述。 其中描述顶点即为输入顶点上的整数，描述边即为输入边上的符号，其中加号用 t 表示，乘号用 x 表示。

$3 \le n \le 50$。数据保证无论玩家如何操作，顶点上的数值均在 $[-32768,32767]$ 之内。

【输出格式】

输出包含两行，第一行输出最高分数。在第二行，将满足得到最高分数的情况下，所有的可以在第一步删除的边的编号从小到大输出，数据之间用空格隔开。

【分析】

可以仿照上一题，将环拆成长度为 $2n$ 的链。由于负数和乘法的存在，所以我们除了维护最大得分外，还要维护最小得分。设状态 $f[l][r][0]$ 表示区间 $[l,r]$ 的最大得分，状态 $f[l][r][1]$ 表示区间 $[l,r]$ 的最小得分，则有状态转移方程

$$f[l][r][0] = \max_{\substack{l \le k < r \\ 0 \le j,l \le 1}} (f[l][k][j] \; op \; f[k+1][r][l])$$

$$f[l][r][1] = \min_{\substack{l \le k < r \\ 0 \le j,l \le 1}} (f[l][k][j] \; op \; f[k+1][r][l])$$

答案则为 $\max\limits_{1 \le i \le n} f[i][i+n-1][0]$。

```
template <typename T>
inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}

template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(1e2);
const int inf = 0x3f3f3f3f;

char op[M + 5];
int a[M + 5];
int f[M + 5][M + 5][2];

void work() {
    int n; scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        getchar();
        scanf("%c %d", &op[i - 1], &a[i]);
        op[i - 1 + n] = op[i - 1];
        a[i + n] = a[i];
        f[i + n][i + n][0] = f[i + n][i + n][1] = f[i][i][0] = f[i][i][1] = a[i];
    }
    for (int len = 2; len <= n; len++) {
```

```
24.         for (int l = 1; l + len - 1 < 2 * n; l++) {
25.             int r = l + len - 1;
26.             f[l][r][0] = -inf, f[l][r][1] = inf;
27.             for (int k = l; k < r; k++) {
28.                 auto fop = op[k] == 't' ? [](int a, int b) {return a + b;} : [](int
a, int b) {return a * b;};
29.                 for (int i = 0; i < 1<<2; i++)
30.                     cmax(f[l][r][0], fop(f[l][k][i>>1], f[k + 1][r][i&1])),
31.                     cmin(f[l][r][1], fop(f[l][k][i>>1], f[k + 1][r][i&1]));
32.             }
33.         }
34.     }
35.     int mx = -inf;
36.     for (int i = 1; i <= n; i++) cmax(mx, f[i][i + n - 1][0]);
37.     printf("%d\n", mx);
38.     for (int i = 1, flag = 0; i <= n; i++) {
39.         if (f[i][i + n - 1][0] == mx) {
40.             if (flag) printf(" ");
41.             flag = 1;
42.             printf("%d", i);
43.         }
44.     }
45.     printf("\n");
46. }
```

9.2 树形 DP

由于树固有的递归性质，树形 DP 一般都是递归进行的。

9.2.1 树形 DP 基础

例题　　没有上司的舞会（AcWing283）

某大学有 n 个职员，编号为 $1,2,\cdots,n$。

他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。

现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 r_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。

所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

【输入格式】

输入的第一行是一个整数 n。

第 2 到第 $n+1$ 行，每行一个整数，第 $i+1$ 行的整数表示 i 号职员的快乐指数 r_i。

第 $n+2$ 到第 $2n$ 行，每行输入一对整数 l,k，代表 k 是 l 的直接上司。

$1 \le n \le 6 \times 10^3, -128 \le r_i \le 127, 1 \le l,k \le n$。

【输出格式】

输出一行一个整数代表最大的快乐指数。

【分析】

设状态 $f[u][0]$ 表示从 u 的子树中选取若干员工参加聚会且 u 不参加聚会的最大快乐指数，状态 $f[u][1]$ 表示从 u 的子树中选取若干员工参加聚会且 u 参加聚会的最大快乐指数。于是有状态转移方程：

$$f[u][0] = \sum_{v \in son(u)} \max(f[v][0], f[v][1])$$

$$f[u][1] = r[u] + \sum_{v \in son(u)} f[v][0]$$

其中，$son(u)$ 表示 u 的子节点集合。设根为 rt，则答案为 $\max(f[rt][0], f[rt][1])$。

时间复杂度为 $O(n)$。

```
const int M = int(6e3);
const int inf = 0x3f3f3f3f;

int r[M + 5];
bool in[M + 5];
vector<int> g[M + 5];
int f[M + 5][2];

void dfs(int u) {
    f[u][0] = 0, f[u][1] = r[u];
    for (const int &v: g[u]) {
        dfs(v);
        f[u][0] += max(f[v][0], f[v][1]);
        f[u][1] += f[v][0];
    }
}

void work() {
    int n; scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &r[i]);
    for (int i = 2, u, fa; i <= n; i++) {
        scanf("%d %d", &u, &fa);
        g[fa].push_back(u);
        in[u] = true;
    }
```

```
26.     int rt = -1;
27.     for (int i = 1; i <= n; i++) if (!in[i]) rt = i;
28.     dfs(rt);
29.     printf("%d\n", max(f[rt][0], f[rt][1]));
30. }
```

9.2.2 树上背包

例题　[CTSC1997] 选课（洛谷 P2014）

在大学里每个学生，为了达到一定的学分，必须从很多课程里选择一些课程来学习，在课程里有些课程必须在某些课程之前学习，如高等数学总是在其它课程之前学习。现在有 n 门功课，每门课有个学分，每门课有一门或没有直接先修课（若课程 a 是课程 b 的先修课即只有学完了课程 a，才能学习课程 b）。一个学生要从这些课程里选择 m 门课程学习，问他能获得的最大学分是多少？

【输入格式】

第一行有两个整数 n,m 用空格隔开。

接下来的 n 行,第 $i+1$ 行包含两个整数 k_i 和 s_i，k_i 表示第 i 门课的直接先修课，s_i 表示第 i 门课的学分。若 $k_i=0$ 表示没有直接先修课。

$1\le n,m\le 3\times 10^2, 1\le k_i\le n, 1\le s_i\le 20$。

【输出格式】

只有一行，选 m 门课程的最大得分。

【分析】

解法一

因为每门课的先修课最多只有一门，所以这 n 门课程构成了森林结构。简便起见，我们可以新建一个 0 号节点作为“没有先修课的课程”的先修课。

设状态 $f[u][i]$ 表示在以 u 的子树中，选取了 i 门课程的最大得分。修完 u 这门课之后，我们可以在 v_j 为根的子树中选修若干门课（记为 c_j），在满足 $\sum_{v_j\in son(u)} c_j=i-1$ 的基础上获得尽量多的学分。

显然有 $f[u][0]=0$, $f[u][1]=s[u]$。当 $i>1$ 时，有状态转移方程

$$f[u][i]=s[u]+\max_{\sum_{j=1}^{|son(u)|}c_j=i-1}\sum_{j=1}^{|son(u)|} f[v_j][c_j]$$

该方程其实是一个分组背包模型。有 $|son(u)|$ 组物品，每组物品有 $i-1$ 个，其中第 j 组的第 k 个物品体积为 k，价值为 $f[v_j][k]$。

时间复杂度为 $O(nm^2)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(3e2);
const int inf = 0x3f3f3f3f;

int n, m;
int a[M + 5];
vector<int> g[M + 5];
int f[M + 5][M + 5];
int backup[M + 5];

void dfs(int u) {
    f[u][0] = 0, f[u][1] = a[u];
    for (const int &v: g[u]) {
        dfs(v);
        memcpy(backup, f[u], sizeof f[u]);
        for (int i = 1; i <= m; i++)
            for (int j = 1; i + j <= m; j++) cmax(f[u][i + j], backup[i] + f[v][j]);
    }
}

void work() {
    scanf("%d %d", &n, &m); ++m;
    for (int i = 1, fa; i <= n; i++) {
        scanf("%d %d", &fa, &a[i]);
        g[fa].push_back(i);
    }
    memset(f, -inf, sizeof f);
    dfs(0);
    printf("%d\n", f[0][m]);
}
```

解法二

记 $sz[u]$ 表示以 u 为根的子树大小。解法一在状态转移时，枚举了许多非法状态，例如 $f[u][sz[u]+1]$。为此，我们可以顺带统计 $sz[u]$，借助 $sz[u]$ 只去枚举合法状态，可以证明，卡紧枚举上下界之后的时间复杂度为 $O(nm)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(3e2);
```

```
const int inf = 0x3f3f3f3f;

int n, m;
int a[M + 5];
vector<int> g[M + 5];
int sz[M + 5];
int f[M + 5][M + 5];
int backup[M + 5];

void dfs(int u) {
    sz[u] = 1, f[u][0] = 0, f[u][1] = a[u];
    for (const int &v: g[u]) {
        dfs(v);
        memcpy(backup, f[u], sizeof f[u]);
        for (int i = 1; i <= sz[u]; i++)
            for (int j = 1; j <= min(m - i, sz[v]); j++)
                cmax(f[u][i + j], backup[i] + f[v][j]);
        sz[u] += sz[v];
    }
}

void work() {
    scanf("%d %d", &n, &m); ++m;
    for (int i = 1, fa; i <= n; i++) {
        scanf("%d %d", &fa, &a[i]);
        g[fa].push_back(i);
    }
    memset(f, -inf, sizeof f);
    dfs(0);
    printf("%d\n", f[0][m]);
}
```

这类题目被称为树形背包，它实际上是背包与树形 DP 的结合。通常，我们也像线性 DP 一样将体积作为第二维状态。在状态转移时，我们要处理的实际上是一个分组背包问题。

例题　[JSOI2018] 潜入行动（洛谷 P2014）

外星人又双叒叕要攻打地球了，外星母舰已经向地球航行！这一次，JYY 已经联系好了黄金舰队，打算联合所有 JSOIer 抵御外星人的进攻。

在黄金舰队就位之前，JYY 打算事先了解外星人的进攻计划。现在，携带了监听设备的特工已经秘密潜入了外星人的母舰，准备对外星人的通信实施监听。

外星人的母舰可以看成是一棵 n 个节点、$n-1$ 条边的无向树，树上的节点用 $1,2,\cdots,n$ 编号。JYY 的特工已经装备了隐形模块，可以在外星人母舰中不受限制地活动，

可以神不知鬼不觉地在节点上安装监听设备。

如果在节点 u 上安装监听设备，则 JYY 能够监听与 u 直接相邻所有的节点的通信。换言之，如果在节点 u 安装监听设备，则对于树中每一条边 (u,v)，节点 v 都会被监听。特别注意放置在节点 u 的监听设备并不监听 u 本身的通信，这是 JYY 特别为了防止外星人察觉部署的战术。

JYY 的特工一共携带了 k 个监听设备，现在 JYY 想知道，有多少种不同的放置监听设备的方法，能够使得母舰上所有节点的通信都被监听？为了避免浪费，每个节点至多只能安装一个监听设备，且监听设备必须被用完。

【输入格式】

输入第一行包含两个整数 n,k，表示母舰节点的数量 n 和监听设备的数量 k。 接下来 $n-1$ 行，每行两个整数 u,v，表示树中的一条边。

$1 \le n \le 10^5, 1 \le k \le \min(n, 10^2), 1 \le u, v \le n$。

【输出格式】

输出一行，表示满足条件的方案数。因为答案可能很大，你只需要输出答案 $mod\ 10^9 + 7$ 的余数即可。

【分析】

设状态 $f[u][i][0/1][0/1]$ 为在以 u 为根的子树中，放置了 i 个监听设备，节点 u 是否放置监听设备，节点 u 是否被监听到的方案数。对于当前子节点 v，考虑 $f[u][i][0/1][0/1]$ 和 $f[v][j][0/1][0/1]$ 对 $f[u][i+j][0/1][0/1]$ 的贡献，下面进行分情况讨论。

对于状态 $f[u][i+j][0][0]$，因为节点 u 未放置监听设备且未被监听，所以节点 v 不可以放置监听设备且必须已经被监听，于是有

$$f[u][i+j][0][0] += f[u][i][0][0] \times f[v][j][0][1]$$

对于状态 $f[u][i+j][0][1]$，节点 u 未放置监听设备且被监听，此时有两种情况。

第一种情况是在考虑节点 v 之前节点 u 已经被监听，此时节点 v 只需要保证被监听，是否放置监听设备则无所谓，因此贡献可以表示为

$$s_1 = f[u][i][0][1] \times (f[v][j][0][1] + f[v][j][1][1])$$

第二种情况是在考虑节点 v 之前节点 u 未被监听，此时节点 v 需要放置监听设备使得节点 u 被监听，且节点 v 自身也要被监听，因此贡献可以表示为

$$s_2 = f[u][i][0][0] \times f[v][j][1][1]$$

于是有 $f[u][i+j][0][1] += s_1 + s_2$。

对于状态 $f[u][i+j][1][0]$，因为节点 u 放置了监听设备且未被监听，所以节点 v 在以 v 为根的子树中是否被监听则无所谓，但是不可以放置监听设备，于是有

$$f[u][i+j][1][0] += f[u][i][1][0] \times (f[v][j][0][0] + f[v][j][0][1])$$

对于状态 $f[u][i+j][1][1]$，节点 u 放置了监听设备且被监听，此时有两种情况。

第一种情况是在考虑节点 v 之前节点 u 已经被监听，此时节点 v 没有任何限制，因此贡献可以表示为

$$s_3 = f[u][i][1][1] \times (f[v][j][0][0] + f[v][j][0][1] + f[v][j][1][0] + f[v][j][1][1])$$

第二种情况是在考虑节点 v 之前节点 u 未被监听，此时节点 v 在以 v 为根的子树中是否被监听则无所谓，但是必须要放置监听设备使得节点 u 被监听，因此贡献可以表示为

$$s_4 = f[u][i][1][0] \times (f[v][j][1][0] + f[v][j][1][1])$$

于是有 $f[u][i+j][1][1] += s_3 + s_4$。

树上背包时使用 sz 数组卡紧枚举上下界，时间复杂度为 $O(nk)$。

```
typedef long long ll;

const int M = int(1e5);
const int N = int(1e2);
const int mod = int(1e9) + 7;

int n, k;
int sz[M + 5];
vector<int> g[M + 5];
int backup[N + 5][2][2];
int f[M + 5][N + 5][2][2];

void dfs(int u, int fa) {
    sz[u] = 1, f[u][0][0][0] = 1, f[u][1][1][0] = 1;
    for (const int &v: g[u]) if (v != fa) {
        dfs(v, u);
        memcpy(backup, f[u], sizeof f[u]);
        memset(f[u], 0, sizeof f[u]);
        for (int i = 0; i <= min(k, sz[u]); i++) {
            for (int j = 0; j <= min(k - i, sz[v]); j++) {
                (f[u][i + j][0][0] +=
                    (ll)backup[i][0][0] * f[v][j][0][1] % mod) %= mod;
                (f[u][i + j][0][1] +=
                    ((ll)backup[i][0][1] * (f[v][j][0][1] + f[v][j][1][1])
                    + (ll)backup[i][0][0] * f[v][j][1][1]) % mod) %= mod;
```

```
26.                 (f[u][i + j][1][0] +=
27.                 (ll)backup[i][1][0] * (f[v][j][0][0] + f[v][j][0][1]) % mod) %= mod;
28.                 (f[u][i + j][1][1] +=
29.                     (backup[i][1][1] * ((ll)f[v][j][0][0] + f[v][j][0][1]
30.                         + f[v][j][1][0] + f[v][j][1][1])
31.                     + (ll)backup[i][1][0] * (f[v][j][1][0]
32.                         + f[v][j][1][1])) % mod) %= mod;
33.             }
34.         }
35.         sz[u] += sz[v];
36.     }
37. }
38.
39. void work() {
40.     scanf("%d %d", &n, &k);
41.     for (int i = 2, u, v; i <= n; i++) {
42.         scanf("%d %d", &u, &v);
43.         g[u].push_back(v), g[v].push_back(u);
44.     }
45.     dfs(1, 0);
46.     printf("%d\n", (f[1][k][0][1] + f[1][k][1][1]) % mod);
47. }
48.
```

9.2.3 换根 DP

例题　[POI2008] STA-Station（洛谷 P3478）

给定一个 n 个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

一个结点的深度之定义为该节点到根的简单路径上边的数量。

【输入格式】

第一行有一个整数，表示树的结点个数 n。 接下来 $n-1$ 行，每行两个整数 u,v，表示存在一条连接 u,v 的边。

$1 \le n \le 10^6, 1 \le u,v \le n$。

【输出格式】

输出一行一个整数表示你选择的结点编号。如果有多个结点符合要求，输出任意一个即可。

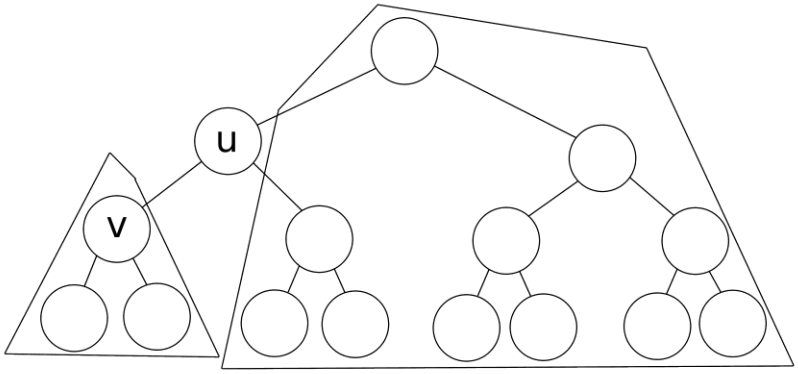
【分析】

这是一道经典的换根 DP。换根 DP 又被称为二次扫描，通常不会指定根节点，并且根

节点的变化对答案的计算有一定影响，需要进行两次 DFS，第一次 DFS 统计树上的深度、点权和等信息，第二次 DFS 再运用动态规划计算以每个节点作为根节点的答案。

第一次 DFS 以节点 1 为根，记 $sz[u]$ 表示以 u 为根的子树大小，设状态 $f[u]$ 表示以节点 u 为根的子树中，所有节点与节点 u 的距离之和，则有状态转移方程 $f[u] = \sum_{v \in son(u)} f[v] + sz[v]$。

设状态 $h[u]$ 表示以 u 为根时，所有节点深度的之和，容易发现 $h[1] = f[1]$。第二次 DFS 时，考虑如何统计换根的答案，即如何通过 $h[u]$ 计算出 $h[v]$。



如图，$h[u]$ 的计算可以划分为两部分，分别是以 v 为根的子树对 $h[u]$ 的贡献和其余部分对 $h[u]$ 的贡献。我们可以把 $h[v]$ 的计算划分为两部分，分别是以 v 为根的子树对 $h[v]$ 的贡献和其余部分对 $h[v]$ 的贡献。其中以 v 为根的子树对 $h[v]$ 的贡献为 $f[v]$，其余部分对 $h[v]$ 的贡献为 $h[u] - (f[v] + sz[v]) + sz[1] - sz[v]$，因此有状态转移方程

$$h[v] = f[v] + h[u] - (f[v] + sz[v]) + sz[1] - sz[v]$$

化简一下可以得到 $h[v] = h[u] + sz[1] - 2sz[v]$。

时间复杂度为 $O(n)$。

```
typedef long long ll;

const int M = int(1e6);

int sz[M + 5];
ll f[M + 5];
ll h[M + 5];
vector<int> g[M + 5];

void dfs1(int u, int fa) {
    sz[u] = 1;
    for (const int &v: g[u]) if (v != fa) {
        dfs1(v, u);
```

```
14.         sz[u] += sz[v];
15.         f[u] += f[v] + sz[v];
16.     }
17. }
18.
19. void dfs2(int u, int fa) {
20.     for (const int &v: g[u]) if (v != fa) {
21.         h[v] = h[u] + sz[1] - 2 * sz[v];
22.         dfs2(v, u);
23.     }
24. }
25.
26. void work() {
27.     int n; scanf("%d", &n);
28.     for (int i = 2, u, v; i <= n; i++) {
29.         scanf("%d %d", &u, &v);
30.         g[u].push_back(v), g[v].push_back(u);
31.     }
32.     dfs1(1, 0);
33.     h[1] = f[1]; dfs2(1, 0);
34.     printf("%d\n", int(max_element(h + 1, h + n + 1) - h));
35. }
36.
```

例题　积蓄程度（AcWing287）

有一个树形的水系，由 $n-1$ 条河道和 n 个交叉点组成。

我们可以把交叉点看作树中的节点，编号为 $1 \sim n$，河道则看作树中的无向边。

每条河道都有一个容量，连接 x 与 y 的河道的容量记为 $c(x,y)$。

河道中单位时间流过的水量不能超过河道的容量。

有一个节点是整个水系的发源地，可以源源不断地流出水，我们称之为源点。

除了源点之外，树中所有度数为 1 的节点都是入海口，可以吸收无限多的水，我们称之为汇点。

也就是说，水系中的水从源点出发，沿着每条河道，最终流向各个汇点。

在整个水系稳定时，每条河道中的水都以单位时间固定的水量流向固定的方向。

除源点和汇点之外，其余各点不贮存水，也就是流入该点的河道水量之和等于从该点流出的河道水量之和。

整个水系的流量就定义为源点单位时间发出的水量。

在流量不超过河道容量的前提下，求哪个点作为源点时，整个水系的流量最大，输出这

个最大值。

【输入格式】

输入第一行包含整数 T，表示共有 T 组测试数据。

每组测试数据，第一行包含整数 n。

接下来 $n-1$ 行，每行包含三个整数 x,y,z，表示 x,y 之间存在河道，且河道容量为 z。 节点编号从 1 开始。

$1 \le n,z \le 2 \times 10^5, 1 \le x,y \le n$。

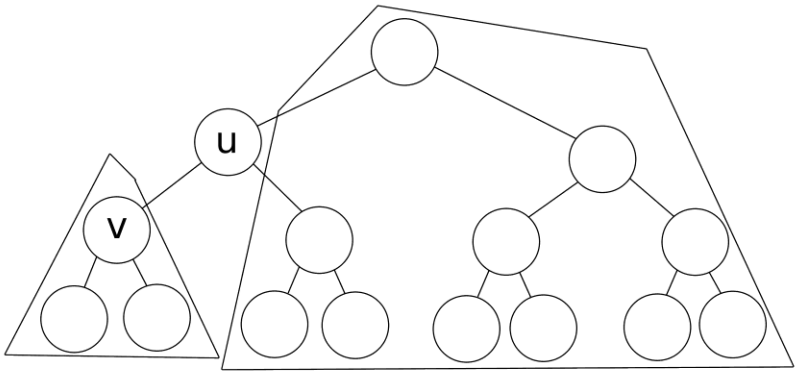
【输出格式】

每组数据输出一个结果，每个结果占一行。

【分析】

为了简化逻辑判断，我们可以先选定一个度大于 1 的节点 rt 作为根，第一次 DFS 以节点 rt 为根，设状态 $f[u]$ 表示以节点 u 为根的子树中，以 u 作为源点的发水量，则有状态转移方程 $f[u]=\sum_{v \in son(u)} \min(z,f[v])$。

设状态 $h[u]$ 表示以 u 为根时，整个水系的流量，容易发现 $h[rt]=f[rt]$。第二次 DFS 时，考虑如何统计换根的答案，即如何通过 $h[u]$ 计算出 $h[v]$。



如图，$h[u]$ 的计算可以划分为两部分，分别是以 v 为根的子树对 $h[u]$ 的贡献和其余部分对 $h[u]$ 的贡献。我们可以把 $h[v]$ 的计算划分为两部分，分别是以 v 为根的子树对 $h[v]$ 的贡献和其余部分对 $h[v]$ 的贡献。其中以 v 为根的子树对 $h[v]$ 的贡献为 $f[v]$，其余部分对 $h[v]$ 的贡献为 $\min(w,h[u]-\min(w,f[v]))$，因此有状态转移方程

$$h[v]=f[v]+\min(w,h[u]-\min(w,f[v]))$$

时间复杂度为 $O(n)$。

```
1. typedef long long ll;
2.
3. const int M = int(2e5);
```

```cpp
const ll linf = 0x3f3f3f3f3f3f3f3f;

ll f[M + 5];
ll h[M + 5];
vector<pair<int, int>> g[M + 5];

void dfs1(int u, int fa) {
    f[u] = 0;
    bool leaf = true;
    for (const auto &[v, w]: g[u]) if (v != fa) {
        dfs1(v, u);
        leaf = false;
        f[u] += min((ll)w, f[v]);
    }
    if (leaf) f[u] = linf;
}

void dfs2(int u, int fa) {
    for (const auto &[v, w]: g[u]) if (v != fa) {
        h[v] = (f[v] == linf ? 0 : f[v]) + min((ll)w, h[u] - min((ll)w, f[v]));
        dfs2(v, u);
    }
}

void work() {
    int n; scanf("%d", &n);
    if (n == 1) return printf("0\n"), void();
    else if (n == 2) {
        int w; scanf("%*d %*d %d", &w);
        return printf("%d\n", w), void();
    }
    for (int i = 1; i <= n; i++) g[i].clear();
    for (int i = 2, u, v, w; i <= n; i++) {
        scanf("%d %d %d", &u, &v, &w);
        g[u].push_back({v, w}), g[v].push_back({u, w});
    }
    int rt = -1;
    for (int i = 1; i <= n; i++) if (g[i].size() > 1) {rt = i; break;}
    dfs1(rt, 0);
    h[rt] = f[rt]; dfs2(rt, 0);
    printf("%lld\n", *max_element(h + 1, h + n + 1));
}
```

9.3 状压 DP

在线性 DP 中，动态规划的过程是随着“阶段”的增长，在每个状态维度上不断扩展的。在任意时刻，已经求出最优解的状态与尚未求出最优解的状态在各维度上的分界点组成了 DP 扩展的“轮廓”。对于某些问题，我们需要在动态规划的“状态”中记录一个集合，保存这个“轮廓”的详细信息，以便进行状态转移。若集合大小不超过 n，集合中每个元素都是小于 k 的自然数，则我们可以把这个集合看作一个 n 位 k 进制数，以一个 $[0,k^n-1]$ 之间的十进制整数的形式作为 DP 状态的一维。这种把集合转化为整数记录在 DP 状态中的一类算法，被称为状态压缩动态规划算法。

9.3.1 状压 DP

例题　　最短 Hamilton 路径（AcWing91）

给定一张 n 个点的带权无向图，点从 $0\sim n-1$ 标号，求起点 0 到终点 $n-1$ 的最短 Hamilton 路径。

Hamilton 路径的定义是从 0 到 $n-1$ 不重不漏地经过每个点恰好一次。

【输入格式】

第一行输入整数 n。

接下来 n 行每行 n 个整数，其中第 i 行第 j 个整数表示点 i 到 j 的距离（记为 $a[i,j]$）。

对于任意的 x,y,z，数据保证 $a[x,x]=0$，$a[x,y]=a[y,x]$ 并且 $a[x,y]+a[y,z]\geq a[x,z]$。

$1\leq n\leq 20, 0\leq a[i,j]\leq 10^7$。

【输出格式】

输出一个整数，表示最短 Hamilton 路径的长度。

【分析】

设状态 $f[s][u]$ 表示遍历点集的状态为 s，当前处于节点 u 的最短路。

考虑遍历点集的状态 s 的表示方法。由于每个点要么已遍历，要么未遍历，只有两种状态，因此我们可以使用二进制来表示 s，i 号点拥有二进制数中 2^i 的位权。例如 0 号点已遍历，1 号点已遍历，2 号点未遍历，那么 s 可以表示为二进制 011_B。同理，若 $s=100_B$ 则表示 0 号点未遍历，1 号点未遍历，2 号点已遍历。

当遍历点集状态为 s 且当前处于节点 u 时，考虑走向未遍历节点 v 后的状态变化，

不难发现状态由 $f[s][u]$ 变为状态 $f[s|(1\ll v)][v]$，于是有状态转移方程

$$f[s|(1\ll v)][v]=\min_{u\in s,v\notin s}(f[s][u]+a[u][v])$$

共有 $n2^n$ 个状态，每个状态的计算复杂度为 $O(n)$，因此总的时间复杂度为 $O(n^2 2^n)$。

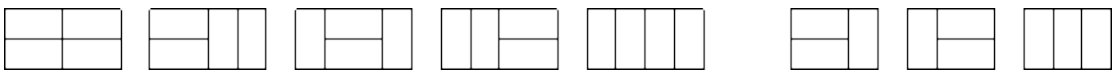
```
#include <bits/stdc++.h>
using namespace std;

const int inf = 0x3f3f3f3f;

template <typename T>
inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}

const int M = int(2e1);

int a[M][M];
int f[1<<M][M];

void work() {
    int n; scanf("%d", &n);
    for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) scanf("%d", &a[i][j]);
    for (int i = 0; i < (1<<n); i++) memset(f[i], inf, sizeof(f[i][0]) * n);
    f[1][0] = 0;
    for (int i = 0; i < (1<<n); i++) {
        for (int j = 0; j < n; j++) if (i >> j & 1) {
            for (int k = 0; k < n; k++) if (!(i >> k & 1)) {
                cmin(f[i|(1<<k)][k], f[i][j] + a[j][k]);
            }
        }
    }
    printf("%d\n", f[(1<<n) - 1][n - 1]);
}

int main() {
    work();
    return 0;
}
```

例题　蒙德里安的梦想（AcWing291）

求把 $n\times m$ 的棋盘分割成若干个 1×2 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。

当输入用例 $n=0,m=0$ 时，表示输入终止，且该用例无需处理。

$1 \le n,m \le 11$。

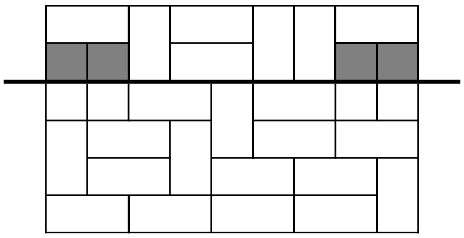
【输出格式】

每个测试用例输出一个结果，每个结果占一行。

【分析】

解法一

对于任意一种方案，考虑以某一行为界，把整个棋盘横着分成两半。



如图所示，在上半部分的最后一行中：

1. 每个灰色背景的部分都是一个竖着的 1×2 长方形的上半部分，决定了下一行必须补全其下半部分。
2. 其余部分对下一行没有影响。下一行既可以在连续两个位置安排一个横着的 1×2 长方形，也可以让某个位置作为一个竖着的 1×2 长方形的上半部分。

综上所述，我们可以把“行号”作为 DP 的“阶段”，把上半部分不断向下扩展，直至确定整个棋盘的分割方法。为了描述上半部分，我们可以使用一个 m 位二进制数，其中第 k 位为 1 表示第 k 列是一个竖着的 1×2 长方形的上半部分，第 k 位为 0 表示其他情况。

设状态 $f[i][j]$ 表示考虑了前 i 行，第 i 行的放置方案为 j 的方案数。

第 $i-1$ 行的放置方案 k 能转移到第 i 行的方案方案 j，当且仅当：

1. $j\&k=0$。这保证了每个数字 1 的下方必须是数字 0。
2. $j|k$ 的二进制表示中，每一段连续的 0 都必须有偶数个。这些 0 代表若干个横着的 1×2 长方形，奇数个 0 无法分割成这种状态。

我们可以在 DP 前预处理出 $[0,2^m-1]$ 内所有满足“二进制表示下每一段连续的 0 都有偶数个”的整数，记录在集合 S 中。于是有状态转移方程：

$$f[i,j]=\sum_{\substack{j\,\&k=0\\ j|k\in S}} f[i-1,k]$$

初始值为 $\forall j\in S\ f[0][s]=1$，其余均为 0。答案为 $f[n-1][0]$。时间复杂度为 $O(4^m n)$。

```
typedef long long ll;

const int M = 11;

int n, m;
bool ok[1<<M];
ll f[M + 5][1<<M];

void work() {
    for (int i = 0; i < (1<<m); i++) {
        bool odd = false;
        for (int j = 0; j < m; j++) {
            if (i >> j & 1) {
                if (odd) break;
                else odd = false;
            }
            else odd ^= 1;
        }
        ok[i] = !odd;
    }
    for (int i = 0; i < (1<<m); i++) f[0][i] = ok[i];
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < (1<<m); j++) {
            f[i][j] = 0;
            for (int k = 0; k < (1<<m); k++)
                if (!(j & k) && ok[j | k])
                    f[i][j] += f[i - 1][k];
        }
    }
    printf("%lld\n", f[n - 1][0]);
}
```

解法二

在解法一中，我们使用了填表法，其中枚举了部分无效状态。因此，我们可以使用刷表法进行优化。

虽然理论时间复杂度与解法一一样，都为 $O(4^m n)$，但在代码的实际表现上，解法二比解法一快了将近 7 倍。

```
typedef long long ll;
```

```
2.
3. const int M = 11;
4.
5. int n, m;
6. bool ok[1<<M];
7. ll f[M + 5][1<<M];
8.
9. void work() {
10.     for (int i = 0; i < (1<<m); i++) {
11.         bool odd = false;
12.         for (int j = 0; j < m; j++) {
13.             if (i >> j & 1) {
14.                 if (odd) break;
15.                 else odd = false;
16.             }
17.             else odd ^= 1;
18.         }
19.         ok[i] = !odd;
20.     }
21.     for (int i = 0; i < (1<<m); i++) f[0][i] = ok[i];
22.     for (int i = 1; i < n; i++) memset(f[i], 0, sizeof(ll) * (1<<m));
23.     for (int i = 0; i < n - 1; i++) {
24.         for (int j = 0; j < (1<<m); j++) if (f[i][j]) {
25.             for (int k = 0; k < (1<<m); k++) if (!(j & k) && ok[j | k]) f[i + 1][k]
+= f[i][j];
26.         }
27.     }
28.     printf("%lld\n", f[n - 1][0]);
29. }
```

解法三

继续在解法二的基础上优化。在解法二中，对于每个 $j \in [0,2^m-1]$ 都要枚举 $k \in [0,2^m-1]$，其中包含了大量的非法转移。为此，我们可以预处理出每个状态 j 其下一行的所有合法状态 k，这样在 DP 阶段可以避免非法转移。

时间复杂度为 $O(4^m n)$。

```
1. typedef long long ll;
2.
3. const int M = 11;
4.
5. int n, m;
6. bool ok[1<<M];
7. ll f[M + 5][1<<M];
8. vector<int> nx[1<<M];
```

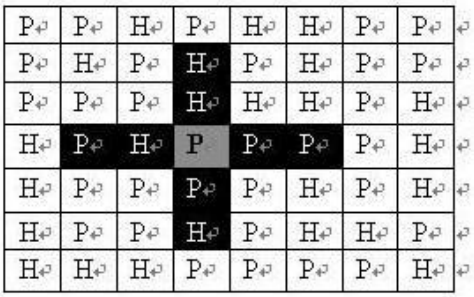
```
 9.
10. void work() {
11.     for (int i = 0; i < (1<<m); i++) {
12.         bool odd = false;
13.         for (int j = 0; j < m; j++) {
14.             if (i >> j & 1) {
15.                 if (odd) break;
16.                 else odd = false;
17.             }
18.             else odd ^= 1;
19.         }
20.         ok[i] = !odd;
21.     }
22.     for (int i = 0; i < (1<<m); i++) {
23.         nx[i].clear();
24.         for (int j = 0; j < (1<<m); j++)
25.             if (!(i & j) && ok[i | j]) nx[i].push_back(j);
26.     }
27.     for (int i = 0; i < (1<<m); i++) f[0][i] = ok[i];
28.     for (int i = 1; i < n; i++) memset(f[i], 0, sizeof(ll) * (1<<m));
29.     for (int i = 0; i < n - 1; i++) {
30.         for (int j = 0; j < (1<<m); j++) if (f[i][j]) {
31.             for (const int &k: nx[j]) f[i + 1][k] += f[i][j];
32.         }
33.     }
34.     printf("%lld\n", f[n - 1][0]);
35. }
```

例题　[NOI2001] 炮兵阵地（洛谷 P2704）

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】

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。

当输入用例 $n=0,m=0$ 时，表示输入终止，且该用例无需处理。

$1 \le n,m \le 11$。

【输出格式】

第一行包含两个由空格分割开的正整数，分别表示 n 和 m。

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。

$1 \le n \le 10^2, 1 \le m \le 10$，保证字符仅包含 P 与 H。

【分析】

因为每个位置能否放置炮兵与它上面两行对应位置上是否放置炮兵有关，所以在向第 i 行的状态转移时，需要知道第 $i-1$ 行和第 $i-2$ 行的状态。我们把每一行的状态看作一个 m 位的二进制数，其中第 p 位为 1 表示该行第 p 列放置了炮兵，为 0 则表示没有放置炮兵。

我们在 DP 前预处理出集合 S，存储“相邻两个 1 的距离不小于 3”的所有 m 位二进制数，这些二进制数代表每一行中的两个炮兵的距离不能小于 3，实践可以发现集合 S 的大小最大为 60。

设 $count(x)$ 表示 m 位二进制数 x 中 1 的个数，$valid(i,x)$ 表示 m 位二进制数 x 属于集合 S，并且 x 中的每个 1 对应在地图的第 i 行中的位置都是平原。

设状态 $f[i][j][k]$ 表示第 $i-1$ 行的状态是 j，第 i 行的状态时 k 时，前 i 行最多摆放多少个炮兵，于是有状态转移方程

$$f[i][j][k] = \max_{\substack{j\&l=0,j\&k=0 \\ valid(i,j),valid(i-1,k)}} (f[i-1][k] + count(j))$$

可以发现集合 S 的大小约为时间复杂度为 $O(n|S|^3)$。

```cpp
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(1e2);
const int N = int(1e1);
const int inf = 0x3f3f3f3f;

int a[M + 5];
char s[N + 5];
vector<int> v;
int cnt[1<<N];
int f[2][1<<N][1<<N];

void work() {
    int n, m; scanf("%d %d", &n, &m);
    for (int i = 0; i < n; i++) {
        scanf("%s", s);
        for (int j = 0; j < m; j++) a[i] = a[i] * 2 + (s[j] == 'P');
    }
    for (int i = 0; i < (1<<m); i++) {
        if ((i&(i>>1)) || (i&(i>>2))) continue;
        v.push_back(i);
        cnt[i] = __builtin_popcount(i);
    }
    if (n == 1) {
        int mx = -inf;
        for (const int &i: v) if ((i && a[0]) == i) cmax(mx, cnt[i]);
        return printf("%d\n", mx), void();
    }
    for (const int &i: v) if ((i & a[0]) == i) {
        for (const int &j: v)
            if ((j & a[1]) == j && !(i & j)) f[1][i][j] = cnt[i] + cnt[j];
    }
    for (int i = 1, u; i < n - 1; i++) {
        u = i & 1;
        for (const int &j: v) for (const int &k: v) f[u ^ 1][j][k] = -inf;
        for (const int &j: v) {
            for (const int &k: v) if (f[u][j][k] >= 0) {
                for (const int &l: v)
                    if ((l & a[i + 1]) == l && !(l & j) && !(l & k))
                    cmax(f[u ^ 1][k][l], f[u][j][k] + cnt[l]);
            }
        }
    }
```

```
45.     int mx = -inf;
46.     for (const int &i: v) for (const int &j: v) cmax(mx, f[(n - 1) & 1][i][j]);
47.     printf("%d\n", mx);
48. }
```

9.3.2 高维前缀和

高维前缀和（SOSDP，Sum Over Subsets Dynamic Programming）它一般是用来解决子集类的求和问题。

回忆前缀和的递推式，设数组 a 表述原数组，数组 s 表示前缀和数组。

在 1 维中有 $s[i]=s[i-1]+a[i]$；

在 2 维中有 $s[i][j]=s[i-1][j]+s[i][j-1]-s[i-1][j-1]+a[i][j]$；

在 3 维中有 $s[i][j][k]=s[i-1][j][k]+s[i][j-1][k]+s[i][j][k-1]-s[i-1][j-1]-s[i-1][j][k-1]-s[i][j-1][k-1]+s[i-1][j-1][k-1]+a[i][j][k]$。

我们发现，随着维数 k 的增加，容斥操作不能视为 $O(1)$，它与 k 成幂函数关系。通过容斥操作实现的高维前缀和的时间复杂度为 $O(|状态个数|\times 2^k)$。

另一种求前缀和的方式如下。这种方式可以理解成二维前缀和是数组在求完关于第一个维度的前缀和，然后再求它关于第二个维度的前缀和。

```
1. for (int i = 1; i <= n; i++)
2.     for (int j = 1; j <= m; j++)
3.         a[i][j] += a[i][j - 1]
4. for (int i = 1; i <= n; i++)
5.     for (int j = 1; j <= m; j++)
6.         a[i][j] += a[i-1][j]
```

使用同样的方法在求三维以上前缀和的时候，就体现出这种写法在求解高维前缀和上的优越性了。

```
 1. for (int i = 1; i <= n; i++)
 2.     for (int j = 1; j <= m; j++)
 3.         for (int k = 1; k <= p; k++)
 4.             a[i][j][k] += a[i - 1][j][k]
 5. for (int i = 1; i <= n; i++)
 6.     for (int j = 1; j <= m; j++)
 7.         for (int k = 1; k <= p; k++)
 8.             a[i][j][k] += a[i][j - 1][k]
 9. for (int i = 1; i <= n; i++)
10.     for (int j = 1; j <= m; j++)
11.         for (int k = 1; k <= p; k++)
12.             a[i][j][k] += a[i][j][k - 1]
```

这样的话无需借助容斥原理，求高维前缀和的复杂度变为$O(|高维空间容量| \times k)$，可以处理 k 稍大一些的情况。

高维前缀和一般是用来解决子集类的求和问题。所谓子集求和是指有 $n\,(1 \le n \le 20)$ 个物品，确定每个物品的选取与否可以表示一个集合，那么这 n 个物品最多可以表示 2^n 个集合。现在对每个集合都给定个权值，然后查询某个集合以及其包含的所有子集的权值和。

以 $n = 2$ 的情况为例。二维数组 $a[2][2]$ 表示选取关系第一个维度的数字为 0 表示不选第一件物品，第一个维度的数字为 1 表示选第一个物品。同理第二个维度的数字为 0 表示不选第二件物品，第二个维度的数字为 1 表示选第二个物品。

我们用 $s[2][2]$ 表示两个物品选取关系下子集的和，于是有

$$s[0][0] = a[0][0]$$

$$s[0][1] = a[0][0] + a[0][1]$$

$$s[1][0] = a[0][0] + a[1][0]$$

$$s[1][1] = a[0][0] + a[0][1] + a[1][0] + a[1][1]$$

可以发现如果要求某个集合以及其子集的和，其实就是它对应的二维前缀和。这个结论推广到 k 维同样是成立的，因此对于子集求和类问题，可以使用高维前缀和解决，时间复杂度为 $O(2^k k)$。

例题　膜法记录（牛客小白月赛 23A）

牛牛最近在玩一款叫做《膜法记录》的游戏，这个游戏的机制是这样的：

在一局游戏中，所有的敌人都排布在一个 n 行 m 列的网格中，牛牛指挥着他的魔法少女对敌人进行攻击。

攻击有两种类型：行 blast，列 blast

行 blast 能消灭一整行的敌人，列 blast 能消灭一整列的敌人。

牛牛总共能够释放 a 次行 blast， b 次列 blast 给定某局游戏的初始局面，请问牛牛能否将敌人全歼？

【输入格式】

第一行包含一个正整数 T，表示测试数据组数，接下来是 T 组测试数据。

每组测试数据的第一行有四个正整数 n, m, a, b。

接下来有 n 行，每行是一个长度为 m 的字符串，第 i 行第 j 列的字符如果是 $*$ 则说明这里有一个敌人；如果是 . 则说明这里没有。

要么 $1 \le T \le 10^5, 1 \le n \le 5, 1 \le \sum m \le 10^5$；要么 $T = 1, 1 \le n \le 20, 1 \le m \le 10^5$。$1 \le$

$a \le n, 1 \le b \le m$。

【输出格式】

对每组测试数据输出一行，如果能消灭所有的敌人，就输出 yes，否则输出 no

【分析】

由于 n 很小，所以我们可以考虑二进制枚举行 blast 的释放情况，即在哪些行释放了行 blast。此外，如果我们计算出当行 blast 释放情况为 i 时整张网格有多少个不含敌人的列，记为 $cnt[i]$，那么当行 blast 释放情况为 i 时需要的列 blast 次数为 $m - cnt[i]$。

考虑如何计算 $cnt[i]$。我们可以先统计出所有列的二进制状态，记 $cnt2[i]$ 表示列状态为 i 的个数，那么不难发现 $cnt[i] = \sum_{j\&i=j} cnt2[j]$，即对 $cnt2$ 做一遍子集求和就可以得到 cnt。

时间复杂度为 $n(m + 2^n)$。

```
const int M = 20;
const int N = int(1e5);

char s[M + 5][N + 5];
int cnt[1<<M];

void work() {
    int n, m, a, b; scanf("%d %d %d %d", &n, &m, &a, &b);
    for (int i = 0; i < n; i++) scanf("%s", s[i]);
    memset(cnt, 0, sizeof(cnt[0]) * (1<<n));
    for (int j = 0; j < m; j++) {
        int state = 0;
        for (int i = 0; i < n; i++) state = state * 2 + (s[i][j] == '*');
        cnt[state]++;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < (1<<n); j++) if (j >> i & 1) cnt[j] += cnt[j ^ (1<<i)];
    }
    for (int i = 0; i < (1<<n); i++) {
        if (__builtin_popcount(i) <= a && m - cnt[i] <= b) {
            puts("yes");
            return;
        }
    }
    puts("no");
}
```

9.4 数据结构优化 DP

例题　清理班次 2（AcWing296）

农夫约翰雇佣他的 n 头奶牛帮他进行牛棚的清理工作。

他将全天分为了很多个班次，其中第 M 个班次到第 E 个班次（包括这两个班次）之间必须都有牛进行清理。

这 n 头牛中，第 i 头牛可以从第 a_i 个班次工作到第 b_i 个班次，同时，它会索取 c_i 的佣金。

请你安排一个合理的清理班次，使得 $[M,E]$ 时间段内都有奶牛在清理，并且所需支付给奶牛的报酬最少。

【输入格式】

第 1 行：包含三个整数 n,M 和 E。

第 $2\cdots n+1$ 行：第 $i+1$ 行包含三个整数 a_i,b_i,c_i。

$1\le n\le 10^4, 0\le M,E\le 86399, M\le a_i\le b_i\le E, 0\le c_i\le 5\times 10^5$。

【输出格式】

输出一个整数，表示所需的最少佣金。

如果无法做到在要求时间段内都有奶牛清理，则输出 -1。

【分析】

设状态 $f[i]$ 表示覆盖了区间 $[M,i]$ 的最小代价。

把所有班次按照右端点 b_i 排序，顺序扫描这些班次。设当前班次为 $[a_i,b_i]$，价格为 c_i，则有状态转移方程

$$f[b_i]=\min_{a_i-1\le x<b_i} f[x]+c_i$$

初始值为 $f[L-1]=0$，其余值为正无穷，答案为 $\min\limits_{b_i\ge E} f[b_i]$。

在这个状态转移方程中，需要查询 f 在区间 $[a_i-1,b_i]$ 上的最小值，同时 f 会不断发生更新，这是一个带有修改操作的区间最值问题，需要用线段树维护数组 f。

时间复杂度为 $O(nlogn)$。

```
1. template <typename T>
2. inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}
3.
4. typedef long long ll;
5.
6. const int M = int(1e4);
```

```
 7. const ll linf = 0x3f3f3f3f3f3f3f3f;
 8.
 9. int a[M + 5], b[M + 5], c[M + 5];
10. int id[M + 5];
11. int d[M * 2 + 5], len;
12.
13. struct segTree {
14.     ll mi[(2 * M + 2) * 4 + 5];
15.
16.     int lc(int k) {return k<<1;}
17.     int rc(int k) {return k<<1|1;}
18.     void push_up(int k) {mi[k] = min(mi[lc(k)], mi[rc(k)]);}
19.
20.     void build(int k, int l, int r) {
21.         if (l == r) return mi[k] = linf, void();
22.         int mid = (l + r) >> 1;
23.         build(lc(k), l, mid);
24.         build(rc(k), mid + 1, r);
25.         push_up(k);
26.     }
27.
28.     void update(int k, int l, int r, int a, ll b) {
29.         if (l == r) return cmin(mi[k], b), void();
30.         int mid = (l + r) >> 1;
31.         if (a <= mid) update(lc(k), l, mid, a, b);
32.         else update(rc(k), mid + 1, r, a, b);
33.         push_up(k);
34.     }
35.
36.     ll query(int k, int l, int r, int a, int b) {
37.         if (l >= a && r <= b) return mi[k];
38.         int mid = (l + r) >> 1; ll res = linf;
39.         if (a <= mid) cmin(res, query(lc(k), l, mid, a, b));
40.         if (mid < b)  cmin(res, query(rc(k), mid + 1, r, a, b));
41.         return res;
42.     }
43. } tr;
44.
45. int tofind(int x) {
46.     return lower_bound(d + 1, d + len + 1, x) - d;
47. }
48.
49. void work() {
50.     int n, m, e; scanf("%d %d %d", &n, &m, &e);
```

```
    d[++len] = m - 1; d[++len] = e;
    for (int i = 1; i <= n; i++) {
        scanf("%d %d %d", &a[i], &b[i], &c[i]), id[i] = i;
        d[++len] = a[i] - 1; d[++len] = b[i];
    }
    sort(id + 1, id + n + 1, [&](int x, int y) {return b[x] < b[y];});
    sort(d + 1, d + len + 1); len = unique(d + 1, d + len + 1) - (d + 1);
    tr.build(1, 1, len);
    tr.update(1, 1, len, tofind(m - 1), 0);
    for (int i = 1, u; i <= n; i++) {
        u = id[i];
        ll mi = tr.query(1, 1, len, tofind(a[u] - 1), len);
        tr.update(1, 1, len, tofind(b[u]), mi + c[u]);
    }
    ll mi = tr.query(1, 1, len, tofind(e), len);
    printf("%lld\n", mi == linf ? -1 : mi);
}

```

例题　赤壁之战（AcWing297）

给定一个长度为 n 的序列 a，求 a 有多少个长度为 m 的严格递增子序列。

【输入格式】

第一行包含整数 T，表示共有 T 组测试数据。

每组数据，第一行包含两个整数 n 和 m。第二行包含 n 个整数，表示完整的序列 a。

$1 \le T \le 10^2, 1 \le m \le n \le 10^3, \sum n \times m \le 10^7, |a| \le 10^9$。

【输出格式】

每组数据输出一个结果，每个结果占一行。

输出格式为 Case #x: y，x 为数据组别序号，从 1 开始，y 为结果。

由于答案可能很大，请你输出对 10^9+7 取模后的结果。

【分析】

设状态 $f[i][j]$ 表示在序列 a 的前 j 个数中，以 $a[j]$ 结尾且长度为 i 的严格递增子序列的个数，则有状态转移方程

$$f[i][j] = \sum_{\substack{k<j \\ a[k]<a[j]}} f[i-1][k]$$

我们先写出枚举 k 的暴力代码

```
const int mod = int(1e9) + 7
memset(f, 0, sizeof f);
a[0] = -inf
```

```
f[0][0] = 1
for (int i = 1; i <= m; ++i)
    for (int j = 1; j <= n; j++)
        for (int k = 0; k < j; k++)
            if (a[k] < a[j])
                f[i][j] = (f[i][j] + f[i - 1][k]) % mod
int ans = 0;
for (int i = 1; i <= n; i++) ans = (ans + f[m][i]) % mod;

```

其中，共有 nm 个状态，每个状态的转移复杂度为 $O(n)$，总时间复杂度为 $O(n^2m)$，显然是不可接受的，考虑如何优化转移复杂度。

在内层循环 j 和 k 进行时，可以把外层循环变量 i 看作定值。我们可以发现，当 j 增加 1 时，k 的取值范围从 $0 \le k < j$ 变为 $0 \le k < j+1$，也就是只多了 $k = j$ 这 1 个新决策。所以，我们需要维护一个支持如下操作的决策候选集合，其中每个决策可以用一个二元组 $(a_k, f[i-1][k])$ 来存储。

1. 插入一个新的决策。具体来说，在 j 增加 1 前，把 $(a_j, f[i-1][j])$ 加入集合。
2. 给定一个值 a_j，查询满足 $a_k < a_j$ 的二元组对应的 $f[i-1][k]$ 的和。

这是一个经典的“二维数点”问题，我们可以把序列 a 中的元素离散化到区间 $[2, n+1]$ 之间，然后在区间 $[1, n+1]$ 上建立树状数组，初始值全为 0。设 a_i 离散化后的值为 $val(a_i)$。

1. 对于插入决策的操作，就把 $val(a_k)$ 位置上的值增加 $f[i-1.k]$。
2. 对于查询操作，就在树状数组中计算 $[1, val(a_j)-1]$ 的前缀和。

综上所述，我们使用树状数组来维护前缀和，就把 DP 的时间复杂度优化到了 $O(nmlogn)$。

```
const int M = int(1e3);
const int mod = int(1e9) + 7;
const int inf = 0x3f3f3f3f;

int a[M + 5], d[M + 5], len;
int f[2][M + 5];

struct TA {
    int a[M + 5], n;
    void init(int _n) {n = _n; memset(a, 0, sizeof(a[0]) * (n + 1));}
    int lowbit(int n) {return n & -n;}
    void add(int p, int x) {while (p <= n) (a[p] += x) %= mod, p += lowbit(p);}
```

```
13.     int ask(int p) {int s = 0; while (p) (s += a[p]) %= mod, p -= lowbit(p); return
s;}
14.     int ask(int l, int r) {return ask(r) - ask(l - 1);}
15. } tr[2];
16.
17. int tofind(int x) {
18.     return lower_bound(d + 1, d + len + 1, x) - d;
19. }
20.
21. void work() {
22.     int n, m; scanf("%d %d", &n, &m);
23.     for (int i = 1; i <= n; i++) scanf("%d", &a[i]), d[i] = a[i];
24.     a[0] = d[len = n + 1] = -inf;
25.     sort(d + 1, d + len + 1); len = unique(d + 1, d + len + 1) - (d + 1);
26.     for (int i = 0; i <= n; i++) a[i] = tofind(a[i]);
27.     tr[0].init(len); memset(f[0], 0, sizeof f[0]); f[0][0] = 1;
28.     for (int i = 1, u; i <= m; i++) {
29.         u = i & 1;
30.         tr[u].init(len);
31.         for (int j = i; j <= n; j++) {
32.             tr[u ^ 1].add(a[j - 1], f[u ^ 1][j - 1]);
33.             f[u][j] = tr[u ^ 1].ask(a[j] - 1);
34.         }
35.     }
36.     printf("%lld\n", accumulate(f[m & 1] + m, f[m & 1] + n + 1, 0ll) % mod);
37. }
```

例题　修剪草坪（AcWing1087）

在一年前赢得了小镇的最佳草坪比赛后，FJ 变得很懒，再也没有修剪过草坪。

现在，新一轮的最佳草坪比赛又开始了，FJ 希望能够再次夺冠。

然而，FJ 的草坪非常脏乱，因此，FJ 只能够让他的奶牛来完成这项工作。

FJ 有 n 只排成一排的奶牛，编号为 1 到 n。

每只奶牛的效率是不同的，奶牛 i 的效率为 E_i。

编号相邻的奶牛们很熟悉，如果 FJ 安排超过 K 只编号连续的奶牛，那么这些奶牛就会罢工去开派对。

因此，现在 FJ 需要你的帮助，找到最合理的安排方案并计算 FJ 可以得到的最大效率。

注意，方案需满足不能包含超过 k 只编号连续的奶牛。

【输入格式】

第一行：空格隔开的两个整数 n 和 K；

第二到 $n+1$ 行：第 $i+1$ 行有一个整数 E_i。

$1 \le n \le 10^5, 0 \le E_i \le 10^9$。

【输出格式】

共一行，包含一个数值，表示 FJ 可以得到的最大的效率值。

【分析】

设数组 $s[i]$ 为数组 $E[i]$ 的前缀和，状态 $f[i]$ 表示考虑了前 i 头奶牛，且最后选择了第 i 头奶牛的最大效率，则有状态转移方程

$$f[i] = \max_{1 \le i-j \le k} (f[j+1] + s[i] - s[j])$$

对状态转移方程进行简单变形得到

$$f[i] = \max_{1 \le i-j \le k} (f[j+1] - s[j]) + s[i]$$

可以看到，对于 $f[i]$ 我们只需要求出 $\max_{1 \le i-j \le k} (f[j+1] - s[j])$，把 $f[j+1] - s[j]$ 看作一个整体，于是可以使用单调队列均摊 $O(1)$ 解决。

时间复杂度为 $O(n)$。

```
typedef long long ll;

const int M = int(1e5);

ll s[M + 5];
ll f[M + 5];
int q[M + 5];

void work() {
    int n, k; scanf("%d %d", &n, &k);
    for (int i = 1; i <= n; i++) scanf("%lld", &s[i]), s[i] += s[i - 1];
    int l = 1, r = 0;
    for (int i = 0; i <= k; i++) {
        f[i] = s[i];
        while (l <= r && f[q[r]] - s[q[r] + 1] <= f[i] - s[i + 1]) --r;
        q[++r] = i;
    }
    for (int i = k + 1; i <= n; i++) {
        while (l <= r && i - (q[l] + 2) + 1 > k) ++l;
        f[i] = f[q[l]] - s[q[l] + 1] + s[i];
        while (l <= r && f[q[r]] - s[q[r] + 1] <= f[i] - s[i + 1]) --r;
        q[++r] = i;
    }
```

```
24.     printf("%lld\n", *max_element(f + 1, f + n + 1));
25. }
```

例题　理想的正方形（AcWing1091）

有一个 $a \times b$ 的整数组成的矩阵，现请你从中找出一个 $n \times n$ 的正方形区域，使得该区域所有数中的最大值和最小值的差最小。

【输入格式】

第一行为三个整数，分别表示 a, b, n 的值；

第二行至第 $a+1$ 行每行为 b 个非负整数，表示矩阵中相应位置上的数。

$1 \le a, b \le 10^3, 1 \le n \le \min(a, b, 10^2)$。矩阵中的所有数都不超过 10^9。

【输出格式】

输出仅一个整数，为 $a \times b$ 矩阵中所有“ $n \times n$ 正方形区域中的最大整数和最小整数的差值”的最小值。

【分析】

设 $mx[i][j]$ 表示以坐标 (i, j) 作为矩阵右下角的矩阵中的元素最大值，$mi[i][j]$ 表示以坐标 (i, j) 作为矩阵右下角的矩阵中的元素最小值，那么答案为 $\min\limits_{\substack{n \le i \le a \\ n \le j \le b}} (mx[i][j] - mi[i][j])$。

以求 $mx[i][j]$ 为例。设 $s[i][j]$ 表述原矩阵的元素，$t[i][j] = \max\limits_{0 \le j-k < n} s[i][k]$。借助二位前缀和的思想，通过枚举行，对列使用单调队列求解滑动窗口的方法可以求出 $t[i][j]$。然后再把 $t[i][j]$ 作为处理对象，枚举列，对行使用单调队列求解滑动窗口的方法即可求出 $mx[i][j]$。

时间复杂度为 $O(ab)$。

```
 1. const int M = int(1e3);
 2. const int inf = 0x3f3f3f3f;
 3.
 4. int a, b, n;
 5. int s[M + 5][M + 5];
 6. int t[M + 5][M + 5];
 7. int q[M + 5];
 8. int mx[M + 5][M + 5];
 9. int mi[M + 5][M + 5];
10.
11. void getRes(int res[][M + 5], function<bool(int, int)> cmp) {
12.     for (int i = 1; i <= a; i++) {
13.         int l = 1, r = 0;
```

```
        for (int j = 1; j <= b; j++) {
            while (l <= r && j - q[l] + 1 > n) l++;
            while (l <= r && !cmp(s[i][q[r]], s[i][j])) r--;
            q[++r] = j;
            t[i][j] = s[i][q[l]];
        }
    }

    for (int j = 1; j <= b; j++) {
        int l = 1, r = 0;
        for (int i = 1; i <= a; i++) {
            while (l <= r && i - q[l] + 1 > n) l++;
            while (l <= r && !cmp(t[q[r]][j], t[i][j])) r--;
            q[++r] = i;
            res[i][j] = t[q[l]][j];
        }
    }
}

void work() {
    scanf("%d %d %d", &a, &b, &n);
    for (int i = 1; i <= a; i++) for (int j = 1; j <= b; j++) scanf("%d", &s[i][j]);
    getRes(mx, [](int a, int b) {return a > b;});
    getRes(mi, [](int a, int b) {return a < b;});
    int ans = inf;
    for (int i = n; i <= a; i++)
        for (int j = n; j <= b; j++)
            ans = min(ans, mx[i][j] - mi[i][j]);
    printf("%d\n", ans);
}
```

9.5 计数 DP

计数 DP 强调“不重不漏”，通常需要精确的划分和整体性的计算。

例题　Gerald and Giant Chess（CodeForces - 559C）

给定一个 $h \times w$ 的棋盘，棋盘上只有 n 个格子是黑色的，其他格子都是白色的。

在棋盘左上角有一个卒，每一步可以向右或向下移动一格，并且不能移动到黑色格子中。

求这个卒从左上角移动到右下角，一共有多少种路线。

【输入格式】

第一行包含三个整数 h,w,n。

接下来 n 行，每行包含两个整数 x,y，描述一个黑色格子位于 x 行 y 列。

数据保证左上角和右下角的格子都是白色的。

$1 \le h,w \le 10^5, 1 \le n \le 2 \times 10^3$。

【输出格式】

输出一个整数表示结果对 10^9+7 取模后的值。

【分析】

棋盘上的白色格子数量巨大，而黑色格子最多只有 2000 个，我们应该考虑如何依靠黑色格子进行计数。我们知道从左上角走到右下角的路线总数是 C_{h+w-2}^{h-1}，若再求出从左上角走到右下角经过了至少一个黑色格子的路线数量，两者相减就得到了只经过白色格子的路线数。

把所有黑色格子按照行、列坐标递增的顺序排序。特别地，我们假设左上角是第 0 个黑色格子，右下角是第 $n+1$ 个黑色格子。设第 i 个黑色格子在第 x_i 行第 y_i 列。设状态 $f[i]$ 表示从左上角走到排序后的第 i 个黑色格子，并且途中不经过其他黑色格子的路线数，则有状态转移方程

$$f[i] = C_{x_i-y_i-2}^{x_i-1} - \sum_{j=0}^{i-1} f[j] \times C_{x_i-x_j+y_i-y_j}^{x_i-x_j}，\text{其中 } x_i \ge x_j, y_i \ge y_j$$

上式的含义是，枚举 j 作为从左上角到 (x_i,y_i) 经过的第一个黑色格子，也就是从左上角 (x_j,y_j) 不能经过其他黑色格子，路线数为 $f[j]$。从 (x_j,y_j) 到 (x_i,y_i) 随便走，路线数就是组合数。等式中求和符号的部分求出了从左上角到 (x_i,y_i)，途中经过至少一个黑色格子的路线数，用总数减掉，就得到了 $f[j]$。

通过限制 j 作为经过的第一个黑色格子，相当于找到了一个基准点，围绕这个基准点构造一个不可划分的整体，以避免子问题之间的重叠。既然需要求出从左上角到 (x_i,y_i) 途中经过至少一个黑色格子的路线数，就枚举第一个经过的黑色格子 j，使左上角到 j 构成一个整体，让这段路程只能经过白色格子，不能再进行拆分。因为第一个经过的黑色格子不同，所以不同的 j 对应的路线肯定不会重复。而 j 又取遍了 i 之前的所有黑色格子，所以路线也不会遗漏。结合乘法原理和加法原理，我们就得到了状态转移方程中求和符号的部分。

时间复杂度为 $O(n^2+h+w)$。

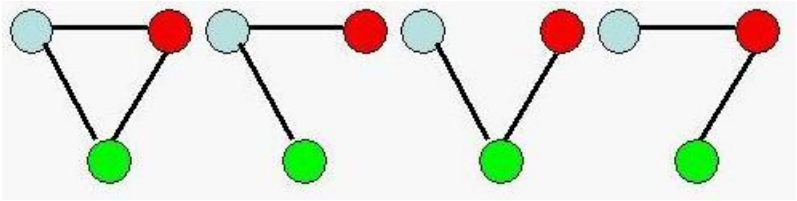
```
1. typedef long long ll;
2.
3. const int M = int(2e3);
4. const int N = int(2e5);
5. const int mod = int(1e9) + 7;
6.
7. int x[M + 5], y[M + 5], id[M + 5];
8. int inv[N + 5], fac[N + 5], invfac[N + 5];
```

```
int f[M + 5];

void init() {
    fac[0] = fac[1] = invfac[0] = invfac[1] = inv[1] = 1;
    for (int i = 2; i <= N; i++) {
        inv[i] = (ll)(mod - mod / i) * inv[mod % i] % mod;
        fac[i] = (ll)fac[i - 1] * i % mod;
        invfac[i] = (ll)invfac[i - 1] * inv[i] % mod;
        assert((ll)fac[i] * invfac[i] % mod == 1);
    }
}

int C(int n, int m) {
    return (ll)fac[n] * invfac[m] % mod * invfac[n - m] % mod;
}

void work() {
    int n, m, c; scanf("%d %d %d", &n, &m, &c);
    for (int i = 1; i <= c; i++) scanf("%d %d", &x[i], &y[i]), id[i] = i;
    x[0] = 1, y[0] = 1;
    ++c; x[c] = n, y[c] = m, id[c] = c;
    sort(id + 1, id + c + 1, [&](int a, int b) {
        if (x[a] != x[b]) return x[a] < x[b];
        return y[a] < y[b];
    });
    f[0] = 1;
    for (int i = 1, u; i <= c; i++) {
        u = id[i];
        f[i] = C(x[u] + y[u] - 2, x[u] - 1);
        for (int j = 1, v; j < i; j++) {
            v = id[j];
            if (x[u] >= x[v] && y[u] >= y[v])
                f[i] -= (ll)f[j] * C(x[u] - x[v] + y[u] - y[v], x[u] - x[v]) % mod;
            if (f[i] < 0) f[i] += mod;
        }
    }
    printf("%d\n", f[c]);
}

int main() {
    init();
    work();
    return 0;
}
```

例题　连通图（AcWing307）

求 n 个节点的无向连通图有多少个，节点有标号，编号为 $1 \sim n$。

例如下列图示，三个节点的无向连通图共 4 个。



【输入格式】

输入包含多组测试数据。每组数据包含一个整数 n。当输入为 0 时，表示输入终止。

$1 \leq n \leq 50$。

【输出格式】

每组测试数据输出一个结果，每个结果占一行。

【分析】

一个连通图不容易进行划分，而一个不连通的无向图则很容易划分成更少的两部分。所以我们可以考虑求出 n 个点的无向图总数。减去 n 个点的不连通无向图的数量，就是 n 个点的连通无向图个数。

n 个点的无向图总数显然是 $2^{\frac{n(n-1)}{2}}$。一张不连通的无向图必定是由若干个连通块构成。我们可以枚举标号为 1 的节点所在的连通块包含的节点个数 k，从 $2 \sim n$ 这 $n-1$ 个节点中选出 $k-1$ 个，与 1 号节点一起构成大小为 k 的连通块。显然，我们有 C_{n-1}^{k-1} 种选法。剩余的 $n-k$ 个节点构成任意无向图，有 $2^{(n-k)(n-k-1)/2}$ 种方法。

综上所述，设 $f[i]$ 表示第 i 个节点的无向连通图个数，状态转移方程为

$$f[i] = 2^{i(i-1)/2} - \sum_{j=1}^{i-1} f[j] \times C_{i-1}^{j-1} \times 2^{(i-j)(i-j-1)/2}$$

时间复杂度为 $O(n^2)$。

此外，需要注意的是答案可能会很大，所以我们可以使用 python 进行编码。

```
1. M = 50
2.
3. c = []
4. c.append([1])
5. for i in range(1, M + 1):
6.     c.append([1])
7.     for j in range(1, i):
8.         c[-1].append(c[-2][j - 1] + c[-2][j])
```

```
    c[-1].append(1)

f = []
f.append(1)
for i in range(1, M + 1):
    f.append(1<<((i*(i-1))//2))
    for j in range(1, i):
        f[-1] -= f[j] * c[i - 1][j - 1] * (1<<((i-j)*(i-j-1)//2))

while True:
    n = int(input())
    if n == 0:
        break
    print(f[n])
```

9.6 数位 DP

数位 DP 往往是给定一个闭区间 $[l,r]$，求区间中满足某种条件的数的个数，而这个区间可能很大，简单的暴力代码如下

```
int ans = 0;
for (int i = l; i <= r; i++)
    if (check(i)) ans++;
```

我们发现，若区间长度超过 10^8，这样暴力枚举就会超时了，而数位 DP 则可以解决此问题。实际上，数位 DP 就是把数位看作阶段，按照数位从高到低的顺序一位一位地进行 DP。

显然，区间 $[l,r]$ 的答案可以转化为区间 $[1.r]$ 的答案与区间 $[1,l-1]$ 的答案的差值。因此原问题可以转化为求区间 $[1,n]$ 的答案。

假设 $n=213$，那么我们可以从百位开始枚举，百位上数字的可能情况有 0,1,2。因为是要统计区间 $[1,n]$ 的答案，所以我们不能让枚举的数超过上界 213。当百位枚举到 0 或 1 时，十位可以枚举 $0\sim9$，这是因为此时枚举的百位已经比百位的上界 2 小了，因此后面的数位无论如何取值都不可能超过上界。而当百位刚好枚举到百位上界 2 时，十位只能枚举 $0\sim1$，此时如果十位枚举到 1，那么个位只能枚举 $0\sim3$，因此我们需要用一个变量 lim 来记录当前枚举的数字是否卡在数位上界。此外，百位枚举 0 相当于此时我们枚举的数最多是两位数，如果十位继续枚举 0，那就相当于在枚举个位数。不过，这样可能会带来一个问题，那就是前导零的问题。在有些情况下，前导零可能会对答案的计算产生影响，为此我们可以用一个变量 $lead$ 来记录当前是否是前导零。

通常，数位 DP 可以使用记忆化搜索实现，具有较为统一模板。

```
// 20 是数位最大长度
```

```
int num[20];
// 不同题目状态不同
// f[i][j]表示在没有数位上界限制的条件下，数字高位状态为 j 的 i 位数字的个数
int f[20][state];

int dfs(int pos, int state, bool lim, bool lead) {
    // 递归边界，最低位是 0，pos==-1 则说明枚举完了一个数
    if (pos == -1) return 1;
    // 记忆化
    if (!lim && !lead && ~f[pos][state]) return f[pos][state];
    // 计算当前数位能填的数字的上界
    int up = (lim ? num[pos] : 9), res = 0;
    for (int i = 0; i <= up; i++) {
        res += dfs(pos - 1, /* 状态转移 */, lim && i == up, lead && i == 0);
    }
    // 记录状态
    if (!lim && !lead) f[pos][state] = res;
    return res;
}

int cal(int n) {
    int pos = 0;
    // 数位分解到 pos 数字
    while (n) {
        num[pos++] = n % 10;
        n /= 10;
    }
    return dfs(pos - 1, /* 初始状态 */, true, true);
}

int main() {
    int l, r; scanf("%d %d", &l, &r);
    printf("%d\n", cal(r) - cal(l - 1));
    return 0;
}
```

例题　不要 62（洛谷 1085）

杭州人称那些傻乎乎粘嗒嗒的人为 62 （音：laoer）。

杭州交通管理局经常会扩充一些的士车牌照，新近出来一个好消息，以后上牌照，不再含有不吉利的数字了，这样一来，就可以消除个别的士司机和乘客的心理障碍，更安全地服务大众。

不吉利的数字为所有含有 4 或 62 的号码。例如：62315,73418,88914 都属于不吉利号

码。但是，61152 虽然含有 6 和 2 ，但不是连号，所以不属于不吉利数字之列。

你的任务是，对于每次给出的一个牌照号区间 $[n,m]$，推断出交管局今后又要实际上给多少辆新的士车上牌照了。

【输入格式】

输入包含多组测试数据，每组数据占一行。 每组数据包含一个整数对 n 和 m。 当输入一行为“0 0”时，表示输入结束。

$1 \le n \le m \le 10^9$。

【输出格式】

对于每个整数对，输出一个不含有不吉利数字的统计个数，该数值占一行位置。

【分析】

对于不包含 4 的要求，我们在枚举数位的时候跳过 4 即可。对于数位中不能出现连续 62 的要求，设状态 $f[pos][0/1]$ 表示当前枚举到第 pos 位，前面一位枚举的不是/是 6 的数的个数。

时间复杂度为 $O(logm)$。

```
1. int f[11][2];
2. int num[11];
3.
4. int dfs(int pos, bool pre, bool lim) {
5.     if (pos == -1) return 1;
6.     if (!lim && ~f[pos][pre]) return f[pos][pre];
7.     int up = (lim ? num[pos] : 9), res = 0;
8.     for (int i = 0; i <= up; i++) {
9.         if (i == 4 || (pre && i == 2)) continue;
10.        res += dfs(pos - 1, i == 6, lim && i == up);
11.    }
12.    if (!lim) f[pos][pre] = res;
13.    return res;
14. }
15.
16. int cal(int n) {
17.    if (n == 0) return 1;
18.    int pos = 0;
19.    while (n) {
20.        num[pos++] = n % 10;
21.        n /= 10;
22.    }
23.    return dfs(pos - 1, false, true);
```

```
}

int main() {
    memset(f, -1, sizeof f);
    int l, r;
    while (~scanf("%d %d", &l, &r) && l + r)
        printf("%d\n", cal(r) - cal(l - 1));
    return 0;
}
```

例题　启示录（AcWing310）

古代人认为 666 是属于魔鬼的数。

不但如此，只要某数字的十进制表示中有三个连续的 6，古代人也认为这是个魔鬼的数，比如 666,1666,6663,16666,6660666 等等。

古代典籍中经常用“第 X 小的魔鬼的数”来指代这些数，这给研究人员带来了极大的不便。

现在请编写一个程序，可以实现输入 X，输出对应的魔鬼数。

【输入格式】

第一行包含整数 T，表示共有 T 组测试数据。

每组测试数据占一行，包含一个整数 X。

$1 \leq T \leq 10^3, 1 \leq X \leq 5 \times 10^7$。

【输出格式】

每组测试数据占一行，输出一个魔鬼数。

【分析】

我们可以用数位 DP 求出区间 $[1,n]$ 中魔鬼数的个数，方法类似于上一题，设状态 $f[pos][0/1][i]$ 表示当前枚举到第 pos 位，高位未有/已有三个连续的 6，当前有 i 个连续的 6 的数的个数。

然后我们可以二分答案，求出最小的 n 满足区间 $[1,n]$ 的魔鬼数至少有 x 个。

```
typedef long long ll;

int num[40];
ll f[40][2][3];

ll dfs(int pos, bool have, int cnt, bool lim) {
    if (pos == -1) return have;
    if (!lim && ~f[pos][have][cnt]) return f[pos][have][cnt];
    int up = (lim ? num[pos] : 9); ll res = 0;
```

```
    for (int i = 0; i <= up; i++) {
        if (have) res += dfs(pos - 1, have, cnt, lim && i == up);
        else if (i != 6) res += dfs(pos - 1, have, 0, lim && i == up);
        else if (cnt == 2) res += dfs(pos - 1, true, 0, lim && i == up);
        else res += dfs(pos - 1, have, cnt + 1, lim && i == up);
    }
    if (!lim) f[pos][have][cnt] = res;
    return res;
}

ll cal(ll n) {
    int pos = 0;
    while (n) {
        num[pos++] = n % 10;
        n /= 10;
    }
    return dfs(pos - 1, false, 0, true);
}

void work() {
    int n; scanf("%d", &n);
    ll l = 1, r = ll(1e35), mid;
    while (l < r) {
        mid = (l + r) >> 1;
        if (cal(mid) >= n) r = mid;
        else l = mid + 1;
    }
    printf("%lld\n", r);
}

int main() {
    memset(f, -1, sizeof f);
    int T; scanf("%d", &T);
    for (int ca = 1; ca <= T; ca++) work();
    return 0;
}
```

例题　[AHOI2009]同类分布（洛谷 P4127）

给出两个数 a,b，求出 $[a,b]$ 中各位数字之和能整除原数的数的个数。

【输入格式】

一行，两个整数 a 和 b。

$1 \le a \le b \le 10^{18}$。

【输出格式】

一个整数，表示答案。

【分析】

因为 $1 \le a \le b \le 10^{18}$，所以数位之和最大为 9×18。

我们可以枚举数位之和 tar，然后用数位 DP 统计数位之和为 tar 且数可以被 tar 整除的数的个数。

```
typedef long long ll;

int num[20];
ll pw[20];
ll f[20][170][170];

ll dfs(int pos, int tar, int sum, int pro, bool lim) {
    if (pos == -1) return sum == tar && pro % sum == 0;
    if (!lim && ~f[pos][sum][pro]) return f[pos][sum][pro];
    int up = (lim ? num[pos] : 9); ll res = 0;
    for (int i = 0; i <= up; i++) {
        if (sum + i <= tar && sum + i + pos * 9 >= tar)
            res += dfs(pos - 1, tar, sum + i, (pro + i * pw[pos]) % tar, lim && i == up);
    }
    if (!lim) f[pos][sum][pro] = res;
    return res;
}

ll cal(ll n) {
    int pos = 0;
    while (n) {
        num[pos++] = n % 10;
        n /= 10;
    }
    ll res = 0;
    for (int i = 1; i <= 9 * pos; i++) {
        memset(f, -1, sizeof f);
        res += dfs(pos - 1, i, 0, 0, true);
    }
    return res;
}

void work() {
    ll a, b; scanf("%lld %lld", &a, &b);
    printf("%lld\n", cal(b) - cal(a - 1));
}
```

```
37.
38. int main() {
39.     pw[0] = 1;
40.     for (int i = 1; i < 20; i++) pw[i] = pw[i - 1] * 10;
41.     memset(f, -1, sizeof f);
42.     work();
43.     return 0;
44. }
```

9.7 概率 DP

概率 DP 用于解决概率问题与期望问题，一般情况下，解决概率问题需要顺序循环，而解决期望问题使用逆序循环。

例题　Race to 1 Again（LightOJ1038）

有一个正整数 n，每次操作会随机选取 n 的一个因子 x，并将 n 变为 n/x，求把 n 变为 1 的期望操作次数。

【输入格式】

T 组测试数据，每组测试数据输入一个整数 n。

$1 \le T \le 10^4, 1 \le n \le 10^5$。

【输出格式】

对于每组测试数据输出一个整数，表示把 n 变为 1 的期望操作次数。

【分析】

设状态 $f[i]$ 表示把 n 变为 1 的期望操作次数，n 的因子按照从小到大依次为 $x_1, x_2, \cdots, x_k$，其中 $x_k = n$，则有

$$f[n] = 1 + \frac{1}{k}\sum_{i=1}^{k} f[x_i]$$

分离出 x_k

$$f[n] = 1 + \frac{1}{k}f[n] + \frac{1}{k}\sum_{i=1}^{k-1} f[x_i]$$

整理一下得到

$$f[n] = \frac{k + \sum_{i=1}^{k-1} f[x_i]}{k-1}$$

我们可以预处理出 f 数组，每次回答 $O(1)$ 查询。

时间复杂度为 $O(n+T)$。

```
1. typedef double db;
```

```
 2.
 3. const int M = int(1e5);
 4.
 5. vector<int> v[M + 5];
 6. db f[M + 5];
 7.
 8. void init() {
 9.     for (int i = 1; i <= M; i++)
10.         for (int j = i; j <= M; j += i)
11.             v[j].push_back(i);
12.     for (int i = 2; i <= M; i++) {
13.         for (const int &j: v[i]) if (i != j) f[i] += f[j];
14.         f[i] = (f[i] + v[i].size()) / (v[i].size() - 1.0);
15.     }
16. }
17.
18. void work() {
19.     int n; scanf("%d", &n);
20.     printf("%.12f\n", f[n]);
21. }
22.
23. int main() {
24.     init();
25.     int T; scanf("%d", &T);
26.     for (int ca = 1; ca <= T; ca++) {
27.         printf("Case %d: ", ca);
28.         work();
29.     }
30.     return 0;
31. }
```

例题　扑克牌（AcWing218）

Admin 生日那天，Rainbow 来找 Admin 玩扑克牌。

玩着玩着 Rainbow 觉得太没意思了，于是决定给 Admin 一个考验。

Rainbow 把一副扑克牌 (54 张) 随机洗开，倒扣着放成一摞。

然后 Admin 从上往下依次翻开每张牌，每翻开一张黑桃、红桃、梅花或者方块，就把它放到对应花色的堆里去。

Rainbow 想问问 Admin，得到 A 张黑桃、B 张红桃、C 张梅花、D 张方块需要翻开的牌的张数的期望值 E 是多少？

特殊地，如果翻开的牌是大王或者小王，Admin 将会把它作为某种花色的牌放入对应堆中，使得放入之后 E 的值尽可能小。 由于 Admin 和 Rainbow 还在玩扑克，所以这个

程序就交给你来写了。

【输入格式】

输入仅由一行，包含四个用空格隔开的整数，A,B,C,D。

$0 \le A,B,C,D \le 15$。

【输出格式】

输出需要翻开的牌数的期望值 E，四舍五入保留 3 位小数。 如果不可能达到输入的状态，输出 -1.000。

【分析】

设状态 $f[a,b,c,d,x,y]$ 表示当前已经翻开了 a 张黑桃，b 张红桃，c 张梅花，d 张方块，并且小王状态为 x，大王状态为 y 时的期望值。具体来说，$x=4$ 表示没有用过小王，$x=0\sim3$ 表示用过小王，且把小王作为相应的花色（0 代表黑桃，1 代表红桃，2 代表梅花，3 代表方块）。大王的记录方法同理。

当前已经翻开的牌总数 $sum=a+b+c+d+(x\neq4)+(y\neq4)$。到目前为止还剩下 $54-sum$ 张牌，其中有 $13-a$ 张黑桃。故翻开一张黑桃的概率为 $\frac{13-a}{54-sum}$。翻开这张黑桃后，还需要翻开的牌的期望张数为 $f[a+1,b,c,d,x,y]$。对于红桃、梅花、方块的情况类似。

特别的，当 $x=4$ 时，有 $\frac{1}{54-sum}$ 的概率翻开小王。根据题意，应选择把小王看作某种花色，使期望值尽量小，即 $\min\limits_{0\le x'\le3} f[a,b,c,d,x',y]$。对于大王，情况类似。

初值：若已经翻开的牌数达到了题目要求的数量，则期望值为 0，否则为 -1。答案即为 $f[0][0][0][0][4][4]$。

值得一提的是，在数学期望递推、数学期望动态规划中，我们通常把终止状态（翻开的牌数达到要求）作为初值，把起始状态（尚未翻开任何牌)作为目标，倒着进行计算。这是因为在很多情况下，起始状态是唯一的 $(0,0,0,0,4,4)$，而终止状态很多（只要各花色翻开的牌数都足够即可）。根据数学期望的定义，若我们正着计算，则还需求出从起始状态到达每个终止状态的概率，与 f 值相乘求和才能得到答案，增加了难度,也容易出错。而如果倒着计算，因为起始状态 $(0,0,0,0,4,4)$ 唯一，所以它的概率一定是 1，直接输出 $f[0][0][0][0][4][4]$ 即为所求。

```
1. template <typename T>
2. inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}
3.
4. typedef double db;
```

```
5.
6. const db eps = 1e-6;
7. const int inf = 0x3f3f3f3f;
8.
9. inline int sign(db a) {return a < -eps ? -1 : a > eps;}
10. inline int cmp(db a, db b) {return sign(a - b);}
11.
12. int A, B, C, D;
13. db f[20][20][20][20][5][5];
14.
15. db dfs(int a, int b, int c, int d, int x, int y) {
16.     if (cmp(f[a][b][c][d][x][y], -1.) != 0) return f[a][b][c][d][x][y];
17.     f[a][b][c][d][x][y] = 0.;
18.     int s = a + b + c + d + (x != 4) + (y != 4);
19.     if (a < 13) f[a][b][c][d][x][y] += (13. - a) / (54. - s) * dfs(a + 1, b, c, d, x,
y);
20.     if (b < 13) f[a][b][c][d][x][y] += (13. - b) / (54. - s) * dfs(a, b + 1, c, d, x,
y);
21.     if (c < 13) f[a][b][c][d][x][y] += (13. - c) / (54. - s) * dfs(a, b, c + 1, d, x,
y);
22.     if (d < 13) f[a][b][c][d][x][y] += (13. - d) / (54. - s) * dfs(a, b, c, d + 1, x,
y);
23.     if (x == 4) {
24.         db mi = inf;
25.         cmin(mi, dfs(a, b, c, d, 0, y));
26.         cmin(mi, dfs(a, b, c, d, 1, y));
27.         cmin(mi, dfs(a, b, c, d, 2, y));
28.         cmin(mi, dfs(a, b, c, d, 3, y));
29.         f[a][b][c][d][x][y] += 1. / (54. - s) * mi;
30.     }
31.     if (y == 4) {
32.         db mi = inf;
33.         cmin(mi, dfs(a, b, c, d, x, 0));
34.         cmin(mi, dfs(a, b, c, d, x, 1));
35.         cmin(mi, dfs(a, b, c, d, x, 2));
36.         cmin(mi, dfs(a, b, c, d, x, 3));
37.         f[a][b][c][d][x][y] += 1. / (54. - s) * mi;
38.     }
39.     return f[a][b][c][d][x][y] += 1.;
40. }
41.
42. void work() {
43.     scanf("%d %d %d %d", &A, &B, &C, &D);
44.     vector<int> v{A, B, C, D};
```

```
45.     if (accumulate(v.begin(), v.end(), 0,
46.         [](int x, int y) {return x + max(0, y - 13);}) > 2)
47.         return puts("-1.000"), void();
48.     for (int i = 0; i <= 13; i++)
49.     for (int j = 0; j <= 13; j++)
50.     for (int k = 0; k <= 13; k++)
51.     for (int l = 0; l <= 13; l++)
52.     for (int x = 0; x < 5; x++)
53.     for (int y = 0; y < 5; y++) {
54.         int a = i + (x == 0) + (y == 0);
55.         int b = j + (x == 1) + (y == 1);
56.         int c = k + (x == 2) + (y == 2);
57.         int d = l + (x == 3) + (y == 3);
58.         if (a >= A && b >= B && c >= C && d >= D)
59.             f[i][j][k][l][x][y] = .0;
60.         else f[i][j][k][l][x][y] = -1.;
61.     }
62.     db ans = dfs(0, 0, 0, 0, 4, 4);
63.     printf("%.3f\n", ans);
64. }
```

例题　绿豆蛙的归宿（AcWing217）

给出一个有向无环的连通图，起点为 1，终点为 n，每条边都有一个长度。

数据保证从起点出发能够到达图中所有的点，图中所有的点也都能够到达终点。

绿豆蛙从起点出发，走向终点。 到达每一个顶点时，如果有 k 条离开该点的道路，绿豆蛙可以选择任意一条道路离开该点，并且走向每条路的概率为 $1/k$。

现在绿豆蛙想知道，从起点走到终点所经过的路径总长度的期望是多少？

【输入格式】

第一行: 两个整数 n,m，代表图中有 n 个点、m 条边。

第二行到第 $1+M$ 行: 每行 3 个整数 a,b,c，代表从 a 到 b 有一条长度为 c 的有向边。

$1 \le n \le 10^5, 1 \le m \le 2n$。

【输出格式】

输出从起点到终点路径总长度的期望值，结果四舍五入保留两位小数。

【分析】

设状态 $f[i]$ 表示从 i 号点出发，到达 n 号点的期望长度，假设当前位于 u 号点，且 u 号点有 k 条出边，对应的点分别是 $v_1, v_2, \cdots, v_k$，则有状态转移方程

$$f[u] = 1 + \frac{1}{k}\sum_{i=1}^{k} f[v_i]$$

时间复杂度为 $O(n+m)$。

```
typedef double db;

const int M = int(1e5);
const db eps = 1e-6;

inline int sign(db a) {return a < -eps ? -1 : a > eps;}
inline int cmp(db a, db b) {return sign(a - b);}

db f[M + 5];
vector<pair<int, int>> g[M + 5];

db dfs(int u) {
    if (cmp(f[u], -1.) != 0) return f[u];
    f[u] = 0.;
    for (const auto &[v, w]: g[u]) f[u] += dfs(v) + w;
    return f[u] = f[u] / g[u].size();
}

void work() {
    int n, m; scanf("%d %d", &n, &m);
    for (int i = 1, a, b, c; i <= m; i++) {
        scanf("%d %d %d", &a ,&b, &c);
        g[a].push_back({b, c});
    }
    for (int i = 1; i < n; i++) f[i] = -1.;
    f[n] = 0.;
    printf("%.2f\n", dfs(1));
}
```