么办呢？这种情况下的处理方法基本相同，对于从下方开始的第i个球，在按照R=0计算的结果上加上$2Ri$就好了。

```
const double g = 10.0;  // 重力加速度

// 输入
int N, H, R, T;

double y[MAX_N];        // 球的最终位置

// 求出T时刻球的位置
double calc(int T) {
  if (T < 0) return H;
  double t = sqrt(2 * H / g);
  int k = (int)(T / t);
  if (k % 2 == 0) {
    double d = T - k * t;
    return H - g * d * d / 2;
  } else {
    double d = k * t + t - T;
    return H - g * d * d / 2;
  }
}

void solve() {
  for (int i = 0; i < N; i++) {
    y[i] = calc(T - i);
  }
  sort(y, y + N);
  for (int i = 0; i < N; i++) {
    printf("%.2f%c", y[i] + 2 * R * i / 100.0, i + 1 == N ? '\n' : ' ');
  }
}
```

3.2.4　折半枚举（双向搜索）[①]

4 Values whose Sum is 0 （POJ No.2785）

给定各有 n 个整数的四个数列 A、B、C、D。要从每个数列中各取出 1 个数，使四个数的和为 0。求出这样的组合的个数。当一个数列中有多个相同的数字时，把它们作为不同的数字看待。

限制条件

- $1 \leqslant n \leqslant 4000$
- $|(\text{数字的值})| \leqslant 2^{28}$

① 本节所介绍的折半枚举与传统的双向搜索并不相同，但其思想来源于传统的双向搜索，有时候我们也会用双向搜索来指代它，特此注明。——译者注

样例

输入

```
n = 6
A = {-45, -41, -36, -36, 26, -32}
B = {22, -27, 53, 30, -38, -54}
C = {42, 56, -37, -75, -10, -6}
D = {-16, 30, 77, -46, 62, 45}
```

输出

```
5(-45-27+42+30=0, 26+30-10-46=0, -32+22+56-46=0, -32+30-75+77=0, -32-54+56+30=0)
```

这个问题是热身题目“抽签”的复习。从4个数列中选择的话总共有n^4种情况，所以全都判断一遍不可行。不过将它们对半分成AB和CD再考虑，就可以快速解决了。从2个数列中选择的话只有n^2种组合，所以可以进行枚举。先从A、B中取出a、b后，为了使总和为0则需要从C、D中取出$c+d=-a-b$。因此先将从C、D中取数字的n^2种方法全都枚举出来，将这些和排好序，这样就可以运用二分搜索了。这个算法的复杂度是$O(n^2\log n)$。

有时候，问题的规模较大，无法枚举所有元素的组合，但能够枚举一半元素的组合。此时，将问题拆成两半后分别枚举，再合并它们的结果这一方法往往非常有效。

```
// 输入
int n;
int A[MAX_N], B[MAX_N], C[MAX_N], D[MAX_N];

int CD[MAX_N * MAX_N];  // C和D中数字的组合方法

void solve() {
  // 枚举从C和D中取出数字的所有方法
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      CD[i * n + j] = C[i] + D[j];
    }
  }
  sort(CD, CD + n * n);

  long long res = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      int cd = -(A[i] + B[j]);
      // 取出C和D中和为cd的部分
      res += upper_bound(CD, CD + n * n, cd) - lower_bound(CD, CD + n * n, cd);
    }
  }

  printf("%lld\n", res);
}
```

超大背包问题

有重量和价值分别为 w_i,v_i 的 n 个物品。从这些物品中挑选总重量不超过 W 的物品，求所有挑选方案中价值总和的最大值。

限制条件

- $1 \leqslant n \leqslant 40$
- $1 \leqslant w_i,v_i \leqslant 10^{15}$
- $1 \leqslant W \leqslant 10^{15}$

样例

输入

```
n = 4
w = {2, 1, 3, 2}
v = {3, 2, 4, 2}
W = 5
```

输出

```
7(挑选0、1、3号物品)
```

这个问题是第二章介绍过的背包问题。不过这次价值和重量都可以是非常大的数值，相比之下n比较小。使用DP求解背包问题的复杂度是$O(nW)$，因此不能用来解决这里的问题。此时我们应该利用n比较小的特点来寻找其他办法。

挑选物品的方法总共有2^n种，所以不能直接枚举，但是像前面一样拆成两半之后再枚举的话，因为每部分只有20个所以是可行的。利用拆成两半后的两部分的价值和重量，我们能求出原先的问题吗？我们把前半部分中的选取方法对应的重量和价值总和记为$w1$、$v1$。这样在后半部分寻找总重$w2 \leqslant W-w1$时使$v2$最大的选取方法就好了。

因此，我们要思考从枚举得到的$(w2,v2)$的集合中高效寻找$\max\{v2|w2 \leqslant W'\}$的方法。首先，显然我们可以排除所有$w2[i] \leqslant w2[j]$并且$v2[i] \geqslant v2[j]$的$j$。这一点可以按照$w2$、$v2$的字典序排序后简单做到。此后剩余的元素都满足$w2[i]<w2[j] \Leftrightarrow v2[i]<v2[j]$，要计算$\max\{v2|w2 \leqslant W'\}$的话，只要寻找满足$w2[i] \leqslant W'$的最大的$i$就可以了。这可以用二分搜索完成，剩余的元素个数为$M$的话，一次搜索需要$O(\log M)$的时间。因为$M \leqslant 2^{(n/2)}$，所以这个算法总的复杂度是$O(2^{(n/2)}n)$，可以在时限内解决这个问题。

```
typedef long long ll;

// 输入
int n;
ll w[MAX_N], v[MAX_N];
ll W;

pair<ll, ll> ps[1 << (MAX_N / 2)];  // (重量, 价值)

void solve() {
  // 枚举前半部分
  int n2 = n / 2;
  for (int i = 0; i < 1 << n2; i++) {
    ll sw = 0, sv = 0;
    for (int j = 0; j < n2; j++) {
      if (i >> j & 1) {
        sw += w[j];
        sv += v[j];
      }
    }
    ps[i] = make_pair(sw, sv);
  }

  // 去除多余的元素
  sort(ps, ps + (1 << n2));
  int m = 1;
  for (int i = 1; i < 1 << n2; i++) {
    if (ps[m - 1].second < ps[i].second) {
      ps[m++] = ps[i];
    }
  }

  // 枚举后半部分并求解
  ll res = 0;
  for (int i = 0; i < 1 << (n - n2); i++) {
    ll sw = 0, sv = 0;
    for (int j = 0; j < n - n2; j++) {
      if (i >> j & 1) {
        sw += w[n2 + j];
        sv += v[n2 + j];
      }
    }
    if (sw <= W) {
      ll tv = (lower_bound(ps, ps + m, make_pair(W - sw, INF)) - 1)->second;
      res = max(res, sv + tv);
    }
  }

  printf("%lld\n", res);
}
```

3.2.5　坐标离散化

区域的个数

$w \times h$ 的格子上画了 n 条或垂直或水平的宽度为 1 的直线。求出这些线将格子划分成了多少个区域。

例

限制条件

- $1 \leqslant w, h \leqslant 1000000$
- $1 \leqslant n \leqslant 500$

样例

```
w = 10, h = 10, n = 5
x1 = {1, 1, 4, 9, 10}
x2 = {6, 10, 4, 9, 10}
y1 = {4, 8, 1, 1, 6}
y2 = {4, 8, 10, 5, 10}
(对应于前面的例图)
```

准备好$w \times h$的数组，并记录是否有直线通过，然后参考2.1节利用深度优先搜索求水坑数的方法，可以求出被分割出的区域个数。但是这个问题中w和h最大为1000000，所以没办法创建$w \times h$的数组。因此我们要使用坐标离散化这一技巧。

坐标离散化示例

如上图所示，将前后没有变化的行列消除后并不会影响区域的个数。

数组里只需要存储有直线的行列以及其前后的行列就足够了，这样的话大小最多$6n \times 6n$就足够了。因此就可以创建出数组并利用搜索[1]求出区域的个数了。

```
// 输入
int W, H, N;
int X1[MAX_N], X2[MAX_N], Y1[MAX_N], Y2[MAX_N];

// 填充用
bool fld[MAX_N * 6][MAX_N * 6];

// 对x1和x2进行坐标离散化，并返回离散化之后的宽度
int compress(int *x1, int *x2, int w) {
  vector<int> xs;

  for (int i = 0; i < N; i++) {
    for (int d = -1; d <= 1; d++) {
      int tx1 = x1[i] + d, tx2 = x2[i] + d;
      if (1 <= tx1 && tx1 <= W) xs.push_back(tx1);
      if (1 <= tx2 && tx2 <= W) xs.push_back(tx2);
    }
  }

  sort(xs.begin(), xs.end());
  xs.erase(unique(xs.begin(), xs.end()), xs.end());

  for (int i = 0; i < N; i++) {
    x1[i] = find(xs.begin(), xs.end(), x1[i]) - xs.begin();
    x2[i] = find(xs.begin(), xs.end(), x2[i]) - xs.begin();
  }

  return xs.size();
}

void solve() {
  // 坐标离散化
  W = compress(X1, X2, W);
  H = compress(Y1, Y2, H);

  // 填充有直线的部分
  memset(fld, 0, sizeof(fld));
  for (int i = 0; i < N; i++) {
    for (int y = Y1[i]; y <= Y2[i]; y++) {
      for (int x = X1[i]; x <= X2[i]; x++) {
        fld[y][x] = true;
      }
    }
  }
```

① 区域可能很大，所以用递归函数实现的话可能会栈溢出。

```
  // 求区域的个数
  int ans = 0;
  for (int y = 0; y < H; y++) {
    for (int x = 0; x < W; x++) {
      if (fld[y][x]) continue;
      ans++;

      // 宽度优先搜索
      queue<pair<int, int> > que;
      que.push(make_pair(x, y));
      while (!que.empty()) {
        int sx = que.front().first, sy = que.front().second;
        que.pop();

        for (int i = 0; i < 4; i++) {
          int tx = sx + dx[i], ty = sy + dy[i];
          if (tx < 0 || W <= tx || ty < 0 || H <= ty) continue;
          if (fld[ty][tx]) continue;
          que.push(make_pair(tx, ty));
          fld[ty][tx] = true;
        }
      }
    }
  }

  printf("%d\n", ans);
}
```

3.3 活用各种数据结构

除了在2.4节中介绍的数据结构之外，还有许多非常有用的数据结构。本节，我们将介绍线段树（Segment Tree）、树状数组（Binary Indexed Tree）等数据结构。

3.3.1 线段树

1. 线段树的概念

线段树是擅长处理区间的，形如下图的数据结构。线段树是一棵完美二叉树（Perfect Binary Tree）[①]（所有的叶子的深度都相同，并且每个节点要么是叶子要么有2个儿子的树），树上的每个节点都维护一个区间。根维护的是整个区间，每个节点维护的是父亲的区间二等分后的其中一个子区间。当有n个元素时，对区间的操作可以在$O(\log n)$的时间内完成。

线段树的样子

根据节点中维护的数据的不同，线段树可以提供不同的功能。下面我们以实现了Range Minimum Query(RMQ)操作的线段树为例，进行说明。

2. 基于线段树的RMQ的结构

下面要建立的线段树在给定数列$a_0, a_1, \cdots, a_{n-1}$的情况下，可以在$O(\log n)$时间内完成如下两种操作

- 给定s和t，求$a_s, a_{s+1}, \cdots, a_t$的最小值
- 给定i和x，把a_i的值改成x

① 注意和完全二叉树（Complete Binary Tree）的区别，完全二叉树指的是在树中除了最后一层外，其余层都是满的，并且最后一层或者是满的，或者是在右边缺少连续的若干节点。不过也有的地方把完美二叉树叫做满二叉树或者完全二叉树。本书以前面的定义为准。——译者注

如下图，线段树的每个节点维护对应区间的最小值。在建树时，只需要按从下到上的顺序分别取左右儿子的值中的较小者就可以了。

数列及其对应的线段树的例子

3. 基于线段树的RMQ的查询

如果要求$a_0, \cdots, a_6$的最小值。我们只需要求下图中的三个节点的值的最小值即可。

3和4和1的最小值→1

需要查看值的3个节点

像这样，即使查询的是一个比较大的区间，由于较靠上的节点对应较大的区间，通过这些区间就可以知道大部分值的最小值，从而只需要访问很少的节点就可以求得最小值。

要求某个区间的最小值，像下面这样递归处理就可以了。

- 如果所查询的区间和当前节点对应的区间完全没有交集，那么就返回一个不影响答案的值（例如INT_MAX）。
- 如果所查询的区间完全包含了当前节点对应的区间，那么就返回当前节点的值。
- 以上两种情况都不满足的话，就对两个儿子递归处理，返回两个结果中的较小者。

4. 基于线段树的RMQ的值的更新

在更新a_0的值时，需要重新计算下图所示的4个节点的值。

需要重新计算值的4个节点

在更新a_i的值时，需要对包含i的所有区间对应的节点的值重新进行计算。在更新时，可以从下面的节点开始向上不断更新，把每个节点的值更新为左右两个儿子的值的较小者就可以了。

5. 基于线段树的RMQ的复杂度

不论哪种操作，对于每个深度都最多访问常数个节点。因此对于n个元素，每一次操作的复杂度是$O(\log n)$。对于二叉搜索树，我们曾经提到过可能有因操作不当而导致退化的情况发生，从而使复杂度变得很糟糕。不过因为线段树不会添加或者删除节点，所以即使是朴素的实现也都能在$O(\log n)$时间内进行各种操作。

此外，n个元素的线段树的初始化的时间复杂度和总的空间复杂度都是$O(n)$。这是因为节点数是$n+n/2+n/4+\cdots=2n$。直觉上很容易让人产生复杂度是$O(n \log n)$的错觉，需要注意。

6. 基于线段树的RMQ的实现

为了简单起见，在建立线段树时，把数列所有的值都初始化为INT_MAX。此外，query的参数中不止传入节点的编号，还传入了节点对应的区间。

虽然从节点的编号也可以计算出对应的区间，但是把区间作为参数传入就可以节省这一步计算，为了简单起见，我们在实现中传入了对应的区间。

```
const int MAX_N = 1 << 17;

// 存储线段树的全局数组
int n, dat[2 * MAX_N - 1];

// 初始化
void init(int n_) {
  // 为了简单起见，把元素个数扩大到2的幂
  n = 1;
  while (n < n_) n *= 2;

  // 把所有的值都设为INT_MAX
```

```
  for (int i = 0; i < 2 * n - 1; i++) dat[i] = INT_MAX;
}

// 把第k个值（0-indexed）更新为a
void update(int k, int a) {
  // 叶子节点
  k += n - 1;
  dat[k] = a;
  // 向上更新
  while (k > 0) {
    k = (k - 1) / 2;
    dat[k] = min(dat[k * 2 + 1], dat[k * 2 + 2]);
  }
}

// 求[a, b)的最小值
// 后面的参数是为了计算起来方便而传入的。
// k是节点的编号，l，r表示这个节点对应的是[l, r)区间。
// 在外部调用时，用query(a, b, 0, 0, n)
int query(int a, int b, int k, int l, int r) {

// 如果[a, b)和[l,r)不相交，则返回TNT_MAX
  if (r <= a || b <= l) return INT_MAX;

  // 如果[a, b)完全包含[l, r)，则返回当前节点的值
  if (a <= l && r <= b) return dat[k];
  else {
    // 否则返回两个儿子中值的较小者
    int vl = query(a, b, k * 2 + 1, l, (l + r) / 2);
    int vr = query(a, b, k * 2 + 2, (l + r) / 2, r);
    return min(vl, vr);
  }
}
```

7. 需要运用线段树的问题

Crane（POJ 2991）

有一台起重机。我们把起重机看成由 N 条线段依次首尾相接而成。第 i 条线段的长度是 L_i。最开始，所有的线段都笔直连接，指向上方。

有 C 条操纵起重机的指令。指令 i 给出两个整数 S_i 和 A_i，效果是使线段 S_i 和 S_i+1 之间的角度变成 A_i 度。其中角度指的是从线段 S_i 开始沿逆时针方向旋转到 S_{i+1} 所经过的角度。最开始时所有角度都是 180 度。

按顺序执行这 C 条指令。在每条指令执行之后，输出起重机的前端（第 N 条线段的端点）的坐标。假设起重机的支点的坐标是(0, 0)。

限制条件

- $1 \leq N, C \leq 10000$

- $1 \leqslant L_i \leqslant 100$
- $1 \leqslant S_i < N$, $0 \leqslant A_i \leqslant 359$

初始状态

样例 1

输入

```
N=2, C=1
L={10, 5}
S={1}
A={90}
```

输出

```
5.00 10.00
```

样例 2

输入

```
N = 3, C = 2
L = {5, 5, 5}
S = {1, 2}
A = {270, 90}
```

输出

```
-10.00 5.00
-5.00 10.00
```

本题可以使用线段树来解决。每个节点表示一段连续的线段的集合，并且维护下面两个值。

- 把对应的线段集合中的第一条线段转至垂直方向之后，从第一条线段的起点指向最后一条线段的终点的向量。
- （如果该节点有儿子节点）两个儿子节点对应的部分连接之后，右儿子需要转动的角度。

也就是说，如果节点i表示的向量是vx_i，vy_i，角度是ang_i，两个儿子节点是chl和chr，那么就有

$$vx_i = vx_{chl} + (\cos(ang_i) \times vx_{chr} - \sin(ang_i) \times vy_{chr})$$

$$vy_i = vy_{chl} + (\sin(ang_i) \times vx_{chr} + \cos(ang_i) \times vy_{chr})$$

这样，每次更新便可在$O(\log n)$时间内完成，而输出的值就是根节点对应的向量的值。

下面的实现和RMQ的有所不同，线段树的大小并没有扩大到2的幂。这个时候，线段树并不是一颗完美二叉树，但是在本题中同样可以完成各种操作。

```
const int ST_SIZE = (1 << 15) - 1;

// 输入
int N, C;
int L[MAX_N];
int S[MAX_C], A[MAX_N];

// 线段树所维护的数据
double vx[ST_SIZE], vy[ST_SIZE];  // 各节点的向量
double ang[ST_SIZE];              // 各节点的角度

// 为了查询角度的变化而保存的当前角度的数组
double prv[MAX_N];


// 初始化线段树
// k是节点的编号，l，r表示当前节点对应的是[l, r)区间
void init(int k, int l, int r) {
  ang[k] = vx[k] = 0.0;

  if (r - l == 1) {
    // 叶子节点
    vy[k] = L[l];
  }
  else {
```

```
    // 非叶子节点
    int chl = k * 2 + 1, chr = k * 2 + 2;
    init(chl, l, (l + r) / 2);
    init(chr, (l + r) / 2, r);
    vy[k] = vy[chl] + vy[chr];
  }
}


// 把s和s+1的角度变为a
// v是节点的编号，l，r表示当前节点对应的是[l, r)区间
void change(int s, double a, int v, int l, int r) {
  if (s <= l) return;
  else if (s < r) {
    int chl = v * 2 + 1, chr = v * 2 + 2;
    int m = (l + r) / 2;
    change(s, a, chl, l, m);
    change(s, a, chr, m, r);
    if (s <= m) ang[v] += a;

    double s = sin(ang[v]), c = cos(ang[v]);
    vx[v] = vx[chl] + (c * vx[chr] - s * vy[chr]);
    vy[v] = vy[chl] + (s * vx[chr] + c * vy[chr]);
  }
}


void solve() {
  // 初始化
  init(0, 0, N);
  for (int i = 1; i < N; i++) prv[i] = M_PI;

  // 处理操作
  for (int i = 0; i < C; i++) {
    int s = S[i];
    double a = A[i] / 360.0 * 2 * M_PI;  // 把角度换算为弧度

    change(s, a - prv[s], 0, 0, N);
    prv[s] = a;

    printf("%.2f %.2f\n", vx[0], vy[0]);
  }
}
```

专栏 基于稀疏表（Sparse Table）的RMQ

在 RMQ 的其他实现方法中，有一种叫做 Sparse Table 的方法较为常见。对于数列 $a_1, a_2, \cdots, a_n$ 构建的 Sparse Table 如下表所示。

i	1	2	3	4	5	6	7	8
a	5	3	7	9	6	4	1	2
t_0	5	3	7	9	6	4	1	2
t_1	3	3	7	6	4	1	1	
t_2	3	3	4	1	1			
t_3	1							

Sparse Table的例子

其中 $t_{i,j}$ 表示的是 $a_j, a_{j+1}, \cdots, a_{j+2^i}$ 的最小值($0 \leqslant i, 2^i \leqslant n, 1 \leqslant j \leqslant n-2^i+1$)

让我们思考如何在给定 x 和 y 的情况下，通过这个表快速求出 $a_x, a_{x+1}, \cdots, a_y$ 的最小值。

首先，求出满足 $2^i \leqslant y-x < 2^{i+1}$ 的 i。也就是使得 2^i 不超过 $y-x$ 的最大的 i。然后，$\min\{t_{i,x}, t_{i,y}-2^i+1\}$ 就是 $a_x, a_{x+1}, \cdots, a_y$ 的最小值了。

要求的区间的最小值=上面的两个区间的最小值

利用Sparse Table求解最小值的例子

用朴素的方法求解 i 需要花费 $O(\log n)$时间，如果用二分搜索法求解 i 则只需要 $O(\log \log n)$ 时间。而且，在实现时，如果使用的是 gcc 编译器，还有_builtin_clz[①]等编译器内置的函数可以使用。因此，它单次查询的效率比基于线段树的 RMQ 要高[②]。

但是，基于 Sparse Table 的 RMQ 在预处理时的时间复杂度和空间复杂度都达到了 $O(n \log n)$。而且，和基于线段树的 RMQ 相比无法高效地对值进行更新。

3.3.2　Binary Indexed Tree

树状数组（Binary Indexed Tree，BIT）是能够完成下述操作的数据结构。

给一个初始值全为0的数列$a_1, a_2, \cdots, a_n$

- 给定i，计算$a_1+a_2+\cdots+a_i$
- 给定i和x，执行$a_i+=x$

① gcc中类似的还有__builtin_parity, __builtin_popcount, __builtin_ffz, __builtin_ctz等非常实用的函数。——译者注

② 不利用__builtin_clz也可以利用预处理，总的来说基于Sparse Table的查询是$O(1)$的。——译者注

1. 基于线段树的实现

如果使用线段树，只需要对前一节中RMQ的样例做少许修改就可以实现这两个功能。线段树的每个节点上维护的是对应的区间的和。

线段树和计算与更新的例子

接下来，我们来看如何计算从s到t的和($a_s+a_{s+1}+\cdots+a_t$)。在基于线段树的实现中，这个和是可以直接求得的。

但是如果我们能够计算(从1到t的和)−(从1到$s-1$的和)，同样可以得到s到t的和。也就是说，只要对于任意i，我们都能计算出1到i的部分和就足够了。

在这样的限制下，会带来哪些改变呢？我们可以发现，线段树上每个节点的右儿子的值都不需要了（在计算时如果要使用这个点的值，那么它的左边的兄弟的值也一定会用到，这个时候只需要使用它们的父亲的值就可以了）。

灰色：不需要用到的节点

不需要用到的节点

基于上面的思路得到的数据结构就是BIT。比起线段树，BIT实现起来更方便，速度也更快。

2. BIT的结构

BIT使用数组维护下图所示的部分和

树状数组中每个元素对应的部分

也就是把线段树中不需要的节点去掉之后，再把剩下的节点对应到数组中。让我们对比每个节点对应的区间的长度和节点编号的二进制表示。以1结尾的1, 3, 5, 7的长度是1，最后有1个0的2, 6的长度是2，最后有2个0的4的长度是4……这样，编号的二进制表示就能够和区间非常容易地对应起来。利用这个性质，BIT可以通过非常简单的位运算实现。

3. BIT的求和

计算前i项的和需要从i开始，不断把当前位置i的值加到结果中，并从i中减去i的二进制最低非0位对应的幂，直到i变成0为止。i的二进制的最后一个1可以通过$i\&-i$得到。

求和的例子

4. BIT的值的更新

使第i项的值增加x需要从i开始，不断把当前位置i的值增加x，并把i的二进制最低非0位对应的幂加到i上。

值的更新的例子

5. BIT的复杂度

总共需要对$O(\log n)$个值进行操作，所以复杂度是$O(\log n)$。

6. BIT的实现

顺便提一下，$i-=i\&-i$也可以写作$i=i\&(i-1)$。

```
// [1, n]
int bit[MAX_N + 1], n;

int sum(int i) {
 int s = 0;
 while (i > 0) {
   s += bit[i];
   i -= i & -i;
 }
 return s;
}

void add(int i, int x) {
 while (i <= n) {
   bit[i] += x;
   i += i & -i;
 }
}
```

7. 二维BIT

BIT可以方便地扩展到二维的情况。对于$W \times H$的二维BIT只需要建立H个大小为x轴方向元素个数W的BIT，然后把这些BIT通过y轴方向的BIT管理起来就可以了。也就是说，y轴方向的BIT的每个元素不是整数，而是一个x轴方向的BIT。这样所有操作的复杂度都是$O(\log W \times \log H)$。用同样的方法可以扩展到更高维度的情况。

8. 需要运用BIT的问题

冒泡排序的交换次数

给定一个 1~n 的排列 $a_0, a_1, \cdots, a_{n-1}$，求对这个数列进行冒泡排序所需要的交换次数（冒泡排序是每次找到满足 $a_i > a_{i+1}$ 的 i，并交换 a_i 和 a_{i+1}，直到这样的 i 不存在为止的算法）。

限制条件

- $1 \leqslant n \leqslant 100000$

样例

输入

```
n=4, a={3,1,4,2}
```

输出

```
3
```

冒泡排序的复杂度是$O(n^2)$，所以无法通过模拟冒泡排序的过程来计算需要的交换次数。不过我们可以通过选取适当的数据结构来解决这个问题。

首先，所求的交换次数等价于满足$i<j$, $a_i>a_j$的(i,j)数对的个数（这种数对的个数叫做逆序数）。而对于每一个j，如果能够快速求出满足$i<j$, $a_i>a_j$的i的个数，那么问题就能迎刃而解。我们构建一个值的范围是1~n的BIT，按照j=0, 1, 2, ⋯, n−1的顺序进行如下操作。

- 把j−(BIT查询得到的前a_j项的和)加到答案中
- 把BIT中a_j位置上的值加1

对于每一个j，(BIT查询得到的前a_j项的和)就是满足$i<j$, $a_i \leqslant a_j$的i的个数。因此把这个值从j中减去之后，得到的就是满足$i<j$, $a_i>a_j$的i的个数。由于对于每一个j的复杂度是$O(\log n)$，所以整个算法的复杂度是$O(n \log n)$。[①]

```
typedef long long ll;

// 输入
int n, a[MAX_N];

// 这里省略了BIT部分的代码
```

① 实现计算逆序数更简单的算法是通过分治思想利用归并排序计算，请参考本书4.6节。——译者注

```
void solve() {
  ll ans = 0;
  for (int j = 0; j < n; j++) {
    ans += j - sum(a[j]);
    add(a[j], 1);
  }
  printf("%lld\n", ans);
}
```

A Simple Problem with Integers（POJ 3468）

给定一个数列 $A_1, A_2, \cdots, A_N$ 以及 Q 个操作，按顺序执行这些操作。操作分 2 种

- 给出 l, r, x，对 $A_l, A_{l+1}, \cdots, A_r$ 同时加上 x
- 给出 l, r，求 $A_l, A_{l+1}, \cdots, A_r$

树状数组可以高效地求出连续的一段元素之和或者更新单个元素的值。但是，无法高效地给某一个区间里的所有元素同时加上一个值。因此，本题无法直接使用树状数组。在这里，我们先考虑利用线段树来求解，然后再考虑改造树状数组来求解该问题。

如果对于每个节点，维护有该节点对应的区间的和，那么就可以在$O(\log n)$时间内求出任意区间的和。但是这样就没有办法高效地实现对一个区间同时加一个值，因为这需要对这个区间相关的所有节点都进行更新才可以。

更新操作变得低效的例子

为了保持线段树的高效，对于每个节点我们维护以下两个数据

a. 给这个节点对应的区间内的所有元素共同加上的值

b. 在这个节点对应的区间中除去a之外其他的值的和

通过单独维护共同加上的值，给区间同时加一个值的操作就可以高效地进行了。如果对于父亲节

点同时加了一个值，那么这个值就不会在儿子节点被重复考虑。在递归计算和时再把这一部分的值加到结果里面就可以了。这样，不论是同时加一个值还是查询一段的和复杂度都是$O(\log n)$。

- T是操作的种类。第i个操作的$T[i]$是C的话，就是给区间同时加一个值，是Q的话则是查询一段的和。
- A, L, R都是以0为下标起点的。

```
typedef long long ll;

const int DAT_SIZE = (1 << 18) - 1;

// 输入
int N, Q;
int A[MAX_N];
char T[MAX_Q];
int L[MAX_Q], R[MAX_Q], X[MAX_Q];

// 线段树
ll data[DAT_SIZE], datb[DAT_SIZE];


// 对区间[a, b)同时加x
// k是节点的编号，对应的区间是[l, r)
void add(int a, int b, int x, int k, int l, int r) {
  if (a <= l && r <= b) {
    data[k] += x;
  }
  else if (l < b && a < r) {
    datb[k] += (min(b, r) - max(a, l)) * x;
    add(a, b, x, k * 2 + 1, l, (l + r) / 2);
    add(a, b, x, k * 2 + 2, (l + r) / 2, r);
  }
}


// 计算[a, b)的和
// k是节点的编号，对应的区间是[l, r)
ll sum(int a, int b, int k, int l, int r) {
  if (b <= l || r <= a) {
    return 0;
  }
  else if (a <= l && r <= b) {
    return data[k] * (r - l) + datb[k];
  }
  else {
    ll res = (min(b, r) - max(a, l)) * data[k];
    res += sum(a, b, k * 2 + 1, l, (l + r) / 2);
    res += sum(a, b, k * 2 + 2, (l + r) / 2, r);
    return res;
  }
}
```

```
void solve() {
  for (int i = 0; i < N; i++) {
    add(i, i + 1, A[i], 0, 0, N);
  }

  for (int i = 0; i < Q; i++) {
    if (T[i] == 'C') {
      add(L[i], R[i] + 1, X[i], 0, 0, N);
    }
    else {
      printf("%lld\n", sum(L[i], R[i] + 1, 0, 0, N));
    }
  }
}
```

和线段树的做法类似，树状数组也可以通过在每个节点上维护两个数据，高效地进行上述操作。

如果给区间$[l, r)$同时加上x的话，每个节点的值将会如何变化呢？如果令

$$s(i) = \text{加上}x\text{之前的}\sum_{j=1}^{i} a_j$$

$$s'(i) = \text{加上}x\text{之后的}\sum_{j=1}^{i} a_j$$

那么就有

$$\begin{aligned} i < l &\rightarrow s'(i) = s(i) \\ l \leqslant i \leqslant r &\rightarrow s'(i) = s(i) + x \times (i - l + 1) \\ &\qquad\quad = s(i) + x \times i - x \times (l - 1) \\ r < i &\rightarrow s'(i) = s(i) + x \times (r - l + 1) \end{aligned}$$

前缀和的增加量

下面记sum(bit, i)为树状数组bit的前i项和。我们构建两个树状数组bit0和bit1，并且设

$$\sum_{j=1}^{i} a_j = \text{sum}(\text{bit1}, i) \times i + \text{sum}(\text{bit0}, i)$$

那么在$[l, r]$区间上同时加上x就可以看成是

- 在bit0的l位置加上$-x(l-1)$
- 在bit1的l位置加上x
- 在bit0的$r+1$位置加上xr
- 在bit1的$r+1$位置加上$-x$

这4个操作。因此，查询和更新操作都可以在$O(\log n)$时间里完成。

更一般地，如果操作得到的结果可以用i的n次多项式表示，那么就可以使用$n+1$个树状数组来进行维护了。

- T是操作的种类。第i个操作的$T[i]$是C的话，就是给区间同时加一个值，是Q的话则是查询一段的和。
- A, L, R都是以1为下标起点的。

```
// 输入
int N, Q;
int A[MAX_N + 1];
char T[MAX_Q];
int L[MAX_Q], R[MAX_Q], X[MAX_Q];

// BIT
ll bit0[MAX_N + 1], bit1[MAX_N + 1];


ll sum(ll *b, int i) {
  ll s = 0;
  while (i > 0) {
    s += b[i];
    i -= i & -i;
  }
  return s;
}


void add(ll *b, int i, int v) {
  while (i <= N) {
    b[i] += v;
    i += i & -i;
```

```
  }
}


void solve() {
  for (int i = 1; i <= N; i++) {
    add(bit0, i, A[i]);
  }

  for (int i = 0; i < Q; i++) {
    if (T[i] == 'C') {
      add(bit0, L[i], -X[i] * (L[i] - 1));
      add(bit1, L[i], X[i]);
      add(bit0, R[i] + 1, X[i] * R[i]);
      add(bit1, R[i] + 1, -X[i]);
    }
    else {
      ll res = 0;
      res += sum(bit0, R[i]) + sum(bit1, R[i]) * R[i];
      res -= sum(bit0, L[i] - 1) + sum(bit1, L[i] - 1) * (L[i] - 1);
      printf("%lld\n", res);
    }
  }
}
```

3.3.3 分桶法和平方分割

分桶法（bucket method）是把一排物品或者平面分成桶，每个桶分别维护自己内部的信息，以达到高效计算的目的的方法。

把一排物品分成桶维护的图例

把平面分成桶维护的图例

其中，平方分割（sqrt decomposition）是把排成一排的n个元素每$\sqrt{n}$个分在一个桶内进行维护的方法的统称。这样的分割方法可以使对区间的操作的复杂度降至$O(\sqrt{n})$。

和线段树一样，根据维护的数据的不同，平方分割可以支持很多不同的操作。接下来，和线段树一样，我们以RMQ为例对平方分割进行讲解。

1. 基于平方分割的RMQ

给定一个数列$a_1, a_2, ..., a_n$，目标是在$O(\sqrt{n})$复杂度内实现以下两个功能

- 给定s, t，求$a_s, a_{s+1}, ..., a_t$的最小值。
- 给定i, x，把a_i的值变为x。

2. 基于平方分割的RMQ的预处理

令$b = \text{floor}(\sqrt{n})$，把$a$中的元素每$b$分成一个桶，并且计算出每个桶内的最小值。

3. 基于平方分割的RMQ的查询

如下图所示，查询

- 如果桶完全包含在区间内，则查询桶的最小值。
- 如果元素所在的桶不完全被区间包含，则逐个检查最小值。

它们的最小值就是区间的最小值了。

查询的例子

4. 基于平方分割的RMQ的值的更新

在更新元素的值时，需要更新该元素所在的桶的最小值。这时只要遍历一遍桶内的元素就可以了。

5. 基于平方分割的RMQ的复杂度

在更新值时，因为每个桶内[1]有b个元素，所以复杂度是$O(b) = O(\sqrt{n})$

而在查询时

- 完全包含在区间内的桶的个数是$O(n / b)$
- 所在的桶不被区间完全包含的元素个数是$O(b)$

---

① 如果在每个桶内用二叉搜索树来维护元素，则可以更加高效地进行更新操作。

因为 $b=O(\sqrt{n})$，所以操作的复杂度是

$$O(\frac{n}{b}+b)=O(\sqrt{n}+\sqrt{n})=O(\sqrt{n})$$

6. 平方分割和线段树

因此，在平方分割中，对于任意区间，完全包含于其中的桶的数量和剩余元素的数量都是 $O(\sqrt{n})$，所以可以在 $O(\sqrt{n})$ 时间内完成各种操作。

在上面的RMQ的例题中，线段树进行各种操作的复杂度是 $O(\log n)$，比平方分割更快一些。一般地，如果线段树和平方分割都能实现某个功能，多数情况下线段树会比平方分割快。但是，因为平方分割在实现上比线段树简单，所以如果运行时间限制不是太紧时，也可以考虑使用平方分割。除此之外，也有一些功能是线段树无法高效维护但是平方分割却可以做到的。[①]

7. 需要运用平方分割的题目

K-th Number（POJ 2104）

给定一个数列 $a_1, a_2, \cdots, a_n$ 和 m 个三元组表示的查询。对于每个查询 (i, j, k)，输出 $a_i, a_{i+1}, \cdots, a_j$ 的升序排列中第 k 个数。

限制条件

- $n \leqslant 100000, m \leqslant 5000$
- $|a_i| \leqslant 10^9$

样例

输入

```
n = 7, m = 3
a = {1, 5, 2, 6, 3, 7, 4}
查询 = {(2, 5, 3), (4, 4, 1), (1, 7, 3)}
```

输出

```
5
6
3
```

① 这里所介绍的平方分割的实现所用的数据结构通常也叫做块状数组，有时候为了适应更复杂的操作还会用到块状链表。也有些问题用到了平方分割的思想但并没有用到任何数据结构。——译者注

因为查询的个数m很大，朴素的求法无法在规定时间内出解。因此应该选用合理的方式维护数据来做到高效地查询。

如果x是第k个数，那么一定有

- 在区间中不超过x的数不少于k个
- 在区间中小于x的数有不到k个

因此，如果可以快速求出区间里不超过x的数的个数，就可以通过对x进行二分搜索来求出第k个数是多少。

接下来，我们来看一下如何计算在某个区间里不超过x的数的个数。如果不进行预处理，那么就只能遍历一遍所有的元素。

另一方面，如果区间是有序的，那么就可以通过二分搜索法高效地求出不超过x的数的个数了。但是，如果对于每个查询都分别做一次排序，就完全无法降低复杂度。所以，可以考虑使用平方分割和线段树进行求解。

首先我们来看如何使用平方分割来解决这个问题。把数列每b个一组分到各个桶里，每个桶内保存有排序后的数列。这样，如果要求在某个区间中不超过x的数的个数，就可以这样求得。

- 对于完全包含在区间内的桶，用二分搜索法计算。
- 对于所在的桶不完全包含在区间内的元素，逐个检查。

如果把b设为$\sqrt{n}$，复杂度就变成

$$O((\frac{n}{b})\log b+b)=O(\sqrt{n}\log n)$$

其中，对每个元素的处理只要$O(1)$时间，而对于每个桶的处理则需要$O(\log b)$，所以比起让桶的数量和桶内元素的个数尽可能接近，我们更应该把桶的数量设置成比桶内元素个数略少一些，这样可以使得程序更加高效。如果把b设为$\sqrt{n\log n}$，复杂度就变成

$$O((\frac{n}{b})\log b+b)=O(\sqrt{n\log n})$$

接下来只需要对x进行二分搜索就可以了。因为答案一定是数列a里的某个元素，所以二分搜索需要执行$O(\log n)$次。因此，如果$b\fallingdotseq\sqrt{n\log n}$，包括预处理在内整个算法的复杂度就是$O(n\log n+m\sqrt{n}\log^{1.5}n)$。

```
const int B = 1000;  // 桶的大小

// 输入
int N, M;
int A[MAX_N];
int I[MAX_M], J[MAX_M], K[MAX_M];

int nums[MAX_N];                  // 对A排序之后的结果
vector<int> bucket[MAX_N / B];    // 每个桶排序之后的结果

void solve() {
  for (int i = 0; i < N; i++) {
    bucket[i / B].push_back(A[i]);
    nums[i] = A[i];
  }
  sort(nums, nums + N);
  // 虽然每B个一组剩下的部分所在的桶没有排序，但是不会产生问题
  for (int i = 0; i < N / B; i++) sort(bucket[i].begin(), bucket[i].end());

  for (int i = 0; i < M; i++) {
    // 求[l, r)区间中第k个数
    int l = I[i], r = J[i] + 1, k = K[i];

    int lb = -1, ub = N - 1;
    while (ub - lb > 1) {
      int md = (lb + ub) / 2;
      int x = nums[md];
      int tl = l, tr = r, c = 0;

      // 区间两端多出的部分
      while (tl < tr && tl % B != 0) if (A[tl++] <= x) c++;
      while (tl < tr && tr % B != 0) if (A[--tr] <= x) c++;

      // 对每一个桶进行计算
      while (tl < tr) {
        int b = tl / B;
        c += upper_bound(bucket[b].begin(),  bucket[b].end(), x)
             - bucket[b].begin();
        tl += B;
      }

      if (c >= k) ub = md;
      else lb = md;
    }

    printf("%d\n", nums[ub]);
  }
}
```

下面我们考虑一下如何使用线段树解决这个问题。我们把数列用线段树维护起来。线段树的每个节点都保存了对应区间排好序后的结果。在这之前我们接触过的线段树节点上保存的都是数值，而这次则有所不同，每个节点保存了一个数列。

在本题中使用的线段树的例子

建立线段树的过程和归并排序的类似，而每个节点的数列就是其两个儿子节点的数列合并后的结果。建树的复杂度是 $O(n\log n)$ 。顺带一提，这棵线段树正是归并排序的完整再现。[①]

要计算在某个区间中不超过x的数的个数，只需要递归地进行如下操作就可以了。

- 如果所给的区间和当前节点的区间完全没有交集，那么返回0个。
- 如果所给的区间完全包含了当前节点对应的区间，那么使用二分搜索法对该节点上保存的数组进行查找。
- 否则对两个儿子递归地进行计算之后求和即可。

由于对于同一深度的节点最多只访问常数个，因此可以在 $O(\log^2 n)$ 时间里求出不超过x的数的个数。所以整个算法的复杂度是 $O(n\log n + m\log^3 n)$ 。

```
const int ST_SIZE = (1 << 18) - 1;

// 输入
int N, M;
int A[MAX_N];
int I[MAX_M], J[MAX_M], K[MAX_M];

int nums[MAX_N];            // 对A排序之后的结果
vector<int> dat[ST_SIZE];   // 线段树的数据



// 构建线段树
// k是节点的编号，和区间[l, r)对应
void init(int k, int l, int r) {
  if (r - l == 1) {
    dat[k].push_back(A[l]);
  }
  else {
    int lch = k * 2 + 1, rch = k * 2 + 2;
    init(lch, l, (l + r) / 2);
    init(rch, (l + r) / 2, r);
    dat[k].resize(r - l);
```

① 所以这样的线段树也叫归并树。——译者注

```
    // 利用STL的merge函数把两个儿子的数列合并
    merge(dat[lch].begin(), dat[lch].end(), dat[rch].begin(), dat[rch].end(),
          dat[k].begin());
  }
}



// 计算[i, j)中不超过x的数的个数
// k是节点的编号，和区间[l, r)对应
int query(int i, int j, int x, int k, int l, int r) {
  if (j <= l || r <= i) {
    // 完全不相交
    return 0;
  }
  else if (i <= l && r <= j) {
    // 完全包含在里面
    return upper_bound(dat[k].begin(), dat[k].end(), x) - dat[k].begin();
  }
  else {
    // 对儿子递归地计算
    int lc = query(i, j, x, k * 2 + 1, l, (l + r) / 2);
    int rc = query(i, j, x, k * 2 + 2, (l + r) / 2, r);
    return lc + rc;
  }
}


void solve() {
  for (int i = 0; i < N; i++) nums[i] = A[i];
  sort(nums, nums + N);

  init(0, 0, N);

  for (int i = 0; i < M; i++) {
    // 查找[l, r)中第k个数
    int l = I[i], r = J[i] + 1, k = K[i];

    int lb = -1, ub = N - 1;
    while (ub - lb > 1) {
      int md = (ub + lb) / 2;
      int c = query(l, r, nums[md], 0, 0, N);
      if (c >= k) ub = md;
      else lb = md;
    }
    printf("%d\n", nums[ub]);
  }
}
```

专栏　区域树（Range Tree）

上面提到的每个节点维护一个数组的线段树和每个节点维护一棵树的线段树也叫做区域树（Range Tree）。

如果把数列 a_i 考虑成平面上 (i, a_i) 的点列，那么上面问题中的查询

- 计算区间 $[1, r)$ 中不超过 v 的数的个数

就可以看成是

- 计算满足 $1 \leq x < r, y \leq v$ 的点的个数

这样的查询。Range Tree 适合对矩形的区域进行处理。并且，和树状数组一样，通过多重嵌套的线段树也可以实现高维度的 Range Tree。

3.4 熟练掌握动态规划

➠在这一节，我们将讨论更加复杂的动态规划算法的问题。到目前为止，我们都是在对整数做动态规划。不过，我们也可以对整数之外的复杂类型进行动态规划。此外，对于一些特殊形式的递推式，我们还可以利用其特殊性质高效地计算。

3.4.1 状态压缩DP

旅行商问题

给定一个 n 个顶点组成的带权有向图的距离矩阵 $d(I,j)$（INF 表示没有边）。要求从顶点 0 出发，经过每个顶点恰好一次后再回到顶点 0。问所经过的边的总权重的最小值是多少？

限制条件

- $2 \leqslant n \leqslant 15$
- $0 \leqslant d(i,j) \leqslant 1000$

样例

输入

n=5（如下图所示）

输出

22（0 → 3 → 4 → 1 → 2 → 0）

这个问题就是著名的旅行商问题（TSP，Traveling Salesman Problem）。TSP问题是NP困难的，没有已知的多项式时间的高效算法可以解决这一问题。不过在程序设计竞赛中还是有可能出现这种范围较小的题目的。

所有可能的路线共有$(n-1)!$种。这是一个非常大的值，即使在本题中n已经很小了，仍然无法试遍每一种情况。对于这个问题，我们可以使用DP来解决。首先让我们试着写出它的递推式。

假设现在已经访问过的顶点的集合（起点0当作还未访问过的顶点）为S，当前所在的顶点为v，用$dp[S][v]$表示从v出发访问剩余的所有顶点，最终回到顶点0的路径的权重总和的最小值。由于从v出发可以移动到任意的一个节点$u \notin S$，因此有如下递推式

$$dp[V][0]=0$$
$$dp[S][v]=min\{dp[S \cup \{u\}][u]+d(v,u) | u \notin S\}$$

我们只要按照这个递推式进行计算就可以了。由于在这个递推式中，有一个下标是集合而不是普通的整数，因此需要稍加处理。首先我们试着使用记忆化搜索求解。虽然有一个下标不是整数，但是我们可以把它编码成一个整数，或者给它们定义一个全序关系并用二叉搜索树存储，从而可以使用记忆化搜索来求解。特别地，对于集合我们可以把每一个元素的选取与否对应到一个二进制位里，从而把状态压缩成一个整数，大大方便了计算和维护。

```
// 输入
int n;
int d[MAX_N][MAX_N];

int dp[1 << MAX_N][MAX_N];  // 记忆化搜索使用的数组

// 已经访问过的节点集合为S，当前位置为v
int rec(int S, int v) {
  if (dp[S][v] >= 0) {
    return dp[S][v];
  }

  if (S == (1 << n) - 1 && v == 0) {
    // 已经访问过所有节点并回到0号点
    return dp[S][v] = 0;
  }

  int res = INF;
  for (int u = 0; u < n; u++) {
    if (!(S >> u & 1)) {
      // 下一步移动到顶点u
      res = min(res, rec(S | 1 << u, u) + d[v][u]);
    }
  }
  return dp[S][v] = res;
}
```

```
void solve() {
  memset(dp, -1, sizeof(dp));
  printf("%d\n", rec(0, 0));
}
```

这样，就可以在$O(2^n n^2)$的时间内完成计算。对于不是整数的情况，很多时候很难确定一个合适的递推顺序，因此使用记忆化搜索可以避免这个问题。不过在这个问题中，对于任意两个整数i和j，如果它们对应的集合满足$S(i) \subseteq S(j)$，就有$i \leqslant j$，因此还可以像下面的写法一样，通过循环求出答案。

```
int dp[1 << MAX_N][MAX_N];  // DP数组

void solve() {
  // 用足够大的值初始化数组
  for (int S = 0; S < 1 << n; S++) {
    fill(dp[S], dp[S] + n, INF);
  }
  dp[(1 << n) - 1][0] = 0;

  // 根据递推式依次计算
  for (int S = (1 << n) - 2; S >= 0; S--) {
    for (int v = 0; v < n; v++) {
      for (int u = 0; u < n; u++) {
        if (!(S >> u & 1)) {
          dp[S][v] = min(dp[S][v], dp[S | 1 << u][u] + d[v][u]);
        }
      }
    }
  }

  printf("%d\n", dp[0][0]);
}
```

像这样针对集合的DP我们一般叫状态压缩DP。

Travelling by Stagecoach （POJ No.2686）

有一个旅行家计划乘马车旅行。他所在的国家里共有 m 个城市，在城市之间有若干道路相连。从某个城市沿着某条道路到相邻的城市需要乘坐马车。而乘坐马车需要使用车票，每用一张车票只可以通过一条道路。每张车票上都记有马的匹数，从一个城市移动到另一个城市的所需时间等于城市之间道路的长度除以马的数量的结果。这位旅行家一共有 n 张车票，第 i 张车票上的马的匹数是 t_i。一张车票只能使用一次，并且换乘所需要的时间可以忽略。求从城市 a 到城市 b 所需要的最短时间。如果无法到达城市 b 则输出“Impossible”。

限制条件

- $1 \leqslant n \leqslant 8$

- $2 \leqslant m \leqslant 30$
- $1 \leqslant a,b \leqslant m\ (a \neq b)$
- $1 \leqslant t_i \leqslant 10$
- 1≤道路的长度≤100

样例

输入

```
n = 2
m = 4
a = 2
b = 1
t = {3, 1}
```

道路网

输出

```
3.667（5 / 3 + 2 / 1）
```

虽然可以把城市看作顶点，道路看作边建图，但是由于有车票相关的限制，无法直接使用Dijkstra算法求解。不过，这种情况下只需要把状态作为顶点，而把状态的转移看成边来建图就可以很好地避免这个问题。

让我们考虑一下“现在在城市v，此时还剩下的车票的集合为S”这样的状态。从这个状态出发，使用一张车票$i \in S$移动到相邻的城市u，就相当于转移到了“在城市u，此时还剩下的车票的集合为$S \backslash \{i\}$”这个状态。把这个转移看成一条边，那么边上的花费是(v–u间道路的长度)/t_i。按照上述方法所构的图，就可以用普通的Dijkstra算法求解了。

集合S使用状态压缩的方法表示就可以了。由于剩余的车票的集合S随着移动元素个数不断变小，因此这个图实际上是个DAG（请参照2.5节）。计算DAG的最短路不需要使用Dijkstra算法，可以简单地通过DP求解。在这道题中如下图所示。

样例所对应的状态图

```
// 输入
int n, m, a, b;
int t[MAX_N];
int d[MAX_M][MAX_M]; // 图的邻接矩阵表示（-1表示没有边）

// dp[S][v]:=到达剩下的车票集合为S并且现在在城市v的状态所需要的最小花费
double dp[1 << MAX_N][MAX_M];

void solve() {
  for (int i = 0; i < 1 << n; i++) {
    fill(dp[i], dp[i] + m, INF); // 用足够大的值初始化
  }
  dp[(1 << n) - 1][a - 1] = 0;
  double res = INF;
  for (int S = (1 << n) - 1; S >= 0; S--) {
    res = min(res, dp[S][b - 1]);
    for (int v = 0; v < m; v++) {
      for (int i = 0; i < n; i++) {
        if (S >> i & 1) {
          for (int u = 0; u < m; u++) {
            if (d[v][u] >= 0) {
              // 使用车票i，从v移动到u
              dp[S & ~(1 << i)][u] = min(dp[S & ~(1 << i)][u], dp[S][v] +
                                         (double)d[v][u] / t[i]);
            }
          }
        }
      }
    }
  }
  if (res == INF) {
    // 无法到达
    printf("Impossible\n");
  } else {
    printf("%.3f\n", res);
  }
}
```

铺砖问题

给定 $n \times m$ 的格子，每个格子被染成了黑色或者白色。现在要用 1×2 的砖块覆盖这些格子，要求块与块之间互相不重叠，且覆盖了所有白色的格子，但不覆盖任意一个黑色格子。求一共有多少种覆盖方法，输出方案数对 M 取余后的结果。

限制条件

- $1 \leqslant n \leqslant 15$
- $1 \leqslant m \leqslant 15$
- $2 \leqslant M \leqslant 10^9$

样例

输入

```
n =  3
m =  3
每个格子的颜色如下所示（.表示白色，x表示黑色）

...
.x.
...
```

输出

```
2
```

2种解

首先考虑一下枚举所有的解这一方法。为了不重复统计，我们每次从最左上方的空格处开始放置。对于哪些格子已经被覆盖过了，我们只需要使用一个bool数组来维护即可，按照下面的方法写就可以了。

```
// 输入
int n, m;
bool color[MAX_N][MAX_M];  // false: 白, true: 黑

// 现在查看的格子是(i, j)，used表示哪些格子已经被覆盖过
```

```
int rec(int i, int j, bool used[MAX_N][MAX_M]) {
  if (j == m) {
    // 到下一行
    return rec(i + 1, 0, used);
  }

  if (i == n) {
    // 已经覆盖了所有的空格
    return 1;
  }

  if (used[i][j] || color[i][j]) {
    // 不需要在(i, j)上放置砖块
    return rec(i, j + 1, used);
  }

  // 尝试2种放法
  int res = 0;
  used[i][j] = true;

  // 横着放
  if (j + 1 < m && !used[i][j + 1] && !color[i][j + 1]) {
    used[i][j + 1] = true;
    res += rec(i, j + 1, used);
    used[i][j + 1] = false;
  }

  // 竖着放
  if (i + 1 < n && !used[i + 1][j] && !color[i + 1][j]) {
    used[i + 1][j] = true;
    res += rec(i, j + 1, used);
    used[i + 1][j] = false;
  }

  used[i][j] = false;
  return res % M;
}

void solve() {
  bool used[MAX_N][MAX_M];
  memset(used, 0, sizeof(used));  // 初始化为false
  printf("%d\n", rec(0, 0, used));
}
```

这个方法当然无法在规定时间内求出答案。而且，递归函数的参数共有$n \times m \times 2^{nm}$种可能，也无法使用记忆化搜索求解。

但是仔细思考后会发现，实际上参数并没有那么多种可能。首先，由于黑色的格子不能被覆盖，因此used里对应的位置总是false。对于白色的格子，如果现在要在(i, j)位置上放置砖块，那么由于总是从最左上方的可放的格子开始放置，因此对于$(i', j') < (i, j)$（按字典序比较）的(i', j')总有used$[i'][j']$=true成立。

此外，由于砖块的大小为1×2，因此对于每一列j'在满足$(i',j')\geqslant(i,j)$的所有i'中，除了最小的i'之外都满足used[i'][j']=false。因此，不确定的只有每一列里还没查询的格子中最上面的一个，共m个。从而可以把这m个格子通过状态压缩编码进行记忆化搜索，复杂度为$O(n\times m\times 2^m)$。按照之前的状态压缩DP的写法就得到了下面的程序。

```
int dp[2][1 << MAX_M];  // DP数组（滚动数组循环利用）

void solve() {
  int *crt = dp[0], *next = dp[1];
  crt[0] = 1;
  for (int i = n - 1; i >= 0; i--) {
    for (int j = m - 1; j >= 0; j--) {
      for (int used = 0; used < 1 << m; used++) {
        if ((used >> j & 1) || color[i][j]) {
          // 不需要在(i, j)放置砖块
          next[used] = crt[used & ~(1 << j)];
        } else {
          // 尝试2种放法
          int res = 0;
          // 横着放
          if (j + 1 < m && !(used >> (j + 1) & 1) && !color[i][j + 1]) {
            res += crt[used | 1 << (j + 1)];
          }
          // 竖着放
          if (i + 1 < n && !color[i + 1][j]) {
            res += crt[used | 1 << j];
          }
          next[used] = res % M;
        }
      }
      swap(crt, next);
    }
  }
  printf("%d\n", crt[0]);
}
```

专栏 完全匹配的个数

上面的铺砖方案，用图论的语言来说就是一个完美匹配。完美匹配的个数虽然可以像上面一样使用状态压缩 DP 求解，但是复杂度是指数级别的，如果问题的规模较大则无法求解。实际上，平面图的完美匹配的个数可以在多项式时间内求解，所以这个问题也可以高效地求解。有兴趣的读者可以试着思考看看。

3.4.2 矩阵的幂

斐波那契数列

斐波那契数列是由如下递推式定义的数列

$F_0=0$
$F_1=1$
$F_{n+2}=F_{n+1}+F_n$

求这个数列第 n 项的值对 10^4 取余后的结果。

限制条件

- $0 \leqslant n \leqslant 10^{16}$

样例

输入

```
n = 10
```

输出

```
55
```

通过逐项计算这个递推式，可以在$O(n)$的时间内算出答案，不过这个算法效率太低了。对于n的规模如此之大的题目应该如何求解呢？可能有人会认为通过递推式求出通项，就可以求解了。可是斐波那契数列的通项公式是

$$F_n=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

由于式中包含无理数，无法简单求得模10^4之后的结果。况且，在其他问题中有一些很难直接求得通项公式。不过这些情况都可以不求出通项，而用矩阵高效地求出第n项的值。

首先，我们先介绍一下对于斐波那契数列应该如何求解。把斐波那契数列的递推式表示成矩阵就得到了下面的式子

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

记这个矩阵为A，则有

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

因此只要求出A^n就可以求出F_n了。关于A^n的计算可以参考2.6节里的快速幂运算，在$O(\log n)$时间里求出第n项的值。

```
// 用二维vector来表示矩阵
typedef vector<int> vec;
typedef vector<vec> mat;
typedef long long ll;

const int M = 10000;

// 计算A*B
mat mul(mat &A, mat&B) {
  mat C(A.size(), vec(B[0].size()));
  for (int i = 0; i < A.size(); i++) {
    for (int k = 0; k < B.size(); k++) {
      for (int j = 0; j < B[0].size(); j++) {
        C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % M;
      }
    }
  }
  return C;
}

// 计算A^n
mat pow(mat A, ll n) {
  mat B(A.size(), vec(A.size()));
  for (int i = 0; i < A.size(); i++) {
    B[i][i] = 1;
  }
  while (n > 0) {
    if (n & 1) B = mul(B, A);
    A = mul(A, A);
    n >>= 1;
  }
  return B;
}
```

```
// 输入
ll n;

void solve() {
  mat A(2, vec(2));
  A[0][0] = 1; A[0][1] = 1;
  A[1][0] = 1; A[1][1] = 0;
  A = pow(A, n);
  printf("%d\n", A[1][0]);
}
```

更一般地，对于m项递推式，如果记递推式为

$$a_{n+m}=\sum_{i=0}^{m-1}b_i a_{n+i}$$

则可以把递推式写成如下矩阵形式

$$\begin{pmatrix} a_{n+m} \\ a_{n+m-1} \\ \vdots \\ a_{n+1} \end{pmatrix}=\begin{pmatrix} b_{m-1} & \cdots & b_1 & b_0 \\ 1 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 \end{pmatrix}\begin{pmatrix} a_{n+m-1} \\ a_{n+m-2} \\ \vdots \\ a_n \end{pmatrix}$$

通过计算这个矩阵的n次幂，就可以在$O(m^3 \log n)$的时间内计算出第n项的值。不过，如果递推式里含有常数项则稍微复杂一些，需变成如下形式

$$\begin{pmatrix} a_{n+m} \\ a_{n+m-1} \\ \vdots \\ a_{n+1} \\ 1 \end{pmatrix}=\begin{pmatrix} b_{m-1} & \cdots & b_1 & b_0 & c \\ 1 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} a_{n+m-1} \\ a_{n+m-2} \\ \vdots \\ a_n \\ 1 \end{pmatrix}$$

专栏　更快地计算递推式

事实上，要求 m 项递推式的第 n 项的值可以不使用矩阵，而是使用初项的线性表示，通过快速幂在 $O(m^2 \log n)$的时间内求出答案。有兴趣的读者可以试着思考看看。

Blocks（POJ No.3734）

给定 N 个方块排成一列。现在要用红、蓝、绿、黄四种颜色的油漆给这些方块染色。求染成红色的方块和染成绿色的方块的个数同时为偶数的染色方案的个数，输出对 10007 取余后的答案。

限制条件

- $1 \leqslant N \leqslant 10^9$

样例 1

输入

```
N = 1
```

输出

```
2（蓝、黄）
```

样例 2

输入

```
N = 2
```

输出

```
6（红红、蓝蓝、蓝黄、绿绿、黄蓝、黄黄）
```

让我们试着从左边开始依次染色。设染到第 i 个方块为止，红绿都是偶数的方案数为 a_i，红绿恰有一个是偶数的方案数为 b_i，红绿都是奇数的方案数为 c_i。这样，染到第 $i+1$ 个方块为止，红绿都是偶数的方案数有如下两种可能

- 到第 i 个方块为止红绿都是偶数，并且第 $i+1$ 个方块染成了蓝色或者黄色
- 到第 i 个方块为止红绿恰有一个是奇数，并且第 $i+1$ 个方块染成了奇数个对应的那种颜色，因此有如下递推式

$$a_{i+1}=2\times a_i+b_i$$

同样地，有

$$b_{i+1}=2\times a_i+2\times b_i+2\times c_i$$
$$c_{i+1}=b_i+2\times c_i$$

把a_i, b_i, c_i的递推式用矩阵表示如下

$$\begin{pmatrix} a_{i+1} \\ b_{i+1} \\ c_{i+1} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix}$$

因此就有

$$\begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{pmatrix}^i \begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{pmatrix}^i \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

这样，用和之前一样的方法计算矩阵的幂就可以求出这个问题的答案了。

```
// 输入
int N;

void solve() {
  mat A(3, vec(3));
  A[0][0] = 2; A[0][1] = 1; A[0][2] = 0;
  A[1][0] = 2; A[1][1] = 2; A[1][2] = 2;
  A[2][0] = 0; A[2][1] = 1; A[2][2] = 2;
  A = pow(A, N);
  printf("%d\n", A[0][0]);
}
```

图中长度为 k 的路径的计数

给定一个 n 个顶点，边长为 1 的有向图的邻接矩阵。求这个图里长度为 k 的路径的总数。路径中同一条边允许通过多次。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant k \leqslant 10^9$

样例

输入

```
n = 4
k = 2
图如下图所示
```

输出

```
6（1→2→3，1→3→4，2→3→4，3→4→1，4→1→2，4→1→3）
```

假设从u出发，到v的长度为k的路径的总数为$G_k[u][v]$。首先，k=1时值和边数相等，因此G_1就等于图的邻接矩阵。假设我们已经得到了G_{k_1}和G_{k_2}。则有

$$G_{k_1+k_2}[u][v]=\sum_{w=1}^{n}G_{k_1}[u][w]\times G_{k_2}[w][v]$$

表示成矩阵的积的形式即为

$$G_{k_1+k_2}=G_{k_1}G_{k_2}$$

因此，可以使用矩阵的幂表示出

$$G_k=G_1^k$$

这个算法的复杂度是$O(n^3\log n)$。

Matrix Power Series （POJ No.3233）

给定 $n\times n$ 的矩阵 A 和正整数 k 和 m。求矩阵 A 的幂的和。

$$S=A+A^2+\cdots+A^k$$

输出 S 的各元素对 M 取余后的答案。

限制条件

- $1\leqslant n\leqslant 30$
- $1\leqslant k\leqslant 10^9$
- $1\leqslant M\leqslant 10^4$

样例

输入

```
n = 2
k = 2
M = 4
A = {{0, 1}, {1, 1}}
```

输出

```
{{1, 2}, {2, 3}}
```

$n\times n$的矩阵的k次幂可以通过快速幂在$O(n^3 \log k)$的时间内求出。但是，本题求的不是k次幂，而是累乘和。如果按顺序逐个加起来，复杂度就变成了$O(n^3 k)$。要计算幂的和，只需按照如下方法计算就可以了（I是$n\times n$的单位矩阵）。

$$\left(\begin{array}{c|c} A & 0 \\ \hline I & I \end{array}\right)$$

此时，如果令

$$S_k=I+A+\cdots+A^{k-1}$$

则有

$$\left(\begin{array}{c} A^k \\ \hline S_k \end{array}\right)=\left(\begin{array}{c|c} A & 0 \\ \hline I & I \end{array}\right)\left(\begin{array}{c} A^{k-1} \\ \hline S_{k-1} \end{array}\right)=\left(\begin{array}{c|c} A & 0 \\ \hline I & I \end{array}\right)^k\left(\begin{array}{c} I \\ \hline 0 \end{array}\right)$$

因此，通过计算这个矩阵的k次幂就可以求出A的累乘和。时间复杂度为$O(n^3 \log k)$。

```
// 输入
int n, k;
mat A;

void solve() {
  mat B(n * 2, vec(n * 2));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      B[i][j] = A[i][j];
    }
    B[n + i][i] = B[n + i][n + i] = 1;
  }
  B = pow(B, k + 1); // I+A+A^2+...+A^k
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
```

```
        int a = B[n + i][j] % M;
        // 减去I
        if (i == j) a = (a + M - 1) % M;
        printf("%d%c", a, j + 1 == n ? '\n' : ' ');
      }
    }
  }
```

3.4.3 利用数据结构高效求解

Minimizing Maximizer （POJ No.1769）

Maximizer 是一个接受 n 个数作为输入，并输出它们的最大值的装置。这个装置由 m 个叫做 Sorter 的装置依次连接而成。第 k 个 Sorter 把第 k–1 个 Sorter 的输出作为输入，然后将第 s_k 到第 t_k 个值进行排序后，保持其余部分不变输出。Maximizer 的输入就是第一个 Sorter 的输入，最后一个 Sorter 输出的第 n 个值就是 Maximizer 的输出。从组成 Maximizer 的 Sorter 中去掉几个之后，Maximizer 有可能还可以正常工作。现在给定 Sorter 的序列，求其中的最短的一个子序列（可以不连续）使得 Maximizer 仍然可以正常工作。

限制条件

- $2 \leqslant n \leqslant 50000$
- $1 \leqslant m \leqslant 500000$
- $1 \leqslant s_k < t_k \leqslant n$

样例

输入

```
n = 40
m = 6
(s, t) = {(20, 30), (1, 10), (10, 20), (20, 30), (15, 25), (30, 40)}
```

输出

```
4(由2, 3, 4, 6号的Sorter组成的子序列)
```

首先，我们考虑一下在什么样的情况下可以正常工作。假设输入的第i个数是应该输出的最大值。此时，在第一个满足$s_k \leqslant i \leqslant t_k$的Sorter的输出中，这个值被移动到了第$t_k$个位置。

接下去，在第一个满足$s_{k'} \leqslant t_k \leqslant t_{k'}$且$k'>k$的Sorter的输出中，这个值又被移动到了第$t_{k'}$个。不断重复这样的操作，如果最后可以被移动到第$n$个，那么就表示Maximizer可以正常工作。

i=12的情况

从这个图中也里可以看出，只要对i=1的情况可以正常工作，那么对于任意的i都可以正常工作。因此，我们不妨假设输入的第一个数是应该输出的最大值，然后考虑如下DP。

$dp[i][j]$:=到第i个Sorter为止，最大值被移动到第j个位置所需要的最短的子序列的长度（INF表示不存在这样的序列）

$$dp[0][1]=0$$
$$dp[0][j]=INF(j>1)$$
$$dp[i+1][j]=\begin{cases} dp[i][j](t_i \neq j) \\ \min(dp[i][j],\min\{dp[i][j'] \mid s_i \leqslant j' \leqslant t_i\}+1)(t_i=j) \end{cases}$$

由于这个DP的复杂度是$O(nm)$的，仍然无法在规定时间内求出答案。但是对于$t_i \neq j$时有$dp[i+1][j]=dp[i][j]$。如果我们使用同一个数组不断对自己更新又会怎么样呢？

$dp[j]$:=最大值被移动到第j个位置所需要的最短的子序列的长度（INF表示不存在这样的序列）

进行如下初始化：$dp[1]=0$, $dp[j]:=INF$ $(j>1)$
对于每个i，这样更新
$dp[t_i]=min(dp[t_i],min\{dp[j'] \mid s_i \leqslant j' \leqslant t_i\}+1)$

这样，对于每个i都只需更新一个值就可以了。但是可能会认为求解最小值时，最坏情况下仍然要$O(n)$的时间，最后复杂度还是$O(nm)$。不过，如果使用之前介绍的线段树来维护，就可以在$O(m \log n)$的时间内求解了。

```
// 输入
int n, m;
int s[MAX_M], t[MAX_M];

int dp[MAX_N + 1];  // DP数组

void solve() {
```

```
    rmq_init(n);  // 初始化线段树
    fill(dp, dp + n + 1, INF);
    dp[1] = 0;
    update(1, 0);
    for (int i = 0; i < m; i++) {
      int v = min(dp[t[i]], query(s[i], t[i] + 1) + 1);
      dp[t[i]] = v;
      update(t[i], v);
    }
    printf("%d\n", dp[n]);
  }
```

有时，选择合适的数据结构对DP进行优化，可以降低计算的复杂度。

3.5 借助水流解决问题的网络流

➠本节将围绕最大流和最小费用最大流等问题，介绍图上的网络流。网络流具有各种各样的性质和应用，程序设计竞赛当中也常出现相关的题目。

3.5.1 最大流

最大传输量

网络中有两台计算机 s 和 t，现在想从 s 传输数据到 t。该网络中一共有 N 台计算机，其中一些计算机之间连有一条单向的通信电缆，每条通信电缆都有对应的 1 秒钟内所能传输的最大数据量。当其他计算机之间没有数据传输时，在 1 秒钟内 s 最多可以传送多少数据到 t?

把计算机当作顶点，把连接计算机的通信电缆当作边，就可以把这个网络当作一个有向图来考虑了。图中的每条边 $e \in E$ 都有对应的最大可能的数据传输量 $c(e)$。这样，就可以把问题转为如下形式。

- 记每条边对应的实际数据传输量为 $f(e)$。
- 传输量应该满足如下限制。

  $0 \leqslant f(e) \leqslant c(e)$
- 数据在传输过程中既不会增加也不会减少，收到的数据量和发出的数据量应该相等。

对任意 $v \in V \setminus \{s, t\}$ 都有 $\sum_{e \in \delta_-(v)} f(e) = \sum_{e \in \delta_+(v)} f(e)$

■ 目标是最大化从s发出的数据量$\sum_{e\in\delta+(s)} f(e)$。

我们称使得传输量最大的f为最大流，而求解最大流的问题为最大流问题。此外，我们称c为边的容量，f为边的流量，s为源点(source)，t为汇点(sink)。那么，这个问题应该如何求解呢？首先考虑下面这样的贪心算法。

(1) 找一条s到t的只经过$f(e)<c(e)$的边的路径；
(2) 如果不存在满足条件的路径，则结束算法。否则，沿着该路径尽可能地增加$c(e)$，返回第(1)步。

将该算法运用于样例，就得到了如下结果。

那么，这样所得到的是最大流吗？事实上，如果采用下图所示的方案，可以得到更优的结果，于是可以知道这个贪心算法是不正确的。

那么，贪心算法得到的结果是10，而上图得到的结果是11。为了找出二者的区别，不妨来看看它们的流量的差。

通过对流量的差的观察可以发现，我们通过将原先得到的流给推回去（图中的–1部分），而得到了新的流。因此，可以试着在之前的贪心算法中加上这一操作，将算法进行如下改进。

(1) 只利用满足$f(e)<c(e)$的e或者满足$f(e)>0$的e对应的反向边rev(e)，寻找一条s到t的路径。
(2) 如果不存在满足条件的路径，则结束。否则，沿着该路径尽可能地增加流，返回第(1)步。

再将改进后的贪心算法运用于样例。

这样就得到了11这一结果。那么，这个算法总能求得最大流吗？答案是肯定的。我们将在下一小节通过最小割来说明这一点。上面这个求解最大流的算法叫做Ford-Fulkerson算法。另外，我们称在(1')中所考虑的$f(e)<c(e)$的e和满足$f(e)>0$的e对应的反向边rev(e)所组成的图为残余网络，并称残余网络上的s-t路径为增广路。

下面是一个Ford-Fulkerson算法的邻接表实现的例子。这里没有保存$f(e)$的值，取而代之的是直接改变$c(e)$的值。

```
// 用于表示边的结构体（终点、容量、反向边）
struct edge { int to, cap, rev; };

vector<edge> G[MAX_V];   // 图的邻接表表示
bool used[MAX_V];        // DFS中用到的访问标记
```

```
// 向图中增加一条从s到t容量为cap的边
void add_edge(int from, int to, int cap) {
  G[from].push_back((edge){to, cap, G[to].size()});
  G[to].push_back((edge){from, 0, G[from].size() - 1});
}

// 通过DFS寻找增广路
int dfs(int v, int t, int f) {
  if (v == t) return f;
  used[v] = true;
  for (int i = 0; i < G[v].size(); i++) {
    edge &e = G[v][i];
    if (!used[e.to] && e.cap > 0) {
      int d = dfs(e.to, t, min(f, e.cap));
      if (d > 0) {
        e.cap -= d;
        G[e.to][e.rev].cap += d;
        return d;
      }
    }
  }
  return 0;
}

// 求解从s到t的最大流
int max_flow(int s, int t) {
  int flow = 0;
  for (;;) {
    memset(used, 0, sizeof(used));
    int f = dfs(s, t, INF);
    if (f == 0) return flow;
    flow += f;
  }
}
```

记最大流的流量为F，那么Ford-Fulkerson算法最多进行F次深度优先搜索，所以其复杂度为$O(F|E|)$。不过，这是一个很松的上界，达到这种最坏复杂度的情况几乎不存在。所以在多数情况下，即便通过估算得到的复杂度偏高，实际运用当中也还是比较快的。

3.5.2　最小割

为了证明Ford-Fulkerson算法所求得的确实是最大流，我们首先介绍割这一概念。所谓图的割，指的是对于某个顶点集合$S \subseteq V$，从S出发指向S外部的那些边的集合，记为割$(S, V \setminus S)$。这些边的容量之和被称为割的容量。如果有$s \in S$，而$t \in V \setminus S$，那么此时的割又称为s-t割。如果将网络中s-t割所包含的边都删去，也就不再有从s到t的路径了。因此，可以考虑一下如下问题。

对于给定网络，为了保证没有从s到t的路径，需要删去的边的总容量的最小值是多少？

最小割

该问题又被称为最小割问题。事实上，这个问题与之前的最大流问题有着很深的联系。

首先，让我们来考虑一下任意的s-t流f和任意的s-t割$(S, V\setminus S)$。因为有(f的流量)=(s的出边的总流量)，而对$v\in S\setminus\{s\}$又有(v的出边的总流量)=(v的入边的总流量)，所以有(f的流量)=(S的出边的总流量)−(S的入边的总流量)。由此可知(f的流量)≤(割的容量)。

接下来，让我们来考虑通过Ford-Fulkerson算法所求得的流f'。记流f'对应的残余网络中从s可达的顶点v组成的集合为S，因为f'对应的残余网络中不存在s-t路径，因此$(S, V\setminus S)$就是一个s-t割。此外，根据S的定义，对包含在割中的边e应该有$f'(e)=c(e)$，而对从$V\setminus S$到S的边e应该有$f'(e)=0$。因此，(f'的流量)=(S的出边的总流量)−(S的入边的总流量)=(割的容量)，再由之前的不等式可以知道，f'即是最大流。

于是我们证明了Ford-Fulkerson算法的正确性。同时还推导出了最大流等于最小割这一重要性质。该性质又被称为最大流最小割定理。根据该定理，我们就可以直接利用求解最大流问题的算法来求解最小割问题了。事实上，也常会遇到将问题规约到图的最小割来求解的题目。此外，由Ford-Fulkerson算法的正确性可以知道，如果所有边的容量都是整数，那么最大流和最小割也是整数。

专栏　最大流的各种变体

■ 多个源点和汇点的情况

我们已经介绍了如何求解恰有一个源点和一个汇点的网络流。那么，如果有多个源点和汇点，并且它们都有对应的最大流出容量和流入容量限制时该怎么做呢？答案很简单，只要增加一个超级源点s和一个超级汇点t，从s向每个源点连一条容量为对应最大流出容量的边，从每个汇点向t连一条容量为对应最大流入容量的边。不过，如果源和汇之间存在对应关系（从不同源点流出的流要流入指定的汇点）时，是无法这样求解的。这种情况被称为多物网络流问题，尚未有已知的高效算法，这类问题也几乎不会出现在程序设计竞赛当中。

■ 无向图的情况

下面来考虑无向图的情况。此时的容量表示的是两个方向流量之和的上界。不过，如果两个方向都有流量，则与它们相互抵消之后是等价的，所以可以知道最大流中没有必要在两个方向都有流量。因此把无向图中容量为 c 的一条边当作有向图中两个方向各有一条容量为 c 的两条边，就能够得到同样的结果。

■ 顶点上也有容量限制的情况

图中不光边上有容量限制，途中经过的顶点也有总流入量和总流出量的限制的情况应该如何处理呢？此时，我们可以把每个顶点拆成两个。拆点之后得到入顶点和出顶点，将指向原先顶点的边改成指向入顶点，将从原先顶点指出的边改成从出顶点指出。并且，再从入顶点向出顶点连容量为原先顶点容量的边，就可以把顶点上的容量限制转为边上的容量限制了。

■ 有最小流量限制的情况

接下来，我们考虑一下不光有最大流量限制 $c(e)$，还有最小流量限制 $b(e)$的情况($b(e)\leqslant f(e)\leqslant c(e)$)。令 $f'(e)=f(e)-b(e)$，就可以转为只有最大流量限制 $0\leqslant f'(e)\leqslant c(e)-b(e)$的情况。而此时顶点对应的总流入量和总流出量的关系变为

$$\sum_{e\in\delta_-(v)} f'(e)+b(e)=\sum_{e\in\delta_+(v)} f'(e)+b(e)$$

这可以看作是有一个最大流出量为 $\sum_{e\in\delta_-(v)} b(e)$ 的源点和一个最大流入量为 $\sum_{e\in\delta_+(v)} b(e)$ 的汇点与之相连。于是，可以增加新的源点 S 和汇点 T，对于每条边 $e=(u,v)$，令 $c'(e)=c(e)-b(e)$，

并从 S 向 v 连一条容量为 $b(e)$的边，从 u 向 T 连一条容量为 $b(e)$的边，并从 S 向 s 连一条容量为 ∞ 的边，从 t 向 T 连一条容量为 ∞ 的边，这样就转为了没有最小流量限制的情况。如果新图中从 S 到 T 的最大流流量为 F'，那么原图中的最大流流量为 $F=F'-\sum_{e\in E} b(e)$。不过，原图的下限限制未必能够满足时，应该在 S 与 s、T 与 t 之间连边之前，检查从 S 到 T 的最大流流量是否为 $\sum_{e\in E} b(e)$。如果不是满流，则原上下界网络流问题没有可行解。

■ 图发生部分变化的情况

在某些问题中，求完某个图的最大流之后，需要对原图中的一部分做一些变化，再对新图求最大流。这种情况下，有时不需要重新计算最大流，而可以重复利用前一步的结果，高效地求出新的最大流。

首先，让我们来考虑边 $e=(u,v)$的容量增加的情况。回想一下之前最大流的算法就会知道，通过在任意的非最大流 f 上不断寻找增广路增广就能得到最大流。因此，只要在原图的最大流 f 的基础上，不断寻找增广路增广，就可以求得新图的最大流。当有多条边的容量同时增加的情况也一样，可以在原图的流的基础上进行求解。

接下来，让我们来考虑一下边 $e=(u,v)$的容量减小 1 的情况。如果原图的最大流中，有 $f(e)\leq c(e)-1$ 的话，那么它也是新图的最大流。否则，如果 $f(e)=c(e)$，为了让新图满足流量限制，需要将多出部分的流退回去。假如 f 的残余网络中存在从 u 到 v 的路径，那么就可以沿这条路径增广 1 并把 $f(e)$减小 1，而保持最大流流量不变。否则，需要找 $t\to v$ 和 $u\to s$ 的路径，沿它们增广 1 并把 $f(e)$减小 1 之后，最大流流量也减小 1。当有多条边的容量同时减小或减小量不止 1 时，也可以类似处理。在求字典序最小的最大流之类的问题中会用到这种技巧。

■ 容量为负数的情况

虽然在网络流问题中，通常不会有容量为负数的边。但是将问题转化为最小割时，有可能出现容量为负数的边。一般情况下，我们不能利用最大流算法来求解包含负容量边的图的最小割，也没有已知的有效算法。但有些情况下，可以采取适当的变形而避免出现负容量边。这类例子可以参考 3.7 节的 Wi-fi Towers 等问题。

专栏 更高效的最大流算法

之前介绍的 Ford-Fulkerson 算法的复杂度为 $O(F|E|)$。大多数情况下，这个算法已经足够高效了。但当顶点数或最大流流量非常大时，这个算法就显得不够快了。事实上，还有许许多多不同的求解最大流问题的算法，在此我们介绍一下实现起来比较简单，实际运行也比较快速的 Dinic 算法。[①]

Ford-Fulkerson 算法是通过深度优先搜索寻找增广路，并沿着它增广。与之相对，Dinic 算法总是寻找最短的增广路，并沿着它增广。因为最短增广路的长度在增广过程中始终不会变短，所以无需每次都通过宽度预先搜索来寻找最短增广路。我们可以先进行一次宽度优先搜索，然后考虑由近距离顶点指向远距离顶点的边所组成的分层图，在上面进行深度优先搜索寻找最短增广路。如果在分层图上找不到新的增广路了[②]，则说明最短增广路的长度确实变长了，或不存在增广路了，于是重新通过宽度优先搜索构造新的分层图。每一步构造分层图的复杂度为 $O(|E|)$，而每一步完成之后最短增广路的长度都会至少增加 1，由于增广路的长度不会超过 $|V|-1$，因此最多重复 $O(|V|)$ 步就可以了。

另外，在每次对分层图进行深度优先搜索寻找增广路时，如果避免对一条没有用的边进行多次检查[③]，就可以保证复杂度为 $O(|E||V|)$，这样总的复杂度就是 $O(|E||V|^2)$。不过，该算法在实际应用中速度非常快，很多时候即便图的规模比较大也没有问题。

```
// 用于表示边的结构体（终点、容量、反向边）
struct edge { int to, cap, rev; };

vector<edge> G[MAX_V]; // 图的邻接表表示
int level[MAX_V]; // 顶点到源点的距离标号
int iter[MAX_V]; // 当前弧，在其之前的边已经没有用了

// 向图中增加一条从from到to的容量为cap的边
void add_edge(int from, int to, int cap) {
  G[from].push_back((edge){to, cap, G[to].size()});
  G[to].push_back((edge){from, 0, G[from].size() - 1});
}

// 通过BFS计算从源点出发的距离标号
void bfs(int s) {
  memset(level, -1, sizeof(level));
  queue<int> que;
  level[s] = 0;
  que.push(s);
  while (!que.empty()) {
```

① 最大流算法主要有两类，增广路算法和预流推进算法，本书中介绍的几种算法都属于增广路算法。最大流算法有很多，它们有不同的复杂度，不同的优缺点和对不同的图不同的实际运行效率，有兴趣的读者可以查阅有关资料。——译者注

② 此时我们得到了分层图所对应的阻塞流(blocking flow)。——译者注

③ 这个优化称作当前弧优化。——译者注。

```
    int v = que.front(); que.pop();
    for (int i = 0; i < G[v].size(); i++) {
      edge &e = G[v][i];
      if (e.cap > 0 && level[e.to] < 0) {
        level[e.to] = level[v] + 1;
        que.push(e.to);
      }
    }
  }
}

// 通过DFS寻找增广路
int dfs(int v, int t, int f) {
  if (v == t) return f;
  for (int &i = iter[v]; i < G[v].size(); i++) {
    edge &e = G[v][i];
    if (e.cap > 0 && level[v] < level[e.to]) {
      int d = dfs(e.to, t, min(f, e.cap));
      if (d > 0) {
        e.cap -= d;
        G[e.to][e.rev].cap += d;
        return d;
      }
    }
  }
  return 0;
}

// 求解从s到t的最大流
int max_flow(int s, int t) {
  int flow = 0;
  for (;;) {
    bfs(s);
    if (level[t] < 0) return flow;
    memset(iter, 0, sizeof(iter));
    int f;
    while ((f = dfs(s, t, INF)) > 0) {
      flow += f;
    }
  }
}
```

3.5.3 二分图匹配

指派问题

有 N 台计算机和 K 个任务。我们可以给每台计算机分配一个任务，每台计算机能够处理的任务种类各不相同。请求出最多能够处理的任务的个数。

这个问题可以像下面这样转化为图论模型来分析。我们可以像下面这样来定义无向二分图$G=(U\cup V, E)$。

> U是代表计算机的顶点集合，V是代表任务的顶点集合，对于任意$u\in U$和$v\in V$，计算机u能够处理任务$v \Leftrightarrow (u, v)\in E$

而G中满足两两不含公共端点的边集合$M\subseteq E$的基数$|M|$的最大值，就是我们所要求的最大的任务个数。

二分图的例子

图论术语中，我们将这种两两不含公共端点的边集合M称为匹配，而元素最多的M则称为最大匹配。当最大匹配的匹配数满足$2|M|=|V|$时，又称为完美匹配。特别地，二分图中的匹配又称为二分图匹配。像这道题目一样，二分图匹配常常在指派问题的模型中出现，也常常在程序设计竞赛中登场。那么这道题目应该如何求解呢？

实际上，可以将二分图最大匹配问题看成是最大流问题的一种特殊情况。不妨对原图作如下变形。

> 将原图中的所有无向边e改成有向边，方向从U到V，容量为1。增加源点s和汇点t，从s向所有的顶点$u\in U$连一条容量为1的边，从所有的顶点$v\in V$向t连一条容量为1的边。

这样变形得到的新图G'中最大s-t流的流量就是原二分图G中最大匹配的匹配数，而U-V之间流量为正的边集合就是最大匹配。该算法的复杂度为$O(|V||E|)$。

转化为最大流问题

```
// 输入
int N, K;
bool can[MAX_N][MAX_K];  // can[i][j]:=计算机i能够处理任务j

void solve() {
```

```
  // 0~N-1: 计算机对应的顶点
  // N~N+K-1: 任务对应的顶点
  int s = N + K, t = s + 1;

  // 在源点和计算机之间连边
  for (int i = 0; i < N; i++) {
    add_edge(s, i, 1);
  }

  // 在任务和汇点之间连边
  for (int i = 0; i < K; i++) {
    add_edge(N + i, t, 1);
  }

  // 在计算机和任务之间连边
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < K; j++) {
      if (can[i][j]) {
        add_edge(i, N + j, 1);
      }
    }
  }

  printf("%d\n", max_flow(s, t));
}
```

利用所有边的容量都是1以及二分图的性质，我们还可以像下面这样将二分图最大匹配算法更简单地实现。

```
int V;                    // 顶点数
vector<int> G[MAX_V];     // 图的邻接表表示
int match[MAX_V];         // 所匹配的顶点
bool used[MAX_V];         // DFS中用到的访问标记

// 向图中增加一条连接u和v的边
void add_edge(int u, int v) {
  G[u].push_back(v);
  G[v].push_back(u);
}

// 通过DFS寻找增广路
bool dfs(int v) {
  used[v] = true;
  for (int i = 0; i < G[v].size(); i++) {
    int u = G[v][i], w = match[u];
    if (w < 0 || !used[w] && dfs(w)) {
      match[v] = u;
      match[u] = v;
      return true;
    }
  }
  return false;
}
```

```
// 求解二分图的最大匹配
int bipartite_matching() {
  int res = 0;
  memset(match, -1, sizeof(match));
  for (int v = 0; v < V; v++) {
    if (match[v] < 0) {
      memset(used, 0, sizeof(used));
      if (dfs(v)) {
        res++;
      }
    }
  }
  return res;
}
```

3.5.4　一般图匹配

结对子

2*N* 个学生两两结对子。每个学生都只想和自己的朋友结对子。给定学生之间的朋友关系，求最多能结多少对对子。

如果把这个当作一个以学生为顶点，朋友关系为边的图，就可以把这个问题转为求对应的图的最大匹配数的问题。之前的问题顶点有计算机和任务之分，所以得到的是二分图，而这里得到的却不一定是二分图。这种问题被称为一般图匹配问题，它不能像二分图一样转为最大流问题进行求解。求解一般图匹配问题可以使用Edmonds算法等高效的算法。只不过Edmonds算法的实现较为复杂，所以程序设计竞赛中较少出现这类问题。如果把模型转化成了匹配问题，可以先看看事实上是否是二分图匹配。

如果确实不是二分图，而用其他方法又可能行不通时，我们可以用利用下面介绍的Tutte矩阵来计算一般图最大匹配的匹配数。

对无向图$G=(V, E)$的每一条边随意赋予方向得到有向图$G'=(V, E')$。并对每条边$e\in E'$都关联一个变量x_e。则Tutte矩阵是一个如下定义的$V\times V$的矩阵$T=(t_{u,v})$。

$$t_{u,v}=\begin{cases} x_{(u,v)}((u,v)\in E') \\ -x_{(v,u)}((v,u)\in E') \\ 0\ (\text{其他}) \end{cases}$$

可以证明，此时

G没有完美匹配 $\Leftrightarrow$ 行列式$\det(T)$恒等于0

我们简单介绍一下如何证明。根据定义，行列式的值可以展开为所有排列对应的项的和。将排列看成一个有向图，如果该图包含奇圈的话，那么这个排列所对应的项就会和将奇圈反向后的排列所对应的项相互抵消。因此只要考虑由偶圈组成的排列就好了。如果其中有非零项的话，只要在对应的偶圈上间隔取边就得到了一个完美匹配。并且，该项不会被其他任何项相互抵消。反之，如果存在完美匹配的话，那么通过交换相互匹配的点对而得到的排列所对应的项非零，且该项不会被其他任何项相互抵消。

由此我们还可以证明T的秩等于最大匹配的顶点数。于是我们可以利用随机算法，将随机数代入x_e，从而求得一般图的匹配数。

3.5.5 匹配、边覆盖、独立集和顶点覆盖

我们已经了解了图的匹配的概念，此外还有几个相关的有用的概念，在此我们再介绍除匹配之外的三个新的概念。记图$G=(V, E)$。

匹配……在G中两两没有公共端点的边集合$M\subseteq E$
边覆盖……G中的任意顶点都至少是F中某条边的端点的边集合$F\subseteq E$
独立集……在G中两两互不相连的顶点集合$S\subseteq V$
顶点覆盖……G中的任意边都有至少一个端点属于S的顶点集合$S\subseteq V$

例如在下图中，最大匹配为$\{e1, e3\}$，最小边覆盖为$\{e1, e3, e4\}$，最大独立集为$\{v2, v4, v5\}$，最小顶点覆盖为$\{v1, v3\}$。

此外，它们之间还满足有如下关系。

(a) 对于不存在孤立点的图，|最大匹配|+|最小边覆盖|=|V|
(b) |最大独立集|+|最小顶点覆盖|=|V|

证明并不复杂，读者不妨试着思考一下。(a)中可以通过向最大匹配中加边而得到最小边覆盖。而(b)中有$X\subseteq V$是G的独立集 $\Leftrightarrow V\setminus X$是$G$的顶点覆盖。

借助这些关系，对于最大匹配与最小边覆盖，最大独立集与最小顶点覆盖，我们只要能求解其中一个问题也就能够求解另一个问题了。刚刚我们介绍过了求解最大匹配的方法。利用这一方法我

们也能够求解最小边覆盖问题。但是，要怎么求最大独立集或最小顶点覆盖呢？事实上，这两个问题是NP困难的，没有已知的高效算法。不过，对于二分图而言，有如下等式成立。

(*c*) |最大匹配|=|最小顶点覆盖|

对于二分图$G=(U\cup V, E)$，在通过最大流求解最大匹配所得到的残留网络中，令S=(从s不可达的属于U的顶点)∪(从s可达的属于V的顶点)，则S就是G的一个最小顶点覆盖。

因此，我们可以高效地求解二分图的最大独立集和最小顶点覆盖，事实上，这类问题也常常出现在程序设计竞赛当中。相对的，如果把问题转为一般图的最大独立集或最小顶点覆盖，则无法直接、高效地求解，有必要从其他角度重新思考。

3.5.6　最小费用流

最小传输费用

网络中有两台计算机 s 和 t，现在每秒钟要从 s 传输大小为 F 的数据到 t。该网络中一共有 N 台计算机，其中一些计算机之间连有一条单向的通信电缆，每条通信电缆都有对应的 1 秒钟内所能传输的最大数据量。此外，每条通信电缆还有对应的传输费用，单位传输费用为 d 的通信电缆每秒传输大小为 x 的数据，需要花费的费用为 dx。请问传输数据所需的最小费用。

首先把问题转化为图。这里可以把计算机当作顶点，把通信电缆当作边，从而得到一个有向图。每条边e都有容量$c(e)$和费用$d(e)$。而题目的目标是在保证从s向t有流量为F的流的前提下，要使费用$\sum_e (f(e)\times d(e))$最小。

这就是在最大流问题的网络中，给边新加上了费用，而求的不再是流量的最大值，而是流量为F时费用的最小值。这类问题叫做最小费用流问题。

接下来，让我们来考虑一下这类问题的解法。求解最大流时，我们在残余网络上不断贪心地增广而得到了最大流。现在边上多了费用，如果我们在残余网络上总是沿着最短路增广又如何呢？此

时，残余网络中的反向边的费用应该是原边费用的相反数，以保证过程是可逆而正确的。因为有负权边，所以就不能用 Dijkstra算法求最短路了，而需要用Bellman-Ford算法。

对样例进行的操作

```
// 用于表示边的结构体（终点、容量、费用、反向边）
struct edge { int to, cap, cost, rev; };

int V;                                  // 顶点数
vector<edge> G[MAX_V];                  // 图的邻接表表示
int dist[MAX_V];                        // 最短距离
int prevv[MAX_V], preve[MAX_V];         // 最短路中的前驱节点和对应的边

// 向图中增加一条从from到to容量为cap费用为cost的边
void add_edge(int from, int to, int cap, int cost) {
  G[from].push_back((edge){to, cap, cost, G[to].size()});
  G[to].push_back((edge){from, 0, -cost, G[from].size() - 1});
}

// 求解从s到t流量为f的最小费用流
// 如果不能再增广则返回-1
int min_cost_flow(int s, int t, int f) {
```

```
    int res = 0;
    while (f > 0) {
      // 利用Bellman-Ford算法求s到t的最短路
      fill(dist, dist + V, INF);
      dist[s] = 0;
      bool update = true;
      while (update) {
        update = false;
        for (int v = 0; v < V; v++) {
          if (dist[v] == INF) continue;
          for (int i = 0; i < G[v].size(); i++) {
            edge &e = G[v][i];
            if (e.cap > 0 && dist[e.to] > dist[v] + e.cost) {
              dist[e.to] = dist[v] + e.cost;
              prevv[e.to] = v;
              preve[e.to] = i;
              update = true;
            }
          }
        }
      }

      if (dist[t] == INF) {
        // 不能再增广
        return -1;
      }

      // 沿s到t的最短路尽量增广
      int d = f;
      for (int v = t; v != s; v = prevv[v]) {
        d = min(d, G[prevv[v]][preve[v]].cap);
      }
      f -= d;
      res += d * dist[t];
      for (int v = t; v != s; v = prevv[v]) {
        edge &e = G[prevv[v]][preve[v]];
        e.cap -= d;
        G[v][e.rev].cap += d;
      }
    }
    return res;
  }
```

接下来，我们来证明这个算法所求得的确实是最小费用流。那么我们应该如何判断某个流量的流f的费用是否是最小的呢？

假设还有同样流量而费用比f更小的流f'。让我们来观察一下流$f'-f$。在流f中，除s和t以外的顶点的流入量等于流出量，在流f'中亦然。并且，由流f和流f'的流量相同可知，流$f'-f$中所有顶点的流入量都等于流出量，即它是由若干圈组成的。因为流$f'-f$的费用是负的，所以在这些圈中，至少存在一个负圈。也就是说

f是最小费用流 ⇔ 残余网络中没有负圈

利用这一点，我们就可以通过归纳法证明，在该算法中流量为 i 的流 f_i 是具有相同流量的流中费用最小的。首先，对于流量为 0 的流 f_0，其残余网络便是原图，只要原图不含负圈，那么 f_0 就是流量 0 的最小费用流。假设流量为 i 的流 f_i 是最小费用流，并且下一步我们求得了流量为 $i+1$ 的流 f_{i+1}。此时，$f_{i+1}-f_i$ 就是 f_i 对应的残余网络中 s 到 t 的最短路。

假设 f_{i+1} 不是最小费用流，即存在费用更小的流 f'_{i+1}。$f'_{i+1}-f_i$ 中除 s 和 t 以外的顶点的流入量等于流出量，因而是由一条从 s 到 t 的路径和若干圈组成的。又有 $f_{i+1}-f_i$ 是一条从 s 到 t 的最短路，而 f'_{i+1} 的费用比 f_{i+1} 还要小，所以 $f'_{i+1}-f_i$ 中至少含有一个负圈，这与 f_i 是最小费用流矛盾。所以，f_{i+1} 也是最小费用流，根据归纳法，对任意的 i 都有 f_i 是最小费用流。

另外，由最大流算法的正确性可知，如果原图存在流量不小于F的流的话，那么这个算法也能够得到流量为F的流。综上，我们证明了最小费用流算法的正确性。

该算法最多执行F次Bellman-Ford算法，所以其复杂度为$O(F|V||E|)$。能够对算法做适当优化从而降低复杂度吗？答案是肯定的。通过导入势的概念，我们可以改用Dijkstra算法来求解最短路。下面我们就来介绍这种方法。

这里的势，指的是给每个顶点赋予的一个标号$h(v)$，在这个势的基础上，将边$e=(u,v)$的长度变为$d'(e)=d(e)+h(u)-h(v)$。于是从d'中的s-t路径的长度中减去常数$h(s)-h(t)$，就得到了d中对应路径的长度，因此d'中的最短路也就是d中的最短路。所以，如果合理地选取势，使得对所有的e都有$d'(e)\geqslant 0$的话，我们就可以在d'中用Dijkstra算法求最短路，从而得到d的最短路。对于任意不含负圈的图，我们可以通过取$h(v)$=(s到v的最短距离)做到这一点。这是因为对于边$e=(u,v)$有

$$(s\text{到}v\text{的最短距离})\leqslant(s\text{到}u\text{的最短距离})+d(e)$$

于是有

$$d'(e)=d(e)+h(u)-h(v)\geqslant 0$$

下面来考虑如何依次更新流量为i的最小费用流f_i及其对应的势h_i，来求出最小费用流。首先，定义以下变量。

$f_i(e)$ .....流量为i的最小费用流中边e的流量
$h_i(v)$ .....f_i的残余网络中s到v的最短距离
$d_i(e)$ .....考虑势h_i后边e的长度

如果图中不含负圈，可以将 $f_i(e)$初始化为 0。如果图中也没有负权边的话，还可以直接用 Dijkstra 算法计算 h_0。求得了 f_i 和 h_i 之后，要如何求 f_{i+1} 和 h_{i+1} 呢？通过沿着 f_i 的残余网络中 s 到 t 的最短路增广，我们就得到了 f_{i+1}。只要在求得了 h_i 后，寻找一条只经过那些 $d_i(e)=0$ 的边的 s 到 t 的路径，就可以轻松办到。为了求 h_{i+1}，我们需要求 f_{i+1} 的残余网络上的最短路。利用势 h_i，这可以通过 Dijkstra 算法办到。我们通过下面的论述来说明这一点。

考虑f_{i+1}的残余网络中的边$e=(u, v)$。如果e也是f_i的残余网络中的边的话，那么根据h的定义有$d_i(e)$

≥ 0。如果e不是f_i的残余网络中的边的话，那么rev(e)一定是f_i的残余网络中s到t的最短路中的边，所以有$d_i(e)=-d_i(\mathrm{rev}(e))=0$。综上，$f_{i+1}$的残余网络中的所有边$e$满足$d_i(e)\geq 0$，因而可以用Dijkstra算法求最短路。

如上所述，只要依次更新f_i和h_i，我们就能够在$O(F|E|\log|V|)$或是$O(F|V|^2)$的时间内求出最小费用流了。在下面的实现示例中，我们在计算h_i时记录了最短路中的前驱节点和对应的边，并利用这些信息同时完成f_{i+1}的计算。

```
typedef pair<int, int> P;  // first保存最短距离，second保存顶点编号

// 用于表示边的结构体（终点、容量、费用、反向边）
struct edge { int to, cap, cost, rev; };

int V; // 顶点数
vector<edge> G[MAX_V];            // 图的邻接表表示
int h[MAX_V];                     // 顶点的势
int dist[MAX_V];                  // 最短距离
int prevv[MAX_V], preve[MAX_V];   // 最短路中的前驱节点和对应的边
// 向图中增加一条从from到to容量为cap费用为cost的边

void add_edge(int from, int to, int cap, int cost) {
  G[from].push_back((edge){to, cap, cost, G[to].size()});
  G[to].push_back((edge){from, 0, -cost, G[from].size() - 1});
}

// 求解从s到t流量为f的最小费用流
// 如果没有流量为f的流，则返回-1
int min_cost_flow(int s, int t, int f) {
  int res = 0;
  fill(h, h + V, 0);  // 初始化h
  while (f > 0) {
    // 使用Dijkstra算法更新h
    priority_queue<P, vector<P>, greater<P> > que;
    fill(dist, dist + V, INF);
    dist[s] = 0;
    que.push(P(0, s));
    while (!que.empty()) {
      P p = que.top(); que.pop();
      int v = p.second;
      if (dist[v] < p.first) continue;
      for (int i = 0; i < G[v].size(); i++) {
        edge &e = G[v][i];
        if (e.cap > 0 && dist[e.to] > dist[v] + e.cost + h[v] - h[e.to]) {
          dist[e.to] = dist[v] + e.cost + h[v] - h[e.to];
          prevv[e.to] = v;
          preve[e.to] = i;
          que.push(P(dist[e.to], e.to));
        }
      }
    }
    if (dist[t] == INF) {
      // 不能再增广
```

```
      return -1;
    }
    for (int v = 0; v < V; v++) h[v] += dist[v];

    // 沿s到t的最短路尽量增广
    int d = f;
    for (int v = t; v != s; v = prevv[v]) {
      d = min(d, G[prevv[v]][preve[v]].cap);
    }
    f -= d;
    res += d * h[t];
    for (int v = t; v != s; v = prevv[v]) {
      edge &e = G[prevv[v]][preve[v]];
      e.cap -= d;
      G[v][e.rev].cap += d;
    }
  }
  return res;
}
```

专栏 最小费用流的各种变体

■ 与最大流相同的变体

最小费用流在面对多个源点和汇点、无向图、顶点上也有容量限制等情况时，也可以采取与之前最大流中同样的方法处理。不过需要注意的是，对于无向图的情况，不能直接用邻接阵来表示图。在把无向图转化为有向图时会产生反向边，而当图中有重边或是方向相反的两条边时，我们都不能用邻接阵表示来求解最小费用流。

对于边上有最小流量限制的情况，虽然也可以采取与最大流中同样的方法处理，不过还有更简单的方法。对 $e=(u,v)$，新加一条边 $e'=(u,v)$，再令 $c'(e)=c(e)-b(e)$、$c'(e')=b(e)$、$d'(e)=d(e)$、$d'(e')=d(e)-M$(一个足够大的常数)，对变形后的新图求解最小费用流，再在结果上加上 $M\times\sum_e b(e)$就好了。这样就把问题转为了没有最小流量限制的情况。

■ 流量任意的情况

在有些题目中，需要计算包含负权边的图中流量任意但费用最小的流。这种情况下，根据最小费用流算法中 $h_{i+1}(v)\geq h_i(v)$的性质，我们只要在 $h_i(t)<0$ 时不断增广就好了。

■ 费用为负数的情况

如果图中含有负权边，那么最初计算势的值时就不能用 Dijkstra 算法，而需要改用 Bellman-Ford 算法。另外，如果图中还有负圈，可以利用 Bellman-Ford 算法找到负圈，并在负圈上尽量增广将其消去。

此外，有些情况下，通过适当的变形也可以避免负权边。比如说，如果已知每次增广所用的边数都是相等的（记为 m），那么通过对所有边的费用加上合适的常数 k 就能够把所有边都

转成非负的，而每一步只从新图中的最短路减去 mk 就得到了原图中最短路的长度（这类例子可以参考后面的二分图最小权匹配等问题）。

对于流量 F 一定的情况，也可以采取与有最小流量限制中类似的变形将负权边除去。新增源点 S 和汇点 T，从 S 向 s 连一条容量为 F 费用为 0 的边，从 t 向 T 连一条容量为 F 费用为 0 的边。对于负权边 $e=(u,v)$，可以让它一开始就已经满流，再从 S 向 v 连一条容量为 $c(e)$ 费用为 0 的边，从 u 向 T 连一条容量为 $c(e)$ 费用为 0 的边。这样变形之后，我们就除去了图中的负权边，而原图流量为 $F+\sum_{负权边\ e}c(e)$ 的最小费用流的费用就等于新图流量为 F 的最小费用流的费用加上 $\sum_{负权边\ e}c(e)\times d(e)$。

■ 目标并非最小化流量 × 费用之和，而是最小化有流量的边的费用之和的情况

乍一看，会觉得这是同一类问题，但该问题却无法通过最小费用流求解。当把问题转成了这个模型时，有必要从别的角度重新思考。

3.5.7　应用问题

Asteroids （POJ No.3041）

在 $N\times N$ 的网格中有 K 颗小行星。小行星 i 的位置是 (R_i, C_i)。现在有一个强力武器能够用一发光束将一整行或一整列的小行星轰为灰烬。想要利用这个武器摧毁所有的小行星最少需要几发光束？

限制条件

- $1\leqslant N\leqslant 500$
- $1\leqslant K\leqslant 10000$
- $1\leqslant R,C\leqslant N$

样例

输入

```
N = 3
```

```
K = 4
(R, C) = {(1, 1), (1, 3), (2, 2), (3, 2)}
```

输出

```
2第一发摧毁(1, 1)和(1, 3)，第二发摧毁(2, 2)和(3, 2)。
```

光束的攻击选择可以是横坐标从x=1到x=N和纵坐标从y=1到y=N，一共2N种。显然，同样的选择没有必要执行多次，而攻击的顺序对结果没有影响，所以总的攻击方案共有2^{2N}种。我们只要在这个解空间中，寻找能够摧毁所有小行星的最小的解就可以了。要破坏某个小行星，只能通过对应水平方向或竖直方向的光束的攻击。利用攻击方法只有两种这一点，我们可以将问题按如下方法转换为图。

把光束当作图的顶点，而把小行星当作连接对应光束的边。这样转换之后，光束的攻击方案即对应一个顶点集合S，而要求攻击方案能够摧毁所有的小行星，也就是图中的每条边都至少有一个属于S的端点。这样一来，问题就转为了求最小的满足上述要求的顶点集合S。

样例所对应的图

这正是最小顶点覆盖的问题。之前我们已经介绍过，最小顶点覆盖问题通常是NP困难的，不过在二分图中等于最大匹配，因而可以高效地求解。事实上，本题中所有顶点可以分成水平方向和竖直方向的攻击选择两类，而每颗小行星所对应的边都分别与一个水平方向和一个竖直方向的顶点相连，所以是二分图。因此，只要运用二分图最大匹配算法，问题就会迎刃而解了。

```
// 输入
int N, K;
int R[MAX_K], C[MAX_K];

void solve() {
  V = N * 2;
  for (int i = 0; i < K; i++) {
    add_edge(R[i] - 1, N + C[i] - 1);
  }
  printf("%d\n", bipartite_matching());
}
```

Evacuation （POJ No.3057）

有一个 X×Y 块区域组成的房间，每块区域可能是墙壁'X'、空区域'.'或门'D'。最外层的区域一定是门或者墙壁，而内部的区域一定没有门。假设这个房间起火了。最开始，每个空区域内都恰好站着一个人，现在它们开始向门移动以安全逃脱。每个人每秒钟可以选择停留在原地或是移动到相邻四个区域中的一个。不过，如果相邻的区域是墙壁，则不能移动。当移动到门时，就安全逃脱了，只不过因为门比较狭窄，每秒钟只能通过一个人。请计算在选取最优逃脱方案时，最后一个人逃脱的最短时间。如果有人无法安全逃脱，则输出"impossible"。

限制条件

- $3 \leq X, Y \leq 12$

样例 1

输入

```
X = 5
Y = 5

XXDXX
X...X
D...X
X...D
XXXXX
```

输出

```
3
```

样例 2

输入

```
X = 5
Y = 12

XXXXXXXXXXXX
X..........D
X.XXXXXXXXXX
X..........X
XXXXXXXXXXXX
```

输出

```
21
```

样例 3

输入

```
X = 5
Y = 5

XDXXX
X.X.D
XX.XX
D.X.X
XXXDX
```

输出

```
impossible
```

以每个门为起点进行宽度优先搜索计算最短路，就可以知道每个人到达每个门所需的最短时间。如果发现有人任何一个门都到不了，那么他就无法逃脱，否则只要花上足够长的时间，总能让所有人都逃脱。那么，该如何求出逃脱所需的最短时间呢？

由于每个门每秒钟只能通过一个人这一限制的存在，我们不能直接将所有人到最近的门的距离中的最大值作为答案。我们不妨想想看是否可以快速的判断所有人能否在时间T以内逃脱。如果能办到的话，通过二分搜索就能够求得最短时间了。考虑某一个门，能在时间t从该门逃脱的人，应该是距离该门t以内的人，并且其中只有一人能够从该门逃脱。每个时间和门的二元组，都确定一个对应的能够从中逃脱的人的集合，而通过计算这个二元组和人组成的二分图的匹配数，我们就可以判断所有人是否都可以逃脱。

```
const int dx[4] = {-1, 0, 0, 1}, dy[4] = {0, -1, 1, 0};

// 输入
int X, Y;
char field[MAX_X][MAX_Y + 1];  // 不要忘记保存\0所需的空间

vector<int> dX, dY;                          // 门的坐标
vector<int> pX, pY;                          // 人的坐标
int dist[MAX_X][MAX_Y][MAX_X][MAX_Y];        // 最近距离

// 判断所有人是否能够在时间t以内安全逃离
bool C(int t) {
  int d = dX.size(), p = pX.size();

  // 0~d-1: 时间1对应的门
  // d~2d-1: 时间2对应的门
  // ...
  // (t-1)d~td-1: 时间t对应的门
  // td~td+p-1: 人
  V = t * d + p;
```

```
    for (int v = 0; v < V; v++) G[v].clear();
    for (int i = 0; i < d; i++) {
      for (int j = 0; j < p; j++) {
        if (dist[dX[i]][dY[i]][pX[j]][pY[j]] >= 0) {
          for (int k = dist[dX[i]][dY[i]][pX[j]][pY[j]]; k <= t; k++) {
            add_edge((k - 1) * d + i, t * d + j);
          }
        }
      }
    }

    return bipartite_matching() == p;
  }

  // 通过BFS计算最近距离
  void bfs(int x, int y, int d[MAX_X][MAX_Y]) {
    queue<int> qx, qy;
    d[x][y] = 0;
    qx.push(x);
    qy.push(y);
    while (!qx.empty()) {
      x = qx.front(); qx.pop();
      y = qy.front(); qy.pop();
      for (int k = 0; k < 4; k++) {
        int x2 = x + dx[k], y2 = y + dy[k];
        if (0 <= x2 && x2 < X && 0 <= y2 && y2 < Y && field[x2][y2] == '.' &&
              d[x2][y2] < 0) {
          d[x2][y2] = d[x][y] + 1;
          qx.push(x2);
          qy.push(y2);
        }
      }
    }
  }

  void solve() {
    int n = X * Y;
    dX.clear(); dY.clear();
    pX.clear(); pY.clear();
    memset(dist, -1, sizeof(dist));

    // 计算到各个门的最近距离
    for (int x = 0; x < X; x++) {
      for (int y = 0; y < Y; y++) {
        if (field[x][y] == 'D') {
          dX.push_back(x);
          dY.push_back(y);
          bfs(x, y, dist[x][y]);
        } else if (field[x][y] == '.') {
          pX.push_back(x);
          pY.push_back(y);
        }
      }
```

```
    }

    // 利用二分搜索求解所有人安全逃脱所需的最短时间
    int lb = -1, ub = n + 1;
    while (ub - lb > 1) {
      int mid = (lb + ub) / 2;
      if (C(mid)) ub = mid;
      else lb = mid;
    }

    if (ub > n) {
      // 逃脱失败
      printf("impossible\n");
    } else {
      printf("%d\n", ub);
    }
  }
```

大家发现了上面的程序中有重复的计算了吗？让我们来考虑一下已知在时间T内无法逃脱，要检查时间$T'>T$的情况。T'所对应的图相比T所对应图，只是增加了其中一侧的顶点和与之对应的边。回想一下二分图最大匹配的算法，它是按顺序从一侧的顶点开始寻找增广路增广。因此，要求T'对应的最大匹配，只要在已求得的T'的最大匹配的基础上，继续从新增加的顶点开始寻找增广路增广就好了。因此，不需要进行二分搜索，更有效的求解最短时间的方法是直接每次将T递增1然后求对应的二分图最大匹配。

```
  void solve() {
    int n = X * Y;
    dX.clear(); dY.clear();
    pX.clear(); pY.clear();
    memset(dist, -1, sizeof(dist));

    // 计算到各个门的最近距离
    for (int x = 0; x < X; x++) {
      for (int y = 0; y < Y; y++) {
        if (field[x][y] == 'D') {
          dX.push_back(x);
          dY.push_back(y);
          bfs(x, y, dist[x][y]);
        } else if (field[x][y] == '.') {
          pX.push_back(x);
          pY.push_back(y);
        }
      }
    }

    // 建图
    int d = dX.size(), p = pX.size();
    for (int i = 0; i < d; i++) {
      for (int j = 0; j < p; j++) {
        if (dist[dX[i]][dY[i]][pX[j]][pY[j]] >= 0) {
```

```
        for (int k = dist[dX[i]][dY[i]][pX[j]][pY[j]]; k <= n; k++) {
          add_edge((k - 1) * d + i, n * d + j);
        }
      }
    }
  }

  // 求解所有人安全逃脱所需的最少时间
  if (p == 0) {
    printf("0\n");
    return;
  }
  int num = 0;
  memset(match, -1, sizeof(match));
  for (int v = 0; v < n * d; v++) {
    memset(used, 0, sizeof(used));
    if (dfs(v)) {
      if (++num == p) {
        printf("%d\n", v / d + 1);
        return;
      }
    }
  }

  // 逃脱失败
  printf("impossible\n");
}
```

Dining （POJ No.3281）

农夫约翰为他的牛准备了 F 种食物和 D 种饮料。每头牛都有各自喜欢的食物和饮料，而每种食物或饮料只能分配给一头牛。最多能有多少头牛可以同时得到喜欢的食物和饮料？

限制条件

- $1 \leqslant N \leqslant 100$
- $1 \leqslant F \leqslant 100$
- $1 \leqslant D \leqslant 100$

样例

输入

```
N = 4
F = 3
D = 3
每头牛喜欢的食物和饮料
```

牛	食　物	饮　料
1	{1,2}	{1,3}
2	{2,3}	{1,2}
3	{1,3}	{1,2}
4	{1,3}	{3}

输出

```
（3给牛2食物2和饮料2，给牛3食物1和饮料1，给牛4食物3和饮料3）
```

如果只是分配食物的话，那么用二分图最大匹配就能够解决了。但遇到这种需要同时给一头牛分配所喜欢的食物和饮料的情况，就不能很好的处理了。不过，我们可以将食物和饮料所对应的两个匹配通过下面的方法联合起来求解。

- 图的顶点在食物对应的匹配中的食物和牛，饮料对应的匹配中的饮料和牛之外，还有一个源点s和一个汇点t。
- 在两个匹配相同的牛之间连一条边，在s和所有食物，t和所有饮料之间连一条边。
- 边的方向为s→食物→牛→牛→饮料→t，容量全都为1。

样例对应的图

这个图中的每一条s-t路径都对应一个牛的食物和饮料的分配方案。我们把食物所对应的牛和饮料所对应的牛拆成两个顶点，之间连一条容量为1的边，就保证了一头牛不会被分配多组食物和饮料。所以，只要计算该图中的最大流，原问题就迎刃而解了。

```
// 输入
int N, F, D;
bool likeF[MAX_N][MAX_F];  // 食物的喜好
bool likeD[MAX_N][MAX_D];  // 饮料的喜好

void solve() {
  // 0~N-1: 食物一侧的牛
```

```
// N~2N-1: 饮料一侧的牛
// 2N~2N+F-1: 食物
// 2N+F~2N+F+D-1: 饮料
int s = N * 2 + F + D, t = s + 1;

// 在s与食物之间连边
for (int i = 0; i < F; i++) {
  add_edge(s, N * 2 + i, 1);
}

// 在饮料和t之间连边
for (int i = 0; i < D; i++) {
  add_edge(N * 2 + F + i, t, 1);
}

for (int i = 0; i < N; i++) {
  // 在食物一侧的牛和饮料一侧的牛之间连边
  add_edge(i, N + i, 1);

  // 在牛和所喜欢的食物或饮料之间连边
  for (int j = 0; j < F; j++) {
    if (likeF[i][j]) add_edge(N * 2 + j, i, 1);
  }
  for (int j = 0; j < D; j++) {
    if (likeD[i][j]) add_edge(N + i, N * 2 + F + j, 1);
  }
}

  printf("%d\n", max_flow(s, t));
}
```

Dual Core CPU （POJ No.3469）

要在由核 A 和核 B 组成的双核 CPU 上运行 N 个模块。模块 i 在核 A 上执行的花费为 A_i，在核 B 上执行的花费为 B_i。有 M 个相互之间需要进行数据交换的模块组合(a_i, b_i)，如果这两个模块在同一个核上执行则没有额外花费，否则会产生 w_i 的花费。请计算执行所有模块所需的最小花费。

限制条件

- $1 \leqslant N \leqslant 20000$
- $1 \leqslant M \leqslant 200000$

样例

输入

```
N = 3
M = 1
```

```
(A, B) = {(1, 10), (2, 10), (10, 3)}
(a, b, w) = {(2, 3, 1000)}
```

输出

```
13（全都在核A上执行）
```

用最小的费用将对象划分成两个集合的问题，常常可以转换成最小割后顺利解决。这道题目就是这类问题的一个经典例子。考虑把N个模块按照在哪个核上执行分成两个集合。

记在核A上执行的模块集合为S，而在核B上执行的模块集合为T。考虑以模块为顶点，并且还有额外的源点s和汇点t的图。我们也记图的s-t割所对应的包含s的顶点集合为S，包含t的集合为T，然后来考察它们的对应关系。此时，花费的总和是

$$\sum_{i\in S} A_i + \sum_{i\in T} B_i + \sum_{a_i\in S, b_i\in T} w_i + \sum_{b_i\in S, a_i\in T} w_i$$

如果我们可以通过合适地建边使得花费的总和等价于割的容量的话，那么为了求最小花费只要求最小割就好了。

那么，让我们一步步来建立满足这个条件的图吧。首先，考虑对应

$$\sum_{i\in S} A_i$$

的边。这是顶点属于S时所产生的费用。所以，只要从每个模块向t连一条容量为A_i的边就可以对应起来。而对于

$$\sum_{i\in T} B_i$$

也只要从s向每个模块连一条容量为B_i的边就好了。

接下来，考虑

$$\sum_{a_i\in S, b_i\in T} w_i$$

这是当a_i属于S而b_i属于T时所产生的费用，只要从模块a_i向模块b_i连一条容量为w_i的边就可以对应起来。

对

$$\sum_{b_i\in S, a_i\in T} w_i$$

亦可进行同样处理。

之后只需求这个图的最小割，也就是最大流就好了。这道题目中，因为图的规模和容量都非常大，建议使用Dinic等比较快速的网络流算法。

样例所对应的图

```
// 输入
int N, M;
int A[MAX_N], B[MAX_N];
int a[MAX_M], b[MAX_M], w[MAX_M];

void solve() {
  int s = N, t = s + 1;

  // 在各个核上执行所产生的费用
  for (int i = 0; i < N; i++) {
    add_edge(i, t, A[i]);
    add_edge(s, i, B[i]);
  }

  // 在不同的核上执行所产生的费用
  for (int i = 0; i < M; i++) {
    add_edge(a[i] - 1, b[i] - 1, w[i]);
    add_edge(b[i] - 1, a[i] - 1, w[i]);
  }

  printf("%d\n", max_flow(s, t));
}
```

Farm Tour （POJ No.2135）

农夫约翰的朋友前来拜访，于是他带领大家参观他的农场。农场里有 N 块地，其中约翰的家在 1 号地，而 N 号地有个很大的仓库。农场内有 M 条道路（双向通行），道路 i 连接着 a_i 号地和 b_i 号地，长度为 c_i。约翰希望按照从家里出发，经过若干块地后达到仓库，然后再返回家中的顺序带朋友参观。如果要求往返不能经过同一条道路两次，求参观路线总长度的最小值。

限制条件

- $1 \leqslant N \leqslant 1000$
- $1 \leqslant M \leqslant 10000$
- $1 \leqslant a_i, b_i \leqslant N$
- $1 \leqslant c_i \leqslant 35000$

样例

输入

```
N = 4
M = 5
(a, b, c) = {(1, 2, 1), (2, 3, 1), (3, 4, 1), (1, 3, 2), (2, 4, 2)}
```

输出

```
6 ( 1→2→4→3→1 )
```

如果只考虑去或者回的情况，那么问题只不过是无向图中两点之间的最短路问题而已。但现在既要去又要回，并且有不能经过相同的道路这一限制。那么，如果先计算去时的最短路，然后将所用的道路删去，再在剩下的图上计算回来时的最短路，这样是否可行呢？应该马上就能找到反例证明该方法不总能得到最优结果吧。于是，我们放弃把问题当作去和回的这种想法，转而将问题当作求从1号顶点到N号顶点的两条没有公共边的路径又如何呢？这样转化之后，就不过是求流量为2的最小费用流了，问题得以轻松解决。

```
// 输入
int N, M;
int a[MAX_M], b[MAX_M], c[MAX_M];

void solve() {
  // 建图
  int s = 0, t = N - 1;
  V = N;
  for (int i = 0; i < M; i++) {
    add_edge(a[i] - 1, b[i] - 1, 1, c[i]);
    add_edge(b[i] - 1, a[i] - 1, 1, c[i]);
  }

  printf("%d\n", min_cost_flow(s, t, 2));
}
```

Evacuation Plan （POJ No.2175）

一条街上有N幢大楼。为了防范核战争而建有核防空洞。大楼i的坐标为(X_i, Y_i)，有B_i个人在里面工作。而防空洞j的坐标为(P_j, Q_j)，最多能够容纳C_j个人。大楼i中的人到防空洞j去避难所需的时间为$|X_i-P_j|+|Y_i-Q_j|+1$。为了防止所有人都选择最近的防空洞避难而导致一些防空洞的人数超过设计限制，街道议会指定了一个避难计划。该计划中指定了应该从大楼 i 到防空洞 j 避难的人数 $E_{i,j}$。请判断如果按照该计划避难的话，所有人避难所用时间的总和是不是最小的。如果已经是最小的话，输出"OPTIMAL"，否则输出"SUBOPTIMAL"，并输出一组时间总和更小的避难计划。

限制条件

- $1 \leqslant N \leqslant 100$
- $1 \leqslant M \leqslant 100$
- $-1000 \leqslant X_i, Y_i \leqslant 1000$
- $1 \leqslant B_i \leqslant 1000$
- $-1000 \leqslant P_i, Q_i \leqslant 1000$
- $1 \leqslant C_i \leqslant 1000$
- $0 \leqslant E_{i,j} \leqslant 1000$

样例

输入

```
N = 3
M = 4
(X, Y, B) = {(-3, 3, 5), (-2, -2, 6), (2, 2, 5)}
(P, Q, B) = {(-1, 1, 3), (1, 1, 4), (-2, -2, 7), (0, -1, 3)}
E = {{3, 1, 1, 0}, {0, 0, 6, 0}, {0, 3, 0, 2}}
```

输出

```
SUBOPTIMAL
3 0 1 1
0 0 6 0
0 4 0 1
```

像这样要确定两类物体之间的对应关系，并希望使总花费最小的问题称为指派问题。如果把两类物体当作顶点，并在顶点之间连接权重为对应的花费的边，就转化为了最小权匹配问题。与二分图最大匹配可以用最大流求解类似，二分图最小权匹配可以用最小费用流求解。建图的方法和二分图最大匹配的情况几乎一样。

在代表大楼的顶点集合U和代表防空洞的顶点集合V之外，添加源点s和汇点t。从s向各个大楼$u \in$

U连一条容量为楼内人数、费用为0的边，从各个防空洞 $v \in V$向t连一条容量为防空洞的收容上限、费用为0的边，再从各个大楼$u \in U$向各个防空洞$v \in V$连一条容量为INF、费用为它们之间的距离的边。假设所有大楼里的总人数为F，那么原问题所要求的最小费用就是所建图中流量为F的最小费用流的费用[①]。

事实上，大楼和防空洞之间的边的容量只要取大楼内的人数和防空洞的收容上限中的较小者就足够了。不过为了方便起见，这里直接取了INF。建好图之后，利用最小费用流求出最小花费，再与题中所给方案的所需花费进行比较，就知道题中所给的避难计划是不是最优的了。

```
// 输入
int N, M;
int X[MAX_N], Y[MAX_N], B[MAX_N];
int P[MAX_M], Q[MAX_M], C[MAX_M];
int E[MAX_N][MAX_M];

void solve() {
  // 建图
  // 0~N-1: 大楼
  // N~N+M-1: 防空洞
  int s = N + M, t = s + 1;
  V = t + 1;
  int cost = 0;  // 计算避难计划的总花费
  int F = 0;     // 总人数
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      int c = abs(X[i] - P[j]) + abs(Y[i] - Q[j]) + 1;
      add_edge(i, N + j, INF, c);
      cost += E[i][j] * c;
    }
  }
  for (int i = 0; i < N; i++) {
    add_edge(s, i, B[i], 0);
    F += B[i];
  }
  for (int i = 0; i < M; i++) {
    add_edge(N + i, t, C[i], 0);
  }

  if (min_cost_flow(s, t, F) < cost) {
    // 非最优时
    printf("SUBOPTIMAL\n");
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < M; j++) {
        printf("%d%c", G[N + j][i].cap, j + 1 == M ? '\n' : ' ');
      }
    }
  } else {
```

① F也是最大流的流量，所以此时也称为最小费用最大流。除了前面提到的流量任意等情况外，多数应用场合我们用的都是最小费用最大流。——译者注

```
      // 最优时
      printf("OPTIMAL\n");
    }
  }
```

不过这个题目并不要求最小花费，只要能够判断是不是最小的就足够了，所以可以更为高效地求解。不妨回想一下我们对最小费用流算法正确性的证明。某个流f是同流量中的最小费用流，等价于f的残余网络中没有负圈。因此，我们在指派问题对应的图中，增广所给避难计划所对应的流，然后在残余网络上检查有没有负圈就能够判断解是否是最优的了。

而要判断有向图中有没有负圈，只要用Bellman-Ford算法或是Floyd–Warshall算法就能轻松办到。而如果找到了一个负圈，通过沿着该负圈增广，就能够得到在相同流量下费用更小的流了。[①]

```
Const int MAX_V = MAX_N + MAX_M + 1;

// 输入
int N, M;
int X[MAX_N], Y[MAX_N], B[MAX_N];
int P[MAX_M], Q[MAX_M], C[MAX_M];
int E[MAX_N][MAX_M];

int g[MAX_V][MAX_V];      // 距离矩阵
int prev[MAX_V][MAX_V];   // 最短路中的前驱
bool used[MAX_V]; // 找圈用的标记

void solve() {
  int V = N + M + 1;
  // 计算距离矩阵
  for (int i = 0; i < V; i++) {
    fill(g[i], g[i] + V, INF);
  }
  for (int j = 0; j < M; j++) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      int c = abs(X[i] - P[j]) + abs(Y[i] - Q[j]) + 1;
      g[i][N + j] = c;
      if (E[i][j] > 0) g[N + j][i] = -c;
      sum += E[i][j];
    }
    if (sum > 0) {
      g[N + M][N + j] = 0;
    }
    if (sum < C[j]) {
      g[N + j][N + M] = 0;
    }
  }

  // 用Floyd-Warshall算法查找负圈
```

① 利用这个原理，我们可以先求出任意一个流，然后不断消去负圈而得到最小费用流，这类算法称为消负圈算法。与之相对的，前面所介绍的算法称为连续最短路算法。——译者注

```
for (int i = 0; i < V; i++) {
  for (int j = 0; j < V; j++) {
    prev[i][j] = i;
  }
}
for (int k = 0; k < V; k++) {
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (g[i][j] > g[i][k] + g[k][j]) {
        g[i][j] = g[i][k] + g[k][j];
        prev[i][j] = prev[k][j];
        if (i == j && g[i][i] < 0) {
          fill(used, used + V, false);
          // 找到负圈
          for (int v = i; !used[v]; v = prev[i][v]) {
            used[v] = true;
            if (v != N + M && prev[i][v] != N + M) {
              if (v >= N) {
                E[prev[i][v]][v - N]++;
              } else {
                E[v][prev[i][v] - N]--;
              }
            }
          }
          printf("SUBOPTIMAL\n");
          for (int x = 0; x < N; x++) {
            for (int y = 0; y < M; y++) {
              printf("%d%c", E[x][y], y + 1 == M ? '\n' : ' ');
            }
          }
          return;
        }
      }
    }
  }
}
// 最优时
printf("OPTIMAL\n");
}
```

The Windy's （POJ No.3686）

预定了 N 个玩具，交付给 M 个工厂加工。j 号工厂加工 i 号玩具需要 $Z_{i,j}$ 的时间。每个玩具都应该完全在某一个工厂内加工完。玩具的制作顺序可以是任意的，但每个工厂在完全加工好一个玩具前，都不能处理别的订货单。请问加工完所有玩具的平均时间的最小值。

限制条件

- $1 \leqslant N, M \leqslant 50$
- $1 \leqslant Z_{i,j} \leqslant 100000$

样例 1

输入

```
N = 3
M = 4
Z = {{100, 100, 100, 1}, {99, 99, 99, 1}, {98, 98, 98, 1}}
```

输出

```
2.0（全部交给4号工厂加工）
```

样例 2

输入

```
N = 3
M = 4
Z = {{1, 100, 100, 100}, {99, 1, 99, 99}, {98, 98, 1, 98}}
```

输出

```
1.0（分别交给1、2、3号工厂加工）
```

样例 3

输入

```
N = 3
M = 4
Z = {{1, 100, 100, 100}, {1, 99, 99, 99}, {98, 1, 98, 98}}
```

输出

```
1.333333（1号工厂负责加工1号和2号玩具，2号工厂负责加工3号玩具）
```

因为加工完所有玩具的平均时间就是总时间除以N，所以只要最小化总时间就好了。首先，不妨考虑一下每个工厂只能加工一个玩具的情况。此时，问题就是普通的指派问题，利用最小费用流就能够求解了。而现在的情况是一个工厂能够加工多个玩具。为了分析允许加工多个玩具时，完成的总时间和玩具之间的关系，不妨先考虑只有一个工厂的情形。

假设加工i号玩具所需的时间为Z_i。当我们按照$a_1, a_2, \cdots, a_n$的顺序加工玩具时，显然中途休息是没有好处的，所以总时间的最小值T为

$$T = Z_{a_1} + (Z_{a_1} + Z_{a_2}) + \cdots + (Z_{a_1} + Z_{a_2} + \cdots + Z_{a_n})$$