```
    }
  }

  // 计算u和v的LCA
  int lca(int u, int v) {
    // 让u和v向上走到同一深度
    if (depth[u] > depth[v]) swap(u, v);
    for (int k = 0; k < MAX_LOG_V; k++) {
      if ((depth[v] - depth[u]) >> k & 1) {
        v = parent[k][v];
      }
    }
    if (u == v) return u;
    // 利用二分搜索计算LCA
    for (int k = MAX_LOG_V - 1; k >= 0; k--) {
      if (parent[k][u] != parent[k][v]) {
        u = parent[k][u];
        v = parent[k][v];
      }
    }
    return parent[0][u];
  }
```

像这样，预处理出2^k的表的技巧，在计算LCA之外也很有用，相关的问题也经常出现在程序设计竞赛当中。对此，下一节中还会介绍其他例子。

2. 基于RMQ的算法

对于涉及有根树的问题，将树转为从根DFS标号后得到的序列处理的技巧常常十分有效。对于LCA，利用该技巧也能够高效地计算。首先，按从根DFS访问的顺序得到顶点序列vs[i]和对应的深度depth[i]。对于每个顶点v，记其在vs中首次出现的下标为id[v]。

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vs	1	2	4	2	5	7	5	8	5	2	1	3	6	3	1
depth	0	1	2	1	2	3	2	3	2	1	0	1	2	1	0

i	1	2	3	4	5	6	7	8
id	0	1	11	2	4	12	5	7

样例对应的标号

这些都可以在$O(n)$时间内求得。而LCA(u,v)就是访问u之后到访问v之前所经过顶点中离根最近的那个，假设id[u]≤id[v]，那么有

$$LCA(u,v)=vs[id[u]\leq i\leq id[v]\text{中令}depth(i)\text{最小的}i]$$

而这可以利用RMQ高效地求得。

```
// 输入
vector<int> G[MAX_V];       // 图的邻接表表示
int root;

int vs[MAX_V * 2 - 1];      // DFS访问的顺序
int depth[MAX_V * 2 - 1];   // 节点的深度
int id[MAX_V];              // 各个顶点在vs中首次出现的下标

void dfs(int v, int p, int d, int &k) {
  id[v] = k;
  vs[k] = v;
  depth[k++] = d;
  for (int i = 0; i < G[v].size(); i++) {
    if (G[v][i] != p) {
      dfs(G[v][i], v, d + 1, k);
      vs[k] = v;
      depth[k++] = d;
    }
  }
}

// 预处理
void init(int V) {
  // 预处理出vs、depth和id
  int k = 0;
  dfs(root, -1, 0, k);
  // 预处理出RMQ（返回的不是最小值，而是最小值对应的下标）
  rmq_init(depth, V * 2 - 1);
}

// 计算u和v的LCA
int lca(int u, int v) {
  return vs[query(min(id[u], id[v]), max(id[u], id[v]) + 1)];
}
```

Housewife Wind （POJ No.2763）

××村里有 n 个小屋，小屋之间有双向可达的道路相连，所构成的图是一棵树。通过连接 a_i 号小屋和 b_i 号小屋的道路 i 需要花费 w_i 的时间。你一开始在 s 号小屋。请处理以下 q 个查询。

A：输出从当前位置移动到节点 x 所需的时间。 B：将通过道路 x 所需的时间改为 t。

限制条件

- $1 \leqslant n \leqslant 100000$
- $0 \leqslant q \leqslant 100000$
- $1 \leqslant a_i, b_i \leqslant n$
- $1 \leqslant w_i \leqslant 10000$

样例

输入

```
n = 3
q = 3
s = 1
(a, b, w) = {(1, 2, 1), (2, 3, 2)}
(查询的类型, x(, t)) = {(A, 2), (B, 2, 3), (A, 3)}
```

输出

```
从1移动到2
从2移动到3
```

虽然直接DFS也可以求出树上两点之间的距离，但是这对于每个A类型的查询，都要花费$O(n)$的时间，实在太慢了。必须利用树的特性，得到更为高效的算法。为了高效地处理A类型的查询，可以利用二分搜索版LCA算法中介绍的技巧，记录下从每个顶点向上走2^k步的总长度。这样一来，在$O(\log n)$地计算LCA的同时，也可以同样$O(\log n)$地求出到LCA的距离，因此处理A类型查询的复杂度为$O(\log n)$。但是，这个方法对于B类型的查询却无法高效地处理。

因此，让我们先考虑一下图是链状时这一简单的情况。假设i和$i+1$两点之间的边的长度为w_i，则两点u和v($u<v$)之间的距离为

$$\sum_{i=u}^{v-1} w_i$$

只要用BIT，不论是A类型的查询还是B类型的查询，都能够在$O(\log n)$时间内处理。

在回过头来考虑树的情况。因为树中连接两点的路径是唯一的，如果我们对顶点进行合理排列的话，能否像链状时那样，进行类似的处理呢？考虑利用RMQ计算LCA时所用的，按DFS访问的顺序排列的顶点序列。这样，u和v之间的路径，就是在序列中u和v之间的所有边减去往返重复的部分得到的结果。

将树转为序列

于是，只要令边的权重沿叶子方向为正，沿根方向为负，那么往返重复的部分就自然抵消了，于是有

$$(u,v\text{之间的距离})=(\text{从}LCA(u,v)\text{到}u\text{的边的权重和})+(\text{从}LCA(u,v)\text{到}v\text{的边的权重和})$$

同链状的情况一样，利用BIT的话，计算权重和和更新边权都可以在$O(\log n)$时间内办到，而LCA也能够在$O(\log n)$时间内求得。

```
struct edge { int id, to, cost; };

int n, q, s;
int a[MAX_V - 1], b[MAX_V - 1], w[MAX_V - 1];
int type[MAX_Q];  // 0: A类型, 1: B类型
int x[MAX_Q], t[MAX_Q];

vector<edge> G[MAX_V]; // 图的邻接表表示
int root;

int vs[MAX_V * 2 - 1];      // DFS访问的顺序
int depth[MAX_V * 2 - 1];   // 节点的深度
int id[MAX_V];              // 各个顶点在vs中首次出现的下标
int es[(MAX_V - 1) * 2];    // 边的下标（i*2+(叶子方向:0,根方向:1))

void dfs(int v, int p, int d, int &k) {
  id[v] = k;
  vs[k] = v;
  depth[k++] = d;
  for (int i = 0; i < G[v].size(); i++) {
    edge &e = G[v][i];
    if (e.to != p) {
      add(k, e.cost);
      es[e.id * 2] = k;
      dfs(e.to, v, d + 1, k);
      vs[k] = v;
      depth[k++] = d;
      add(k, -e.cost);
      es[e.id * 2 + 1] = k;
    }
  }
}

int stack_v[MAX_V + 10];
int stack_i[MAX_V + 10];

// 预处理
void init(int V) {
  // 初始化BIT
  bit_n = (V - 1) * 2;
  // 预处理出vs、depth、id和es
  int k = 0;
  dfs(root, -1, 0, k);
  // 预处理出RMQ（返回的不是最小值，而是最小值对应的下标）
```

```
  rmq_init(depth, V * 2 - 1);
}

// 计算u和v的LCA
int lca(int u, int v) {
  return vs[query(min(id[u], id[v]), max(id[u], id[v]) + 1)];
}

void solve() {
  // 预处理
  root = n / 2;  // 不论以哪个节点为根都没有问题
  for (int i = 0; i < n - 1; i++) {
    G[a[i] - 1].push_back((edge){i, b[i] - 1, w[i]});
    G[b[i] - 1].push_back((edge){i, a[i] - 1, w[i]});
  }
  init(n);
  // 处理查询
  int v = s - 1;  // 当前位置
  for (int i = 0; i < q; i++) {
    if (type[i] == 0) {
      // 从当前位置移动到x[i]
      int u = x[i] - 1;
      int p = lca(v, u);
      // 利用BIT计算p到v和p到u的费用之和，即区间(id[p],id[v]]和(id[p],id[u]]的权重和
      printf("%d\n", sum(id[v]) + sum(id[u]) - sum(id[p]) * 2);
      v = u;
    } else {
      // 将通过道路x[i]的权重改为t[i]。
      int k = x[i] - 1;
      add(es[k * 2], t[i] - w[k]);
      add(es[k * 2 + 1], w[k] - t[i]);
      w[k] = t[i];
    }
  }
}
```

4.4 常用技巧精选（二）

➠通过活用栈和队列等简单的数据结构，可以巧妙地降低一些算法的复杂度。在这一节中，将会给大家介绍这些技巧。

4.4.1 栈的运用

Largest Rectangle in a Histogram （POJ No.2559）

柱状图是由一些宽度相等的长方形下端对齐后横向排列得到的图形。现在有由 n 个宽度为 1，高度分别为 $h_1,h_2,\cdots,h_n$ 的长方形从左到右依次排列组成的柱状图。问里面包含的长方形的最大面积是多少。

限制条件

- $1 \leqslant n \leqslant 100000$
- $0 \leqslant h_i \leqslant 10^9$

样例

输入

```
n = 7
h = {2, 1, 4, 5, 1, 3, 3}
```

输出

```
8
```

样例对应的答案

如果确定了长方形的左端点L和右端点R，那么最大可能的高度就是$\min\{h_i|L\leqslant i<R\}$。这样我们就得到了一个$O(n^3)$的算法。如果对计算区间最小值进行一些优化，那么可以把复杂度降为$O(n^2)$。但即使这样，仍然无法在规定时间内求出答案。那么我们应该怎么做才能更加高效地求解呢？设面积最大的长方形的左端是L，右端是R，高度是H。如果$h_{L-1}\geqslant H$，那么左端点就可以更新为$L-1$，从而可以得到更大的长方形。这与假设矛盾，因此$h_{L-1}<H$。同理可得$h_R<H$，并且高度$H=\min\{h_i|L\leqslant i<R\}$。因此，我们固定可以给出这样的$H$的$i$并进行分析。此时，$L$是满足$h_{j-1}<h_i$的最大的$j(\leqslant i)$。$R$是满足$h_j<h_i$的最小的$j(>i)$。我们把这两个值分别表示为$L[i]$和$R[i]$。则

$L[i]=(j\leqslant i$并且$h_{j-1}<h_i$的最大的$j)$
$R[i]=(j>i$并且$h_j>h_i$的最小的$j)$

如果能求出$L[i]$和$R[i]$，那么最大的面积就是$\max\{h_i\times(R[i]-L[i])|0\leqslant i<n\}$。$L$和$R$可以使用栈非常高效地求解。我们先考虑计算$L$的情况。首先定义一个栈，并且将它初始化为空。然后不断增加i的值，并维护这个栈使它按照下面的顺序存储用于推算后面的L值的元素。

设在栈里的元素从上到下的值为x_i，则$x_i>x_{i+1}$且$h_{x_i}>h_{x_{i+1}}$

在计算$L[i]$时，首先，当栈顶的元素j满足$h_j\geqslant h_i$，则不断取出栈顶元素。若栈为空，则$L[i]=0$，若$h_j<h_i$，则$L[i]=j+1$。然后把i压入栈中。例如，对于样例输入，算法按照如下的步骤执行。

```
i = 0
L[0] = 0
压入0 → {0}
i = 1
弹出0 → {} (h[0]>=h[1])
L[1] = 0
压入1 → {1}
i = 2
L[2] = 2
压入2 → {1, 2}
i = 3
L[3] = 3
压入3 → {1, 2, 3}
i = 4
弹出3 → {1, 2} (h[3]>=h[4])
弹出2 → {1} (h[2]>=h[4])
弹出1 → {} (h[1]>=h[4])
L[4] = 0
压入4 → {4}
...
```

由于栈的压入和弹出操作都是$O(n)$次，因此这个算法的复杂度为$O(n)$。对于R也可以用同样的方法计算。

i	0	1	2	3	4	5	6
h	2	1	4	5	1	3	3
L	0	0	2	3	0	5	5
R	1	7	4	4	7	7	7
h(R-L)	2	7	8	5	7	6	6

对于样例的L和R

```
// 输入
int n;
int h[MAX_N];

int L[MAX_N], R[MAX_N];
int st[MAX_N];  // 栈

void solve() {
  // 计算L
  int t = 0;  // 栈的大小
  for (int i = 0; i < n; i++) {
    while (t > 0 && h[st[t - 1]] >= h[i]) t--;
    L[i] = t == 0 ? 0 : (st[t - 1] + 1);
    st[t++] = i;
  }

  // 计算R
  t = 0;
  for (int i = n - 1; i >= 0; i--) {
    while (t > 0 && h[st[t - 1]] >= h[i]) t--;
    R[i] = t == 0 ? n : st[t - 1];
    st[t++] = i;
  }

  long long res = 0;  // 注意防止溢出
  for (int i = 0; i < n; i++) {
    res = max(res, (long long)h[i] * (R[i] - L[i]));
  }
  printf("%lld\n", res);
}
```

4.4.2 双端队列的运用

滑动最小值

给定一个长度为 n 的数列 $a_0,a_1,\cdots,a_{n-1}$ 和一个整数 k。求数列 $b_i=\min\{a_i,a_{i+1},\cdots,a_{i+k-1}\}$ $(i=0,1,\cdots,n-k)$。

限制条件

- $1\leqslant k\leqslant n\leqslant 10^6$
- $0\leqslant a_i\leqslant 10^9$

样例

输入

```
n = 5
k = 3
a = {1, 3, 5, 4, 2}
```

输出

```
b = {1, 3, 2}
```

这个问题可以使用RMQ在$O(n \log n)$复杂度内解决。但是，如果利用要求的范围大小总是一定的这一条件，则可以使用双端队列（Deque：可以在头部和末尾插入和删除元素的数据结构）在$O(n)$时间内解决这个问题。

最开始时双端队列为空，然后不断维护双端队列使它按照下面的顺序，存储用于计算后面的最小值的a的元素的下标。

设双端队列从头部开始的元素的值为x_i，则$x_i<x_i+1$且$a_{x_i} < a_{x_{i+1}}$。

首先，为了计算b_0，把0到$k-1$依次加入队列。在加入i时，当双端队列的末尾的值j满足$a_j \geq a_i$，则不断取出。直到双端队列为空或者$a_j<a_i$之后再在末尾加入i。

等到$k-1$都加入双端队列了之后，查看双端队列头部的值j，那么$b_0=a_j$。如果$j=0$，由于在之后的计算中都不会再用到了，因此从双端队列的头部删去。

接下来，为了计算b_i，需要在双端队列的末尾加入k。不断加入元素，就可以算出后面的b_i的值。由于双端队列的加入和删除都进行了$O(n)$次，因此整个算法的复杂度是$O(n)$。[①]

例如，对于样例输入，算法按照如下的步骤执行。

```
加入0 → {0}
加入1 → {0, 1}
加入2 → {0, 1, 2}
b_0 = a_0 = 1
删除0 → {1, 2}
加入3 → {1, 3} (a_2>=a_3，因此删除2)
b_1 = a_1 = 3
删除1 → {3}
加入4 → {4} （a_3>=a_4，因此删除3）
b_2 = a_4 = 2
```

① 由于双端队列中的元素始终保持单调性，因此这个数据结构也被称作单调队列。——译者注

```
// 输入
int n, k;
int a[MAX_N];

int b[MAX_N];
int deq[MAX_N];  // 双端队列

void solve() {
  int s = 0, t = 0;  // 双端队列的头部和末尾

  for (int i = 0; i < n; i++) {
    // 在双端队列的末尾加入i
    while (s < t && a[deq[t - 1]] >= a[i]) t--;
    deq[t++] = i;

    if (i - k + 1 >= 0) {
      b[i - k + 1] = a[deq[s]];

      if (deq[s] == i - k + 1) {
        // 从双端队列的头部删除元素
        s++;
      }
    }
  }

  for (int i = 0; i <= n - k; i++) {
    printf("%d%c", b[i], i == n - k ? '\n' : ' ');
  }
}
```

多重背包问题

有 n 种物品，它们的重量和价值分别是 w_i 和 v_i。现在要从中选出一些物品使得总重量不超过 W，并且价值的和最大。不过第 i 种物品最多可以选 m_i 个。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i, v_i \leqslant 100$
- $1 \leqslant m_i \leqslant 10000$
- $1 \leqslant W \leqslant 10000$

样例

输入

```
n = 3
(w, v, m) = {(3, 2, 5), (2, 4, 1), (4, 3, 3)}
W = 12
```

输出

```
11（第一种物品2个，第二种物品1个，第三种物品1个）
```

这是一道有个数限制的背包问题。对于每个物品至多选一个或者可以选任意个的问题我们已经可以在$O(nW)$时间内求解了。如果使用同样的方法解答本题，则状态转移方程为

$$dp[i][j]:=\text{到第}i\text{个物品为止总重量不超过}j\text{的所有选法中最大可能的价值}$$
$$dp[i+1][j]=\max\{dp[i][j-k\times w[i]]+k\times v[i]|0\leqslant k\leqslant m_i\text{且}j-k\times w[i]\geqslant 0\}$$

如果使用这个转移方程，复杂度就是$O(nmW)$，无法在规定时间内出解。让我们注意观察转移方程中求最大值的部分。这个式子中若$j \bmod w[i]$的值不同则之间是互相独立的。我们首先考虑一下$j \bmod w[i]=0$的情况。我们定义

$$a[j]=dp[i][j\times w[i]]$$

则转移方程可以写成

$$dp[i+1][(j+k)\times w[i]]=\max\{a[j]+k\times v[i],a[j+1]+(k-1)\times v[i],\cdots,a[j']+(j+k-j')\times v[i],\cdots,a[j+k]\}$$

但是这样还是无法方便地计算，因此再进行如下变形

$$b[j]=a[j]-j\times v[i]$$
$$dp[i+1][(j+k)\times w[i]]=\max\{b[j],b[j+1],\cdots,b[j+k]\}+(j+k)\times v[i]$$

这样变形之后，在求最大值时可以使用之前提到的滑动最小值的方法求解，所以复杂度就降到了$O(nW)$。

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

int dp[MAX_W + 1];  // DP数组（循环使用）
int deq[MAX_W + 1];   // 双端队列（保存数组下标）
int deqv[MAX_W + 1];  // 双端队列（保存值）

void solve() {
  for (int i = 0; i < n; i++) {
    for (int a = 0; a < w[i]; a++) {
      int s = 0, t = 0;  // 双端队列的头部和末尾
      for (int j = 0; j * w[i] + a <= W; j++) {
        // 向双端队列的末尾加入j
        int val = dp[j * w[i] + a] - j * v[i];
        while (s < t && deqv[t - 1] <= val) t--;
        deq[t] = j;
        deqv[t++] = val;
        // 从双端队列的头部取出t
        dp[j * w[i] + a] = deqv[s] + j * v[i];
```

```
        if (deq[s] == j - m[i]) {
          s++;
        }
      }
    }
  }
  printf("%d\n", dp[W]);
}
```

虽然复杂度上差了一些，不过也可以使用下面的方法求解。把m_i分解为如下形式

$$m_i=1+2+4+\cdots+2^k+a(0\leqslant a<2^{k+1})$$

由于$1,2,\cdots,2^k$的组合可以表示出$0\sim 2^{k+1}-1$的所有整数，因此$1,2,\cdots,2^k,a$可以表示出$0\sim m_i$的所有整数。因此，我们把m_i个重量和价值分别为w_i和v_i的物品，看成重量和价值分别为$w_i\times x,v_i\times x(x=1,2,\cdots,2^k,a)$的$k+2$个物品。这样，物品的总个数就变为$O(n\log m)$个，使用一般的01背包DP可以在$O(n\,W\log m)$时间内求出答案。

```
int dp[MAX_W + 1];  // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    int num = m[i];
    for (int k = 1; num > 0; k <<= 1) {
      int mul = min(k, num);
      for (int j = W; j >= w[i] * mul; j--) {
        dp[j] = max(dp[j], dp[j - w[i] * mul] + v[i] * mul);
      }
      num -= mul;
    }
  }
  printf("%d\n", dp[W]);
}
```

K-Anonymous Sequence （POJ No.3709）

给定一个长度为 n 的非严格单调递增数列 $a_0,a_1,\cdots,a_{n-1}$。每一次操作可以使数列中的任何一项的值减小 1。现在要使数列中的每一项都满足其他项中至少有 $k-1$ 项和它相等。求最少要对这个数列操作的次数。

限制条件

- $2\leqslant k\leqslant n\leqslant 500000$
- $0\leqslant a_i\leqslant 500000$

样例

输入

```
n = 7
k = 3
a = {2, 2, 3, 4, 4, 5, 5}
```

输出

```
3(2, 2, 2, 4, 4, 4, 4)
```

由于a_0是数列中最小的值，很显然没有必要把任何一个数减到a_0以下，所以就有至少$k-1$个其他的数需要减少至和a_0相等。又因为减小大的值而保留小的值不会使结果更优，所以可以从小到大选择需要减少至a_0的项。对于剩下的部分也有同样的结论成立。这样，我们就有了下面的DP方程。

$dp[i]$:=在只考虑前i项的情况下，满足题目条件的最少的操作次数（不可能的情况为INF）

$dp[0]=0$

$dp[i]=min\{dp[j]+(a_j+1-a_j)+\cdots+(a_{i-1}-a_j)|0\leqslant j\leqslant i-k\}$

最终答案为$dp[n]$

直接计算的复杂度为$O(n^3)$。不过转移方程中涉及到了部分和的计算，可以通过预处理出这些值对算法进行改进。

$S[i]=a_0+\cdots+a_{i-1}$

$dp[i]=min\{dp[j]+S[i]-S[j]-a_j\times(i-j)|0\leqslant j\leqslant i-k\}$

这样复杂度就降为了$O(n^2)$，不过由于题目中n很大，仍然无法满足要求。由于只是按顺序进行转移是无法在规定时间内出解的，所以有必要挖掘转移方程中的一些特殊性质。考虑到对于某个i，在j从0变化到$i-k$的过程中$S[i]$是一个定值，因此先把它提到外面。

$dp[i]=S[i]+min\{dp[j]-S[j]-a_j\times(i-j)|0\leqslant j\leqslant i-k\}$

这样变形之后，min里面的项就是关于i的线性函数了。

$f_j(x)=-a_j\times x+dp[j]-S[j]+a_j\times j$

$dp[i]=S[i]+min\{f_j(i)|0\leqslant j\leqslant i-k\}$

也就是说，计算$dp[i]$就相当于从$i-k+1$条直线中寻找$x=i$的最小值。而$dp[i]$和$dp[i+1]$的区别仅在于$dp[i+1]$需要多考虑一条直线，并且所求的x坐标增加了1。因此，可以得到如下算法。

1. 使用某种数据结构维护所有可能成为最小值的直线的集合，并以成为最小值的顺序排列保存，记为L。
2. 计算$dp[i]$只需要取L的头部的直线进行计算就可以了。

3. i每增加1，如果L的头部的直线变得不是最小了，则删除之。
4. 增加一条直线时对L进行更新。

下侧的包络线的变化

如果使用二叉搜索树来维护L，就可以得到一个$O(n\log n)$的算法。更进一步，在本题中由于f_j的斜率（$=-a_j$）具有非严格单调递减的性质，因而4的更新可以从L的末尾进行更新。这样，就可以使用双端队列代替二叉搜索树，从而更加高效地进行求解。

1. 使用双端队列维护所有可能成为最小值的直线的集合，并以成为最小值的顺序排列保存，记为L。
2. 计算$dp[i]$，只需要取L的头部的直线进行计算就可以了。
3. i每增加1，当L的头部的直线变得不是最小时，则删除之。
4. 增加一条直线时，先删除所有在L的末尾中已经不可能成为最小值的直线，然后加入L的末尾。

由于共对双端队列进行了最多n次的加入和删除操作，因此这个算法的复杂度是$O(n)$。此外关于4的判断，经过推导之后可以按如下方式进行。

假设有3条直线按照斜率排列有

$$f_1(x)=a_1x+b_1$$
$$f_2(x)=a_2x+b_2$$
$$f_3(x)=a_3x+b_3$$
$$a_1\geqslant a_2\geqslant a_3$$

则

$$f_2\text{不可能成为最小值对应的直线}\Leftrightarrow(a_2-a_1)\times(b_3-b_2)\geqslant(b_2-b_1)\times(a_3-a_2)$$

不可能成为最小值的直线

```
typedef long long ll;

// 输入
int n, k;
ll a[MAX_N];

ll dp[MAX_N + 1];  // DP数组
ll S[MAX_N + 1];   // a的和
int deq[MAX_N];    // 双端队列

// 直线f_j在x位置的值
ll f(int j, int x) {
  return -a[j] * x + dp[j] - S[j] + a[j] * j;
}

// 判断f2是否有可能成为最小值
bool check(int f1, int f2, int f3) {
  ll a1 = -a[f1], b1 = dp[f1] - S[f1] + a[f1] * f1;
  ll a2 = -a[f2], b2 = dp[f2] - S[f2] + a[f2] * f2;
  ll a3 = -a[f3], b3 = dp[f3] - S[f3] + a[f3] * f3;
  return (a2 - a1) * (b3 - b2) >= (b2 - b1) * (a3 - a2);
}

void solve() {
  // 和的计算
  for (int i = 0; i < n; i++) {
    S[i + 1] = S[i] + a[i];
  }

  // 双端队列的初始化
  int s = 0, t = 1;
  deq[0] = 0;

  dp[0] = 0;

  for (int i = k; i <= n; i++) {
    if (i - k >= k) {
      // 从末尾删除不再可能成为最小值的直线
```

```
        while (s + 1 < t && check(deq[t - 2], deq[t - 1], i - k)) t--;

        // 往双端队列中加入i-k
        deq[t++] = i - k;
      }

      // 若头部的值不是最小值了则删去
      while (s + 1 < t && f(deq[s], i) >= f(deq[s + 1], i)) s++;

      dp[i] = S[i] + f(deq[s], i);
    }

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

4.4.3 倍增法

观看计划

有一个喜欢动画的少年，他希望每周都能收看尽可能多的动画。每一部动画都在每周固定的时间段播出。不过，由于他不喜欢录下电视节目留到以后观看，因此只能在播出时观看。此外，他每周看的动画都是固定的，并且他不可能同时观看播出时间有重叠的两部动画。在他的国家，一周共被划分成了 M 个单位的时间。现在假设一共有 N 部动画分别在每周的 s_i 时刻开始播放并在 t_i 时刻播放结束，问每周他最多能看多少部动画。

限制条件

- $1 \leqslant N \leqslant 10^5$
- $2 \leqslant M \leqslant 10^6$
- $0 \leqslant s_i, t_i < M (s_i \neq t_i)$
- $s_i > t_i$ 表示播出时间跨越了每周的最后一个时刻

样例 1

输入

```
N = 3
M = 10
(s, t) = {(0, 3), (3, 7), (7, 0)}
```

输出

```
3（可以看所有的动画）
```

样例 2

输入

```
N = 3
M = 10
(s, t) = {(0, 5), (2, 7), (6, 9)}
```

输出

```
2（可以观看第一部和第三部动画）
```

本题可以看成在圆周上有N个区间，要从中选出尽可能多互不相交的区间的问题。和这个问题类似的有区间调度问题（请参照2.2.2）。本题仅仅是把那个问题的时间全集首尾相连形成一个圆周而已。区间调度问题可以根据结束时间排序之后使用贪心法求解。本题是否也可以用同样的方法求解呢？首先我们先确定一个要选择的区间。这样，和这个区间不相交的区间就不存在圆周，从而变成了简单的区间调度问题。因此，我们可以得到这样一个$O(N^2)$的算法。

```
// 输入
int N, M;
int s[MAX_N], t[MAX_N];

pair<int, int> ps[MAX_N * 2];  // 为了按照结束时间排序而使用的数对的数组

void solve() {
  int res = 0;

  // 为了处理方便而把原来的数据复制了一份存在后面
  for (int i = 0; i < N; i++) {
    if (t[i] < s[i]) t[i] += M;
    s[N + i] = s[i] + M;
    t[N + i] = t[i] + M;
  }

  // 按照结束时间排序
  for (int i = 0; i < N * 2; i++) {
    ps[i] = make_pair(t[i], s[i]);
  }
  sort(ps, ps + N * 2);

  // 确定一个最开始选择的区间
  for (int i = 0; i < N; i++) {
    // 剩下的部分使用贪心法求解
    int tmp = 0, last = 0;
    for (int j = i; ps[j].first <= ps[i].second + M; j++) {
      if (last <= ps[j].second) {
        tmp++;
        last = ps[j].first;
      }
    }
```

```
    res = max(res, tmp);
  }

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

不过$O(N^2)$是不够的，我们应该如何改进这个算法呢？因为确定第一个区间之后使用的贪心算法存在大量的重复计算，所以在这个部分有改良的余地。在贪心算法中，对于某个区间i，我们选择满足$t[i] \leqslant s[j]$的所有j中$t[j]$最小的那个区间来作为i的下一个区间。这样的i和j的对应关系和最开始所选的区间无关。因此，记录下这些对应关系就可以节省不少计算量。这些对应关系的计算可以通过对区间的端点排序之后$O(N \log N)$求得。

```
// 输入
int N, M;
int s[MAX_N * 2], t[MAX_N * 2];

pair<int, int> ps[MAX_N * 4];   // 为了按照结束时间排序而使用的数对的数组
int next[MAX_N * 2];            // 用来存放每个区间的下一个区间的数组

void solve() {
  int res = 0;

  // 为了处理方便而把原来的数据复制了一份存在后面
  for (int i = 0; i < N; i++) {
    if (t[i] < s[i]) t[i] += M;
    s[N + i] = s[i] + M;
    t[N + i] = t[i] + M;
  }

  // 对区间的端点排序
  for (int i = 0; i < N * 2; i++) {
    ps[i] = make_pair(t[i], i);
    ps[N * 2 + i] = make_pair(s[i], N * 2 + i);
  }
  sort(ps, ps + N * 4);

  // 计算next
  int last = -1;
  for (int i = N * 4 - 1; i >= 0; i--) {
    int id = ps[i].second;
    if (id < N * 2) {
      // 区间的末尾
      next[id] = last;
    } else {
      // 区间的开始
      id -= N * 2;
      if (last < 0 || t[last] > t[id]) {
        last = id;
      }
    }
  }
```

```
    // 确定一个最开始选择区间
    for (int i = 0; i < N; i++) {
      // 剩下的部分使用贪心法求解
      int tmp = 0;
      for (int j = i; t[j] <= s[i] + M; j = next[j]) {
        tmp++;
      }
      res = max(res, tmp);
    }

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

但是，即使在贪心法中利用了这样的对应关系，仍然无法降低复杂度。虽然能够节省求解下一个区间的计算，但是仍然需要对要选中的区间遍历一遍，最坏情况下共有$O(N)$个区间。那么怎样才能降低复杂度呢？

现在我们对于某一个区间i，已经算出了i的下一个应该使用的区间是next[i]。利用这个值，我们可以算出下一个区间的再下一个区间是next2[i]=next[next[i]]。然后可以利用2个之后的区间next2算出4个之后的区间next4=next2[next2[i]]。不断重复类似的计算，就可以在$O(N k)$时间内预处理出2^k个之后的区间next[k][i]。如果得到了对于k=floor(log N)的预处理结果，那么就可以把贪心法中一个区间一个区间地遍历，改为使用二分搜索的方式。对于一个初始选取的区间计算的复杂度是$O(\log N)$。因此整个算法的复杂度就是$O(N \log N)$。

```
  // 输入
  int N, M;
  int s[MAX_N * 2], t[MAX_N * 2];

  pair<int, int> ps[MAX_N * 4];     // 为了按照结束时间排序而使用的数对的数组
  int next[MAX_LOG_N][MAX_N * 2];   // 用来存放每个区间的下一个区间的数组

  void solve() {
    int res = 0;

    // 为了处理方便而把原来的数据复制了一份存在后面
    for (int i = 0; i < N; i++) {
      if (t[i] < s[i]) t[i] += M;
      s[N + i] = s[i] + M;
      t[N + i] = t[i] + M;
    }

    // 对区间的端点排序
    for (int i = 0; i < N * 2; i++) {
      ps[i] = make_pair(t[i], i);
      ps[N * 2 + i] = make_pair(s[i], N * 2 + i);
    }
    sort(ps, ps + N * 4);
```

```
// 计算next[0]
int last = -1;
for (int i = N * 4 - 1; i >= 0; i--) {
  int id = ps[i].second;
  if (id < N * 2) {
    // 区间的末尾
    next[0][id] = last;
  } else {
    // 区间的开始
    id -= N * 2;
    if (last < 0 || t[last] > t[id]) {
      last = id;
    }
  }
}

// 计算next
for (int k = 0; k + 1 < MAX_LOG_N; k++) {
  for (int i = 0; i < N * 2; i++) {
    if (next[k][i] < 0) next[k + 1][i] = -1;
    else next[k + 1][i] = next[k][next[k][i]];
  }
}

// 确定一个最开始选择的区间
for (int i = 0; i < N; i++) {
  // 进行二分搜索
  int tmp = 0, j = i;
  for (int k = MAX_LOG_N - 1; k >= 0; k--) {
    int j2 = next[k][j];
    if (j2 >= 0 && t[j2] <= s[i] + M) {
      j = j2;
      tmp |= 1 << k;
    }
  }
  res = max(res, tmp + 1);
}

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

实际上，针对本题还有一种除去排序只需要花费$O(N)$时间的算法。有兴趣的读者可以试着思考一下。

4.5 开动脑筋智慧搜索

➠一旦搜索空间变得比较大，2.1节中介绍的穷竭搜索算法就显得不够高效了。在本节中，将给大家介绍剪枝和A*等在这种情况下对搜索进行优化的方法。

4.5.1　剪枝

本节要介绍的剪枝和A*等方法，通常比较难估算其复杂度。与其他算法不同，很难知道这类方法能对搜索速度有多大的提高，建议通过对各类方法的实际测试，边比较边学习。比较时，不光要比较所用时间，还应该统计搜索过程中的状态数，以便得到更清晰的结果。

在比赛中尝试各种方法是不切实际的，不妨先尝试某种看似可行的方法，如果还不够高效的话，再考虑进一步优化。

1. 调整搜索的顺序

数独（POJ 2676，2918，3074，3076）

给定一个由 3×3 的方块分割而成的 9×9 的格子。其中一些格子中填有 1~9 的数字，其余格子则是空白的。请在空白的格子中填入 1~9 的数字，使得在每行、每列和每个 3×3 的方块中，1~9 的每个数字都恰好出现一次。如果解不唯一，输出任意一组即可。

						5	2	
	8		4					
	3				9			
5		1				6		
2			7					
			3					
6				1				
						7		4
							3	

数独的例子

样例

输入

```
(0表示空白格子)
000000520
080400000
030009000
501000600
200700000
000300000
600010000
000000704
000000030
```

输出

```
416837529
982465371
735129468
571298643
293746185
864351297
647913852
359682714
128574936
```

POJ中有2676、2918、3074、3076四道数独问题。其难度大致是2676=2918<3074<3076。其中，3076是格子更大一圈的16×16的数独问题。

首先考虑从左上角的空白格子开始填数字的深度优先搜索。所填的数字应该是在所在行、列和方块中都没有填过的数字。只要采取这一显而易见的的剪枝，就能够通过POJ 2676和2918了。

但这个算法却无法通过POJ 3074，对于像前面的样例这样空白格子比较多的情况，该方法就行不通，需要进一步优化。

考虑处理某一行时，对于某个还没用过的数字，如果该行只有一个可行的空白格子，就只能将该数字填入该格子中。对于列和方块也一样。反之，如果某个格子可填的数字只有一个，也只能将该数字填入该格子。这样，我们优先处理数字或格子唯一确定的情况。此外，如果搜索过程中发现没有可选的数字或格子这样矛盾的情况，则提前停止搜索。这样优化之后，POJ 3074也能顺利通过了。

但是，还留有16×16的数独问题运行会超过时间限制。当没有唯一确定的数字或格子时，现在的搜索又会回到原来从左上的空白格子开始填数字的方法。而人们在求解数独问题时，是不会特地这样做的，通常会先处理选择少的格子。

例如，假设有一个只有两个候选数字的格子，如果选择其中一个产生了矛盾，那么就可以确定应该选择另一个。而对于有五个候选数字的格子，即使其中一个出现了矛盾，依然还有四个候选数

字需要尝试。也就是说，比起从左上角开始填数字，优先选择候选数字少的格子填数字要更加高效。这样16×16的数独问题也能够解决了。

这样，通过调整搜索的顺序能够大大优化搜索的效率。这里，我们选择了从分支少的部分开始搜索的策略。此外，也可以从一些影响大的部分开始搜索，例如通过确定一个部分，顺带确定尽可能多的其他部分。

2. 没有更优解则剪枝

Square Destroyer （PKU 1084）

有一个由火柴棒作为边组成的 $N \times N$ 的格子。按照下图，给火柴棒编号。将移除某些火柴棒后的状态作为初始状态，需要再移除一些火柴棒，以保证图中一个正方形也没有。请求出所需移除火柴棒的最少根数。

限制条件

- $1 \leqslant N \leqslant 5$

N=3时的网格

样例 1

输入

```
N = 2
没有已经移除的火柴棒
```

输出

```
3
```

样例 2

输入

```
N = 3
火柴棒12、17、23已经移除
```

输出

```
3（再移除火柴棒6、8、19就能够破坏所有正方形）
```

先试一下从1号火柴棒开始依次选择移除或不移除的穷竭搜索。如果不做任何优化，则总的状态数为$O(2^{2N(N+1)})$，太大了。接下来考虑该如何剪枝。

显而易见的剪枝有:如果移除当前火柴棒也不会破坏任何正方形的话则不除去，以及将待选择的所有火柴棒都移除也还有正方形留下的话则不继续搜索等。将这些剪枝实现之后，就可以通过$N \leqslant 3$的数据了。

```
// 下面是由输入处理后得到的数据
// M和S分别是初始状态中剩余的火柴棒数和正方形数
int M, S;
// m[i][j] == true ⇔ 火柴棒i属于正方形j
bool m[MAX_MATCH][MAX_SQUARE];
// mmax[i] = 正方形i中火柴棒的最大编号
int mmax[MAX_SQUARE];

// p是当前考察的火柴棒的编号，num是至今已经移除的火柴棒的根数
// state[i] == true ⇔ 正方形i尚未破坏
int dfs(int p, int num, vector<bool> state) {
  // 如果检查完了所有火柴棒，也就破坏了所有的正方形
  if (p == M) return num;

  // 如果一定要移除火柴棒p，则use == true
  // 如果一定不除去火柴棒p，则notuse == true
  bool use = false, notuse = true;
  for (int i = 0; i < S; i++) {
    // 火柴棒p会破坏正方形i，所以可以移除
    if (state[i] && m[p][i]) notuse = false;

    // 只剩火柴棒p能破坏正方形i了，所以必须移除
    if (state[i] && mmax[i] == p) use = true;
  }

  int res = INF;
  // 不移除火柴棒p的分支
  if (!use) res = min(res, dfs(p + 1, num, state));

  // 移除火柴棒p的分支
```

```
    for (int i = 0; i < S; i++) {
      if (m[p][i]) state[i] = false;
    }
    if (!notuse) res = min(res, dfs(p + 1, num + 1, state));
    return res;
  }

  void solve() {
    vector<bool> state(S, true);
    printf("%d\n", dfs(0, 0, state));
  }
```

此外，当发现无法更新最优解时，即确定继续搜索所得到的解都比当前已知的最优解更差的话，则没有继续搜索下去的必要了。因为这个分支中的解一定不会比当前已经除去的火柴棒的根数小，如果已知的最优解不比这个大的话，就可以不再继续搜索下去了。下面代码中的num >= min_res做的就是这个。

```
  // 已经找到的最优解
  int min_res;
  int dfs(int p, int num, vector<bool> state) {
    // 如果检查完了所有的火柴棒，也就破坏了所有的正方形
    if (p == M) return min_res = num;

    // 如果比已知的最优解要差，则不继续搜索
    if (num >= min_res) return INF;

    // 以下省略
```

如果我们能够更早一点发现得不到更优的解，则可以让剪枝更为有效。例如，接下来所能找到的解总是没有(当前已经除去的火柴棒数)+(剩余的正方形数)/(一根火柴所能破坏的最多正方形数)小，这就是更好的剪枝用下界。

作为一个更好的下界，可以考虑没有公共边的正方形的最大集合的基数。由于集合中的任意两个正方形都没有公共边，所以一根火柴至多只能破坏其中一个正方形，于是可以作为一个解的下界。

没有公共边的正方形的例子

求没有公共边的正方形的最大集合属于最大独立集问题，这本身就是一个非常困难的问题。不过，我们可以用贪心的近似算法得到最大集合的基数的下界（也就是原问题的下界的下界）。

下面的代码使用了如下贪心算法。

- 记所求的集合为X，初始令X=空集。
- 把所有正方形按其所含的火柴棒数排序。
- 按顺序处理正方形，如果它与X中已有的正方形都没有公共边，则将它加到X中。

计算这个下界的函数记为hstar，将之前的num >= min_res改为num + hstar(p, state) >= min_res就好了。

```
// 考虑p以后的火柴时解的下界
int hstar(int p, vector<bool> state) {
  vector<pair<int, int> > ps;
  for (int i = 0; i < S; i++) {
    if (state[i]) {
      // 统计剩余正方形所包含的火柴棒数
      int num = 0;
      for (int j = p; j < M; j++) {
        if (m[j][i]) num++;
      }
      ps.push_back(make_pair(num, i));
    }
  }
  // 按火柴棒数从小到大排序
  sort(ps.begin(), ps.end());
  int res = 0;
  // used[i] == true ⇔ X中已经有包含火柴棒i的正方形了
  vector<bool> used(M, false);
  for (int i = 0; i < ps.size(); i++) {
    int id = ps[i].second;
    bool ok = true;
    // 是否将正方形id加入X中
    for (int j = p; j < M; j++) {
      if (used[j] && m[j][id]) ok = false;
    }
    if (ok) {
      res++;
      for (int j = p; j < M; j++) {
        if (m[j][id]) used[j] = true;
      }
    }
  }
  return res;
}
```

做到这一步，就可以通过$N \leqslant 5$的所有数据了。与(剩余的正方形数)/(一根火柴所能破坏的最多正方形数)这一下界相比，利用这个下界将能够极为显著地加速搜索。[①]

① 还可以进一步优化。譬如说对于火柴棒A和B，移除B所能破坏的正方形都能通过移除A破坏的话，我们就可以无视B而只考虑A。此外，在这里我们只是简单地从1号火柴棒开始按顺序搜索，也可以考虑通过调整搜索的顺序进行优化。

4.5.2　A*与IDA*

1. IDA*

继续考虑前面的问题。刚刚讲到的剪枝，虽然在较快找到接近最优解的解时能够有效减少状态数，但在尚未找到较好的解时，依然会尝试许多不必要的状态。虽然我们在下界（在前面的代码中就是num + hstar(p, state)）比最优解更大时可以剪枝，但是不知道最优解的话就没法做到了。因此，我们可以不直接去求最优解，而是改成通过搜索判断是否有不超过某个x的解。把x从0开始每次增加1，那么首次找到解时的x便是最优解。这样，搜索过程中就不会访问那些下界比最优解更大的状态了。

将程序改成判断是否有不超过某个x的解，只需要在之前的下界超过x时，停止搜索就好了，代码几乎没什么变动。在本题中，由于我们在没有找到解时会返回INF，所以搜索部分的代码无须改动。

```
void solve() {
  vector<bool> state(S, true);

  // min_res从0开始递增，直到找到解
  min_res = 0;
  while (dfs(0, 0, state) == INF) min_res++;

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

考虑平凡下界的情况（即总有hstar(p, state)=0），此时，这个程序就会像宽度优先搜索一样，按距离初始状态的远近顺序访问各个状态。这被称为迭代加深搜索（IDDFS，Interative Deepening Depth-First Search）。

x=1,2时搜索的情况

而像这次这样，通过估算下界提前剪枝优化后的算法则称为IDA*。它通常可以表述如下。

(1) 给出状态v到目标状态（在前面的问题中就是没有正方形的状态）的距离下界的估算函数$h^*(v)$

(2) 令x=0

(3) 对满足$d(v)+h^*(v)\leqslant x$的状态进行深度优先搜索，判断是否有不超过x的解（$d(v)$表示从初始状态到v的距离）

(4) 如果找到解，则x就是最优解，程序结束
(5) 否则，将x增加1并回到第3步

IDA*中所访问的状态总是满足$d(v)+h^*(v)\leqslant$（最优解），而$h^*(v)$的估值越接近到目标状态的实际距离，则搜索经过的状态数越少。

2. A*

正如深度优先算法可以利用下界优化一样，宽度优先搜索和Dijkstra算法也可以利用下界优化。只要将优先队列中的键改成$d(v)+h^*(v)$就可以了，其中$d(v)$是初始状态到状态v的距离，$h^*(v)$是到目标状态的距离下界。这种算法称为A*。只不过需要注意的是，与宽度优先搜索和Dijkstra算法不同，在选用某些下界进行估算的情况下，优先队列顶端的元素对应的$d(v)$未必已经是初始状态到v的最短距离。

如果对于所有的边(u,v)都有$\text{cost}(u,v)+h^*(u)-h^*(v)\geqslant 0$成立，那么第一次取出某个节点$v$时，对应的$d(v)$就一定是最短距离。但是在Square Destroyer中用到的下界并不满足这样的性质。不过，由于h^*是到目标状态t的距离的下界，所以第一次从队首取出的距离同样就是最短距离。

3. A*与IDA*的比较

A*和IDA*分别是针对宽度优先搜索和深度优先搜索的优化算法。就像在Square Destroyer中看到的那样，可以很容易地把深度优先搜索改写成IDA*。同样地，也可以很容易地把宽度优先搜索或Dijkstra算法改写成A*。

当然，不仅在代码上改写很容易，它们各自也继承了一些改写前算法的特点。例如，IDA*基本不怎么花费内存，而A*则要花费关于搜索空间的线性的内存。另一方面，可以通过不同路径达到同一状态时，IDA*可能会重复多次经过某些状态而导致效率一落千丈，而A*通过选取合适的下界则可以保证每个状态至多检查一次。

通常来说，随着搜索深度的增加，搜索空间的大小呈指数级别增长。所以，虽然IDA*在不断增加递归深度限制的过程中重复搜索了很多状态，但总的访问状态数和最后一次所访问的状态数还是同一数量级的。

专栏　整数规划问题

在线性规划问题之上，再给变量加上必须是整数的限制，就得到了整数规划问题（IP，Integer Programming）。此前所讲解的许多问题都可以规约到整数规划问题。例如，Square Destroyer就可以转为整数规划问题中的集合覆盖问题。虽然整数规划问题与线性规划问题形式相似，但与线性规划问题不同的是，要求解整数规划问题的最优解通常是非常困难的。

在前文中，我们似乎没有任何铺垫就提出了通过最大独立集问题估算 Square Destroyer 的最优解下界。如果将 Square Destroyer 写成整数规划问题的形式，那么也就能自然的导出这个下界了。

首先，把 Square Destroyer 写成整数规划问题的形式。记初始状态中剩余火柴棒的根数为 n，剩余正方形的个数为 m。用一个 n 维向量 x 表示火柴棒的取舍，其中 $x_i=1$ 表示除去火柴棒 i，$x_i=0$ 表示留下火柴棒 i。于是除去火柴棒的根数就是

$$\sum_{i=1}^{n} x_i$$

而正方形 j 将因为取走某个火柴棒而遭到破坏这一条件可以写成

$$\sum x_i \geqslant 1$$

其中，求和中的 i 是所有满足取走火柴 i 将会破坏正方形 j 的 i。利用矩阵和向量，我们就可以将问题写成

$$\text{(PI)}\ \min\{c^T x\colon Ax \geqslant b, x \geqslant 0, x \in \mathbb{Z}^n\}$$

的形式。

让我们来计算该整数规划问题的最优解的下界。先不考虑整数规划问题中的变量应该是整数这一限制。

$$\text{(PL)}\ \min\{c^T x\colon Ax \geqslant b, x \geqslant 0, x \in \mathbb{R}^n\}$$

因为(PL)比(PI)的限制更松，所以(PL 的最优解)≤(PI 的最优解)。再考虑(PL)的对偶问题。

$$\text{(DL)}\ \max\{b^T y\colon A^T y \leqslant c, y \geqslant 0, y \in \mathbb{R}^m\}$$

根据强对偶定理，有(PL 的最优解)=(DL 的最优解)。再给 DL 加上变量应该是整数这一限制，得到的问题限制更紧，所以最优解应该更小。

$$\text{(DI)}\ \max\{b^T y\colon A^T y \leqslant c, y \geqslant 0, y \in \mathbb{Z}^m\}$$

综上，有(DI 的最优解)≤(DL 的最优解)=(PL 的最优解)≤(PI 的最优解)。于是我们得到了原问题(PI)的最优解的三个下界。而这里的(DI)就是我们在 Square Destroyer 中估算最优解的下界时所用的最大独立集问题。在求解 Square Destroyer 时，我们又是通过贪心算法求得最大独立集问题的近似解作为下界的。根据前面的分析，如果我们通过单纯形算法求解(PL)或(DL)，则可以得到更为逼近的下界。

4.6 划分、解决、合并：分治法

➠分治法是算法设计方法的一种。它通过将问题划分为规模更小的子问题，递归地解决划分后的子问题，再将结果合并从而高效地解决问题。

4.6.1 数列上的分治法

逆序数

题目描述请参考 3.3 节中的逆序数问题。

在3.3节中，我们利用树状数组在$O(n \log n)$的时间内解决了这个问题。此外，我们还可以通过一个完全不同的分治算法在$O(n \log n)$的时间内解决这个问题。同样的，我们要做的是统计$i<j$而$a_i>a_j$的逆序对(i,j)的个数。

数列的划分

假设我们要统计数列A中逆序对的个数。如图所示，我们可以将数列A分成两半得到数列B和数列C。于是，数列A中所有的逆序对必居下面三者其一。

(1) i,j都属于数列B的逆序对(i,j)
(2) i,j都属于数列C的逆序对(i,j)
(3) i属于数列B而j属于数列C的逆序对(i,j)

三种逆序对

所以，只要分别统计这三种逆序对，再把结果加起来就好了。对于(1)和(2)，可以通过递归求得。而对于(3)，我们可以对数列C中的每个数字，统计在数列B中比它大的数字的个数，再把结果加起来就好了。这可以通过下面这样在归并排序的同时进行统计而得到。

因为每次递归数列的长度都会减半，所以递归的深度为$O(\log n)$，而每一层总的操作都是$O(n)$，所以总的复杂度为$O(n \log n)$。

```
typedef long long ll;

// 输入
vector<int> A;

ll merge_count(vector<int> &a) {
  int n = a.size();
  if (n <= 1) return 0;

  ll cnt = 0;
  vector<int> b(a.begin(), a.begin() + n / 2);
  vector<int> c(a.begin() + n / 2, a.end());

  cnt += merge_count(b);  // (1)
  cnt += merge_count(c);  // (2)
  // 此时，b和c已经分别排好序了

  // (3)
  int ai = 0, bi = 0, ci = 0;
  while (ai < n) {
    if (bi < b.size() && (ci == c.size() || b[bi] <= c[ci])) {
      a[ai++] = b[bi++];
    } else {
      cnt += n / 2 - bi;
      a[ai++] = c[ci++];
    }
  }

  return cnt;
}

void solve() {
  printf("%lld\n", merge_count(A));
}
```

在这类问题中，我们把问题分割成更小的子问题递归求解，再处理不同子问题之间的部分，这种算法设计方法就是分治法。

4.6.2　树上的分治法

1. 重心分解（Centroid Decomposition）

上一节，我们在数列上运用了分治法。而本节，我们要思考的分治法的运用对象不是数列，而是

树。后面的讨论中，我们统一用n表示数上节点的个数。

对数列分治时，我们选择了在数列中央将数列二等分，可是对树应该如何分割才好呢？如果我们不假思索随意选择顶点或边进行分割，就可能导致子问题规模不均匀，发生退化。划分不均匀时，递归的深度就有可能退化成$O(n)$，而问题的复杂度也可能因此变得很高。

而选择使得删除该顶点后得到的最大子树的顶点数最少的顶点作为分割顶点，似乎是个不错的主意。我们称这样的顶点为重心（Centroid）。事实上，删除重心后得到的所有子树，其顶点数必然不超过$n/2$。因此，如果每次都选择重心进行分割的话，那么每次树的大小也至少减半，所以递归的深度是$O(\log n)$，可以保证不发生退化，从而进行高效处理。

要证明重心具有以上性质是很容易的，下面我们就来证明一下。选取任意顶点作为起点，每次都沿着边向最大子树的方向移动，最终一定会到达某个顶点，将其删除后得到的所有子树的顶点数都不超过$n/2$。如果这样的点存在的话，那么也就可以证明删除重心后得到的所有子树的顶点数都不超过$n/2$。

记当前顶点为v，如果顶点v已经满足上述条件则停止。否则，与顶点v邻接的某个子树的顶点数必然大于$n/2$。假设顶点v与该子树中的顶点w邻接，那么我们就把顶点w作为新的顶点v。不断重复这一步骤，必然会在有限步停止。这是因为对于移动中所用的边(v, w)，必有v侧的子树的顶点数小于$n/2$，w侧的子树的顶点数大于$n/2$，所以不可能再从w移动到v。因而该操作永远不会回到已经经过的顶点，而顶点数又是有限的，所以算法必然在有限步停止。

2. 运用重心分解的问题

Tree （POJ 1741）

给定一棵 n 个顶点构成的树。其中连接顶点 a_i 和 b_i 的边 i 的长度为 l_i。请统计最短距离不超过 k 的顶点的对数。

限制条件

- $1 \leqslant n \leqslant 10000$
- $1 \leqslant l_i \leqslant 1000$

样例

输入

```
n = 5, k = 4
a = {1, 1, 1, 3}
b = {2, 3, 4, 5}
l = {3, 1, 2, 1}
```

输出

```
8
```

n比较大，直接枚举所有顶点对是行不通的，于是我们来考虑基于分治法的算法。

假设我们按重心把树分成了若干子树，那么所要求的顶点对必居下面三者其一。

(1) 顶点v,w属于同一子树的顶点对(v,w)
(2) 顶点v,w属于不同子树的顶点对(v,w)
(3) 顶点s和其他顶点v组成的顶点对(s,v)

首先，对于第(1)种情况，可以通过递归得到。对于第(2)种情况，于是从顶点v到顶点w的路径必然经过了顶点s，只要先求出每个顶点到s的距离，就可以轻松统计出和不超过k的顶点对数。而对于第(3)种情况，只要添加一个到s距离为0的顶点，就可以转为第(2)种情况处理了。

三种顶点对

不过，需要注意的是，应该在(1)中统计的属于同一子树的顶点对，要避免在(2)中进行重复统计。在下面的程序中，我们通过先减去重复统计的顶点对数来避免这一问题。

在递归的每一层我们都做了排序，复杂度是$O(n \log n)$，而递归的深度为$O(\log n)$，所以总的复杂度是$O(n \log^2 n)$。[①]

```
struct edge { int to, length; };

// 输入
int N, K;
vector<edge> G[MAX_N];

bool centroid[MAX_N];      // 顶点是否已经作为重心删除的标记
int subtree_size[MAX_N];   // 以该顶点为根的子树的大小（查找重心时使用）

int ans;  // 答案

// 计算子树的大小(subtree_size)的递归函数
int compute_subtree_size(int v, int p) {
```

① 本题也有利用合适的数据结构，自底向上地合并并统计的$O(n\log_2 n)$的解法。——译者注

```
  int c = 1;
  for (int i = 0; i < G[v].size(); i++) {
    int w = G[v][i].to;
    if (w == p || centroid[w]) continue;
    c += compute_subtree_size(G[v][i].to, v);
  }
  subtree_size[v] = c;
  return c;
}

// 查找重心的递归函数。t是整个连通分量的大小。
// 在以v为根的子树中寻找一个顶点，使得删除该顶点后得到的最大子树的顶点数最少，
// 返回值为pair(最大子树的顶点数，顶点编号)
pair<int, int> search_centroid(int v, int p, int t) {
  pair<int, int> res = make_pair(INT_MAX, -1);
  int s = 1, m = 0;
  for (int i = 0; i < G[v].size(); i++) {
    int w = G[v][i].to;
    if (w == p || centroid[w]) continue;

    res = min(res, search_centroid(w, v, t));

    m = max(m, subtree_size[w]);
    s += subtree_size[w];
  }
  m = max(m, t - s);
  res = min(res, make_pair(m, v));
  return res;
}

// 计算子树中的所有顶点到中心的距离的递归函数
void enumerate_paths(int v, int p, int d, vector<int> &ds) {
  ds.push_back(d);
  for (int i = 0; i < G[v].size(); i++) {
    int w = G[v][i].to;
    if (w == p || centroid[w]) continue;
    enumerate_paths(w, v, d + G[v][i].length, ds);
  }
}

// 统计和不超过K的顶点对的个数
int count_pairs(vector<int> &ds) {
  int res = 0;
  sort(ds.begin(), ds.end());
  int j = ds.size();
  for (int i = 0; i < ds.size(); i++) {
    while (j > 0 && ds[i] + ds[j - 1] > K) --j;
    res += j - (j > i ? 1 : 0);  // 除去和本身组成的顶点对
  }
  return res / 2;
}

// 对顶点v所在的子树，查找中心并分割求解的递归函数
void solve_subproblem(int v) {
  // 查找重心s
```

```
    compute_subtree_size(v, -1);
    int s = search_centroid(v, -1, subtree_size[v]).second;
    centroid[s] = true;

    // (1): 统计按顶点s分割后的子树中的对数
    for (int i = 0; i < G[s].size(); i++) {
      if (centroid[G[s][i].to]) continue;
      solve_subproblem(G[s][i].to);
    }

    // (2), (3): 统计经过s的对数
    vector<int> ds;
    ds.push_back(0);  // 包含顶点s的部分
    for (int i = 0; i < G[s].size(); i++) {
      if (centroid[G[s][i].to]) continue;

      vector<int> tds;
      enumerate_paths(G[s][i].to, s, G[s][i].length, tds);

      ans -= count_pairs(tds);  // 先把会重复统计的部分减掉
      ds.insert(ds.end(), tds.begin(), tds.end());
    }

    ans += count_pairs(ds);
    centroid[s] = false;
  }

  void solve() {
    ans = 0;
    solve_subproblem(0);
    printf("%d\n", ans);
  }
```

4.6.3　平面上的分治法

最近点对问题 （UVa 10245）

给定平面上的 n 个点，求距离最近的两个点的距离。

限制条件

- $1 \leq n \leq 10000$

样例

输入

```
n = 5
x = {0, 6, 43, 39, 189}
y = {2, 67, 71, 107, 140}
```

输出

```
36.2215
```

这是一道经典问题。因为n比较大，直接枚举所有点对是行不通的。于是我们来考虑基于分治法的算法。

假设我们把所有点按x坐标分成了左右两半，那么最近点对的距离就是下面二者的最小值。

(1) 2点p和q同属于左半边或右半边时点对(p, q)的距离
(2) 2点p和q属于不同区域时点对(p, q)的距离

两种点对

首先，对于(1)可以通过递归计算。但是，对于(2)该如何处理呢？事实上，如果直接处理的话，并没有比原文题简单多少，仍然不好处理。不过，如果我们已经知道了(1)部分的最小值，不妨记为d，再考虑下面的(2')。

(2') 2点p和q属于不同区域时，距离小于d的点对(p, q)的距离

在(2)的基础上加上了距离小于d的限制。因为我们已经在(1)中找到了距离为d的点对，所以不需要再考虑那么距离大于等于d的点对了。所以把(2)换成(2')依然可以保证答案的正确性。利用这个条件，所需考虑的点对的数量也就减少了。

首先，我们考虑x坐标。假设将点划分为左右两半的直线为l，其x坐标为x_0。那么根据(2')，到直线l的距离大于等于d的点就没有必要考虑了。我们只需要考虑那些到直线l的距离小于d的点，也就是x坐标满足$x_0-d<x<x_0+d$的点。

接下来，我们考虑y坐标。对于每个点，都只需考虑与那些y坐标不比自己大的点组成的点对。另外，也没有必要考虑那些y坐标相差大于等于d的点。也就是说，对于y坐标为y_p的点，我们只需与考虑y坐标满足$y_p-d<y\leqslant y_p$的点组成的点对就足够了。

对于点p所需考虑的点的范围

综合以上两点可知，我们要检查的点都在$x_0-d<x<x_0+d$且$y_p-d<y\leqslant y_p$的矩形区域内。也许大家可能担心这个区域内会有过多的点需要处理，不过因为这里的d不仅是一个上界，而且是由(1)得到的最小值，所以可以证明该区域内的点不会太多。

因为d是(1)中的最小值，所以同属左半边的点，还有同属右半边的点，其距离都不小于d。因此，把待考虑的矩形区域分成左右两个正方形后，每个正方形内至多只包含三个点。所以可以证明，在该矩形区域内，除去p以外，至少还有$3\times2-1$，即5个点。[①]

为了实现按x坐标划分，我们可以在开始时先将所有点按x坐标排序。另一方面，为了更高效地检查对应矩形区域内的点，在(2')的处理之前，我们要把所有待考虑的点按y坐标排序。为此，我们在递归处理的同时，按y坐标进行归并排序。

这样，递归的深度为$O(\log n)$，而每一层有$O(n)$个操作，所以总的复杂度是$O(n \log n)$。[②]

```
typedef pair<double, double> P;  //  first保存x坐标，second保存y坐标

// 输入
int N;
P A[MAX_N];

// 用于按y坐标归并的比较函数
bool compare_y(P a, P b) {
  return a.second < b.second;
}

// 传入的a已经按x坐标排好序了
double closest_pair(P *a, int n) {
  if (n <= 1) return INF;
  int m = n / 2;
  double x = a[m].first;
```

① 事实上我们无须考虑在与p同侧的正方形内的点，不过考虑了也不影响结果，且实现起来更方便。
② 本题也有利用合适的数据结构，进行平面扫描的$O(n\log n)$的解法。

```
  double d = min(closest_pair(a, m), closest_pair(a + m, n - m));  // (1)
  inplace_merge(a, a + m, a + n, compare_y);  // 归并两个排好序的数列
  // 此时，a已经按y坐标排好序了

  // (2')
  vector<P> b;  // 将到直线的距离小于d的顶点加入
  for (int i = 0; i < n; i++) {
    if (fabs(a[i].first - x) >= d) continue;

    // 从后往前检查b中y坐标相差小于d的点
    for (int j = 0; j < b.size(); j++) {
      double dx = a[i].first - b[b.size() - j - 1].first;
      double dy = a[i].second - b[b.size() - j - 1].second;
      if (dy >= d) break;
      d = min(d, sqrt(dx * dx + dy * dy));
    }
    b.push_back(a[i]);
  }

  return d;
}

void solve() {
  sort(A, A + N);  // 按x坐标排序
  printf("%f\n", closest_pair(A, N));
}
```

4.7 华丽地处理字符串

➠由于字符串是信息最自然的表现形式之一，有许多相关的问题，也有许多的相关的算法。本节将要介绍有关字符串的动态规划算法、字符串匹配和后缀数组。

4.7.1 字符串上的动态规划算法

1. 单字符串的情况

禁止字符串

考虑只由'A', 'G', 'C', 'T'四种字符组成的 DNA 字符串。给定一个长度为 k 的字符串 S。请计算长度恰好为 n 且不包含 S 的字符串的个数。输出个数 mod 10009 后的结果。

限制条件

- $1 \leqslant k \leqslant 100$
- $1 \leqslant n \leqslant 10000$

样例

输入

```
n = 3, k = 2, S = "AT"
```

输出

```
56
```

首先，我们来考虑生成所有满足条件的字符串这一直观的解法。字符串的个数可能高达4^n，这显然是行不通的。接下来，与其在生成字符串后再判断它是否包含S，不如在穷竭搜索的过程中，每在末尾加一个字符，都确保其最后k个字符不等于S。可以发现最后k个字符之前的字符对以后的搜索并无影响。所以，我们可以以剩余字符的个数和最后$k-1$个字符为状态进行动态规划。

这样的状态数依然高达4^{k-1}。不过，事实上可以将其中的许多状态看作是等价的，从而大大减少所需的状态数。首先，我们以S等于"ATCG"为例进行分析。动态规划的状态是字符的个数和最后3个字符。

譬如说，我们可以将最后三个字符为"TTA"的状态和"CCA"的状态看作是等价的。这是因为'A'是S的第一个字符，所以"TT"或"CC"这两个字符对以后是否出现S并无影响。同理可知，对于以'A'结尾的状态，'A'前面的字符是什么并不重要。由此可知，我们可以将"**A"（'*'代表任意一个字符）看作一个状态。

以'T'结尾的状态又如何呢？'T'是S的第二个字符。因此要使该字符对S的出现有影响，其前面的字符必须是'A'，并且同之前分析的一样，再前面的字符是什么并无影响。所以，可以将"*AT"看作一个状态。而如果'T'前面的字符不是'A'的话，那么这个'T'也可以无视。

综合以上分析，我们知道最初的4^3种状态，可以归纳为以下4种

- "A", "*AT", "ATC", 其他

也就是把所生成字符串的后缀和S的前缀相匹配的长度作为状态。所以，状态总数只有k个。

接下来，我们来考虑一个更为复杂的例子，令S等于"ATCATCG"。此时，有个地方需要稍微注意一下。简单地像前面一样处理的话，将得到"ATCATC"和"***ATC"这两个状态。如果我们对"***ATC"的前3个字符不加任何限制的话，那么就发生了以"ATCATC"结尾的状态，同时属于这两个状态这一奇怪现象。不过，要解决这一问题也很容易。最初我们引入'*'时，是用于代替那些已知对以后是否出现S无影响的字符。对于"ATCATC"，如果下一个字符是'G'的话，就得到了S，对S的出现是有影响的。所以"ATCATC"不应该属于"***ATC"。

将这个一般化就会发现，状态应该是所生成字符串的后缀和S的前缀相匹配的长度，只不过当有多种匹配时，应当选择最长的作为状态。

为了让动态规划算法部分更加高效，下面的程序预先处理出了从某个状态添加某个字符后的状态转移表，然后利用该表完成动态规划。预处理部分的复杂度是$O(k^3)$，动态规划的复杂度是$O(kn)$。

```
const char *AGCT = "AGCT";
const int MOD = 10009;

// 输入
int N, K;
string S;

int next[MAX_K][4];  // 添加某个字符后转移到的状态
int dp[MAX_N + 1][MAX_K];

void solve() {
  // 预处理
  for (int i = 0; i < K; i++) {
    for (int j = 0; j < 4; j++) {
      // 在S长度为i的前缀后添加一个字符
      string s = S.substr(0, i) + AGCT[j];
      // 反复删除第一个字符，直到成为S的前缀
```

```
        while (S.substr(0, s.length()) != s) {
          s = s.substr(1);
        }
        next[i][j] = s.length();
      }
    }

    // 动态规划边界的初值
    dp[0][0] = 1;
    for (int i = 1; i < K; i++) dp[0][i] = 0;
    // 动态规划
    for (int t = 0; t < N; t++) {
      for (int i = 0; i < K; i++) dp[t + 1][i] = 0;

      for (int i = 0; i < K; i++) {
        for (int j = 0; j < 4; j++) {
          int ti = next[i][j];
          if (ti == K) continue;  // 不允许出现S
          dp[t + 1][ti] = (dp[t + 1][ti] + dp[t][i]) % MOD;
        }
      }
    }

    int ans = 0;
    for (int i = 0; i < K; i++) ans = (ans + dp[N][i]) % MOD;
    printf("%d\n", ans);
  }
```

2. 多字符串的情况

DNA Repair （POJ 3691）

考虑只由'A', 'G', 'C', 'T'四种字符组成的 DNA 字符串。给定一个原字符串 S，和 n 个禁止模式字符串 $P_1, P_2, \cdots, P_n$。请修改字符串 S，使得其中不包含任何禁止模式。每次修改操作只能将 S 中的某个字符修改为其他字符。如果不存在这样的修改，请输出−1，否则，输出所需的最少修改回数。

限制条件

- $1 \leqslant |S| \leqslant 1000$
- $1 \leqslant n \leqslant 50$
- $1 \leqslant |P_i| \leqslant 20$

样例 1

输入

```
S = "AAAG", P = {"AAA", "AAG"}
```

输出

```
1
```

样例 2

输入

```
S = "TGAATG", P = {"A", "TG"}
```

输出

```
4
```

样例 3

输入

```
S = "AGT", P = {"A", "G", "C", "T"}
```

输出

```
-1
```

这个问题看起来与前面的问题很相似。所以我们考虑从左向右修改，依然以已修改到的位置和对应的后缀为状态进行动态规划。只不过与前面不同的是，这里禁止出现的字符串不只一个，而是有多个。那么应该用哪些状态来表示后缀才好呢？

实际上，这并不难。在前面的问题中，我们取字符串S的所有前缀为状态，这里也只要取所有字符串P_i的所有前缀为状态就好了。不过需要注意的是，我们可能从不同的字符串得到相同的前缀。譬如说，"AA"既是"AAA"的前缀，也是"AAG"的前缀。在"AA"后面添加'A'可以得到"AAA"，添加'G'则会得到"AAG"。因此，需要把像"AA"这样的多个字符串的公共前缀作为同一个状态进行正确地处理。

与前面的问题一样，下面的代码中，首先是预处理出状态及其转移关系，然后再进行动态规划。预处理的复杂度是$O(n^2l^2+nl^3\log(nl))$，动态规划的复杂度是$O(nl|S|)$，其中l表示P_i的最大长度。

```
const char *AGCT = "AGCT";

// 输入
int N;
string S, P[MAX_N];

// 预处理得到的数据
int next[MAX_STATE][4];   // 添加某个字符后转移到的状态
bool ng[MAX_STATE];       // 是否是禁止转移到的状态

int dp[MAX_LEN_S + 1][MAX_STATE];
```

```
void solve() {
  // 首先枚举出所有的字符串前缀
  vector<string> pfx;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j <= P[i].length(); j++) {
      pfx.push_back(P[i].substr(0, j));
    }
  }
  // 排序并去重
  sort(pfx.begin(), pfx.end());
  pfx.erase(unique(pfx.begin(), pfx.end()), pfx.end());
  int K = pfx.size();

  // 计算各个状态的相关信息
  for (int i = 0; i < K; i++) {
    // 如果后缀和禁止模式匹配的话，就是禁止转移到的状态
    ng[i] = false;
    for (int j = 0; j < N; j++) {
      ng[i] |= P[j].length() <= pfx[i].length()
          && pfx[i].substr(pfx[i].length() - P[j].length(), P[j].length()) == P[j];
    }
    for (int j = 0; j < 4; j++) {
      // 添加一个字符后得到的字符串
      string s = pfx[i] + AGCT[j];
      // 反复删除第一个字符，直到等于某个状态的字符串，该状态就是转移到的状态
      int k;
      for (;;) {
        k = lower_bound(pfx.begin(), pfx.end(), s) - pfx.begin();
        if (k < K && pfx[k] == s) break;
        s = s.substr(1);
      }
      next[i][j] = k;
    }
  }

  // 动态规划的边界初值
  dp[0][0] = 1;
  for (int i = 1; i < K; i++) dp[0][i] = 0;
  // 动态规划
  for (int t = 0; t < S.length(); t++) {
    for (int i = 0; i < K; ++i) dp[t + 1][i] = INF;

    for (int i = 0; i < K; i++) {
      if (ng[i]) continue;
      for (int j = 0; j < 4; j++) {
        int k = next[i][j];
        dp[t + 1][k] = min(dp[t + 1][k], dp[t][i] + (S[t] == AGCT[j] ? 0 : 1));
      }
    }
  }

  int ans = INF;
  for (int i = 0; i < K; ++i) {
```

```
    if (ng[i]) continue;
    ans = min(ans, dp[S.length()][i]);
  }
  if (ans == INF) puts("-1");
  else printf("%d\n", ans);
}
```

专栏 Trie

像这样以字符串的前缀为状态的动态规划，也被称为 Trie 上的动态规划。所谓 Trie，指的是某个字符串集合对应的形如下图的有根树。树的边上对应有一个字符，每个顶点代表从根到该节点的路径所对应的字符串。其中双圆圈表示顶点所代表的字符串是实际字符串集合的元素。

Trie

虽然在程序设计竞赛中并非非用不可，但我们可以把Trie当作一个高效维护字符串集合的数据结构。比如说，如果利用二叉查找树来维护n个字符串的话，查找长度为l的字符串的复杂度为$O(l \log n)$，而换作Trie则只要$O(l)$。前面的动态规划的状态，也正对应于Trie中的顶点。顺带一提，Trie读作“try”。

专栏 更加高效地完成字符串DP的预处理

这里介绍的两道题，都包含计算状态转移关系的预处理。如果利用 KMP 算法或 Aho-Corasick 算法等字符串匹配算法进行预处理的话，则可以在字符串长度（多字符串时则是长度之和）的线性时间内计算出这个转移关系。

4.7.2 字符串匹配

寻找字符串S中字符串T出现的位置或次数的问题属于字符串匹配问题。我们在接下来的讨论中假设S的长度为n，T的长度为m。最朴素的想法是，枚举所有起始位置，再直接检查是否匹配，复杂度为$O(nm)$的算法。还有几个更为高效的算法。而在此我们只介绍实现起来较为容易，而在一些稍作变化的问题中同样适用，并且可以简单地推广到二维情况的哈希算法[1]。

① 哈希也叫做散列。——译者注

将哈希算法用于字符算匹配的原理非常简单。对于每个起始位置，我们不是$O(m)$地直接比较字符串是否匹配，而是$O(l)$地比较长度为m的字符串子串的哈希值与T的哈希值是否相等。虽然即使哈希值相等字符串也未必相等，但如果哈希值是随机分布的话，不同的字符串哈希值相等的概率是很低的，可以当作这种情况不会发生[①]。

但是，如果我们采用$O(m)$的算法计算长度为m的字符串子串的哈希值的话，那复杂度还是$O(nm)$。这里我们要使用一个叫做滚动哈希的优化技巧。选取两个合适的互素常数b和$h(l<b<h)$，假设字符串$C=c_1c_2\cdots c_m$，定义哈希函数

$$H(C)=(c_1b^{m-1}+c_2b^{m-2}+c_3b^{m-3}+\cdots+c_mb^0) \bmod h$$

其中b是基数，相当于把字符串看作b进制数。这样，字符串$S=s_1s_2\cdots s_n$从位置$k+1$开始长度为m的字符串子串$S[k+1..k+m]$的哈希值，就可以利用从位置k开始的字符串子串$S[k..k+m-1]$的哈希值，直接进行如下计算。

$$H(S[k+1..k+m])=(H(S[k..k+m-1]) \times b-s_kb^m+s_{k+m}) \bmod h$$

于是，只要不断这样计算开始位置右移一位后的字符串子串的哈希值，就可以在$O(n)$时间内得到所有位置对应的哈希值，从而可以在$O(n+m)$时间内完成字符串匹配。在实现时，可以用64位无符号整数计算哈希值，并取h等于2^{64}，通过自然溢出省去求模运算。[②]

```
typedef unsigned long long ull;

const ull B = 100000007;  // 哈希的基数

// a是否在b中出现
bool contain(string a, string b) {
  int al = a.length(), bl = b.length();
  if (al > bl) return false;

  // 计算B的al次方
  ull t = 1;
  for (int i = 0; i < al; i++) t *= B;

  // 计算a和b长度为al的前缀对应的哈希值
  ull ah = 0, bh = 0;
  for (int i = 0; i < al; i++) ah = ah * B + a[i];
  for (int i = 0; i < al; i++) bh = bh * B + b[i];
```

① 虽然在许多实际应用中，这种“鸵鸟法”确实行得通，但需要注意的是，哈希值发生冲突的概率也许比直观想象中来得高。根据生日攻击理论，对于哈希值在$[0, n)$均匀分布的哈希函数，首次冲突发生的期望不是$O(n)$，而是$O(\sqrt{n})$。在选取哈希算法参数时，可以把这一结论作为一个参考。——译者注

② 这种利用滚动哈希的字符串匹配算法叫做Rabin-Karp算法。不过，原本的Rabin-Karp算法在哈希值相等时，还要用传统的字符串比较算法来判断字符串是否相等。而在程序设计竞赛中，往往会特意准备一些出现大量相等情况的测试数据。如果一一检查的话，就会导致复杂度退化成$O(nm)$。而不同字符串的哈希值发生冲突的概率非常低，通常可以忽视。因此，在程序设计竞赛中，我们通常只比较哈希值，而不再做朴素的检查。

```
  // 对b不断右移一位，更新哈希值并判断
  for (int i = 0; i + al <= bl; i++) {
    if (ah == bh) return true;  //  b从位置i开始长度为al的字符串子串等于a
    if (i + al < bl) bh = bh * B + b[i + al] - b[i] * t;
  }
  return false;
}
```

当然，不光是右移一位，对于左移一位、左端或右端加长一位或是缩短一位的情况，也能够进行类似处理。譬如说，假设要求S的后缀和T的前缀相等的最大长度，也可以利用滚动哈希在$O(n+m)$的时间内高效地求得。[①]

```
typedef unsigned long long ull;

const ull B = 100000007;  // 哈希的基数

//  a的后缀和b的前缀相等的最大长度
int overlap(string a, string b) {
  int al = a.length(), bl = b.length();
  int ans = 0;
  ull ah = 0, bh = 0, t = 1;
  for (int i = 1; i <= min(al, bl); i++) {
    ah = ah + a[al - i] * t;  // a的长度为i的后缀的哈希值
    bh = bh * B + b[i - 1];   // b的长度为i的前缀的哈希值
    if (ah == bh) ans = i;
    t *= B;
  }
  return ans;
}
```

星座（POJ 3690）

给定一个由'*'和'0'组成的，大小为 $N\times M$（N行 M列）的匹配对象和 T个大小为 $P\times Q$的匹配模式。请输出在匹配对象中至少出现过一次的匹配模式的个数。

限制条件

- $1\leqslant N, M\leqslant 1000$
- $1\leqslant T\leqslant 100$
- $1\leqslant P, Q\leqslant 50$

① 本题也可以用KMP算法解决。

样例

输入

```
N=M=3, P=Q=2, T=2

匹配对象:
*00
0**
*00

匹配模式:
**  *0
00  **
```

输出

```
1（只有第一个模式）
```

这里要做的不是字符串匹配，而是二维网格的匹配，同样可以运用循环哈希。首先把每一行看成一个字符串，计算从每个位置开始长度为Q的字符串子串的哈希值。然后再把得到的哈希值在列方向看成一个字符串，计算从每个位置开始长度为P的字符串子串的哈希值。这样，我们高效地计算得到了所有$P \times Q$的子阵的哈希值。在两次哈希值的计算中，我们选用了不同的基数。[1]

计算二维网格的哈希值

```
typedef unsigned long long ull;

// 输入
int N, M, T, P, Q;
char field[MAX_SIZE][MAX_SIZE];              // 匹配对象
char patterns[MAX_T][MAX_SIZE][MAX_SIZE];    // 匹配模式

ull hash[MAX_SIZE][MAX_SIZE], tmp[MAX_SIZE][MAX_SIZE];

// 计算a的所有P×Q子阵对应的哈希值
void compute_hash(char a[MAX_SIZE][MAX_SIZE], int n, int m) {
```

① 本题也可以用Aho-Corasick算法解决。

```
const ull B1 = 9973;
const ull B2 = 100000007;

ull t1 = 1;  // B1的Q次方
for (int j = 0; j < Q; j++) t1 *= B1;

// 按行方向计算哈希值
for (int i = 0; i < n; i++) {
  ull e = 0;
  for (int j = 0; j < Q; j++) e = e * B1 + a[i][j];

  for (int j = 0; j + Q <= m; j++) {
    tmp[i][j] = e;
    if (j + Q < m) e = e * B1 - t1 * a[i][j] + a[i][j + Q];
  }
}

ull t2 = 1;  // B2的P次方
for (int i = 0; i < P; i++) t2 *= B2;

// 按列方向计算哈希值
for (int j = 0; j + Q <= m; j++) {
  ull e = 0;
  for (int i = 0; i < P; i++) e = e * B2 + tmp[i][j];

  for (int i = 0; i + P <= n; i++) {
    hash[i][j] = e;
    if (i + P < n) e = e * B2 - t2 * tmp[i][j] + tmp[i + P][j];
  }
}

void solve() {
  // 将所有模式的哈希值放入multiset中
  multiset<ull> unseen;
  for (int k = 0; k < T; k++) {
    compute_hash(patterns[k], P, Q);
    unseen.insert(hash[0][0]);
  }

  // 将出现的哈希值从multiset中删除
  compute_hash(field, N, M);
  for (int i = 0; i + P <= N; i++) {
    for (int j = 0; j + Q <= M; j++) {
      unseen.erase(hash[i][j]);
    }
  }

  // 通过相减得到出现的模式的个数
  int ans = T - unseen.size();
  printf("%d\n", ans);
}
```

4.7.3 后缀数组

字符串后缀（Suffix）指的是从字符串的某个位置开始到其末尾的字符串子串。我们认为原串和空串也是后缀。反之，从字符串开头到某个位置的字符串子串则称为前缀。

后缀数组（Suffix Array）指的是将某个字符串的所有后缀按字典序排序后得到的数组。不过数组中并不需要直接保存所有的后缀字符串，只要记录对应的起始位置就好了。下文中，我们用$S[i..]$来表示字符串S从位置i开始的后缀。

"abracadabra"对应的后缀数组sa

i	sa[i]	S[sa[i]...]	i	sa[i]	S[sa[i]...]
0	11	（空字符串）	6	8	bra
1	10	a	7	1	bracadabra
2	7	abra	8	4	cadabra
3	0	abracadabra	9	6	dabra
4	3	acadabra	10	9	ra
5	5	adabra	11	2	racadabra

后缀数组不但能够高效计算得到，而且能够用于解决许多问题，是非常强有力的工具。

1. 后缀数组的计算

假设我们要计算长度为n的字符串S的后缀数组。最朴素的做法就是直接把所有后缀进行排序，将n个长度为$O(n)$的字符串进行排序的复杂度为$O(n^2\log n)$。而如果灵活运用所有的字符串都是S的后缀这一性质，就可以得到更高效的算法。下面就给大家介绍其中较为简单的一种——由Manber和Myers发明的$O(n \log^2 n)$复杂度的算法。

该算法的基本思想是倍增。首先计算从每个位置开始的长度为2的子串的顺序，再利用这个结果计算长度为4的子串的顺序，接下来计算长度为8的子串的顺序，不断倍增，直到长度大于等于n就得到了后缀数组。下面，我们用$S[i, k]$来表示从位置i开始的长度为k的字符串子串。其中，当从位置i开始，剩余字符不足k个时，表示的是从位置i开始到字符串末尾的子串。

计算abracadabra的后缀数组的过程

要计算长度为2的子串的顺序，只要排序两个字符组成的数对就好了。现在假设已经求得了长度为k的子串的顺序，要求长度为$2k$的子串的顺序。记$\text{rank}_k(i)$为$S[i, k]$在所有排好序的长度为k的子串中是第几小的。要计算长度为$2k$的子串的顺序，就只要对两个rank组成的数对进行排序就好了。我们通过对$\text{rank}_k(i)$与$\text{rank}_k(i+k)$的数对和$\text{rank}_k(j)$与$\text{rank}_k(j+k)$的数对的比较来替代对$S[i, 2k]$和$S[j, 2k]$的直接比较。因为比较$\text{rank}_k(i)$和$\text{rank}_j(j)$就相当于比较$S[i, k]$和$S[j, k]$，比较$\text{rank}_k(i+k)$和$\text{rank}_j(j+k)$就相当于比较$S[i+k, k]$和$S[j+k, k]$。所以，我们可以这样高效地比较长度为$2k$的子串，并将它们排序。

第几小

sa[i]	S[sa[i], 2]	rank_2(sa[i])
11	(空字符串)	0
10	a	1
0	ab	2
7	ab	2
3	ac	3
5	ad	4
1	br	5
8	br	5
4	ca	6
6	da	7
2	ra	8
9	ra	8

首先排序长度为2的子串

排序时，用长度为2的rank的数对的比较来替代长度为4的子串的直接比较

sa[i]	S[sa[i],4]	rank_2(sa[i])	rank_2(sa[i]+2)
11	(空字符串)	0	−1
10	a	1	−1
0	abra	2	8
7	abra	2	8
3	acad	3	4
5	adab	4	2
8	bra	5	1
1	brac	5	3
4	cada	6	7
6	dabr	7	5
9	ra	8	0
2	raca	8	6

再利用其结果对长度为4的子串进行排序

利用长度为2的顺序计算长度为4的顺序

```
int n, k;
int rank[MAX_N + 1];
int tmp[MAX_N + 1];

// 比较(rank[i], rank[i + k])和(rank[j], rank[j + k])
bool compare_sa(int i, int j) {
  if (rank[i] != rank[j]) return rank[i] < rank[j];
  else {
    int ri = i + k <= n ? rank[i + k] : -1;
    int rj = j + k <= n ? rank[j + k] : -1;
    return ri < rj;
  }
}

// 计算字符串S的后缀数组
void construct_sa(string S, int *sa) {
  n = S.length();

  // 初始长度为1，rank直接取字符的编码
  for (int i = 0; i <= n; i++) {
    sa[i] = i;
    rank[i] = i < n ? S[i] : -1;
  }

  // 利用对长度为k的排序的结果对长度为2k的排序
```

```
    for (k = 1; k <= n; k *= 2) {
      sort(sa, sa + n + 1, compare_sa);

      // 先在tmp中临时存储新计算的rank，再转存回rank中
      tmp[sa[0]] = 0;
      for (int i = 1; i <= n; i++) {
        tmp[sa[i]] = tmp[sa[i - 1]] + (compare_sa(sa[i - 1], sa[i]) ? 1 : 0);
      }
      for (int i = 0; i <= n; i++) {
        rank[i] = tmp[i];
      }
    }
  }
```

此外还有许多不同的计算后缀数组的算法。譬如说像SA-IS这样线性复杂度的高效算法等。不过在程序设计竞赛中，多数情况下使用上述的算法就足够了。

2. 基于后缀数组的字符串匹配

后缀数组最基本的应用便是字符串匹配了。假设已经计算好了字符串S的后缀数组，现在要求字符串T在字符串S中出现的位置，只要通过二分搜索就可以在$O(|T|\log|S|)$时间完成。当$|S|$比较大时，比前面介绍的$O(|T|+|S|)$的算法更为高效，所以需要对同样的字符串做多次匹配时，该算法更有优势。

```
  bool contain(string S, int *sa, string T) {
    int a = 0, b = S.length();
    while (b - a > 1) {
      int c = (a + b) / 2;
      // 比较S从位置sa[c]开始长度为|T|的子串与T
      if (S.compare(sa[c], T.length(), T) < 0) a = c;
      else b = c;
    }
    return S.compare(sa[b], T.length(), T) == 0;
  }
```

3. 后缀数组的应用

Sequence （POJ 3581）

给定 N 个数字组成的序列 $A_1, A_2, \cdots, A_n$。其中 A_1 比其他数字都大。现在要把这个序列分成三段，并将每段分别反转，求能得到的字典序最小的序列是什么？要求分得的每段都不为空。

限制条件

- $N \leqslant 200000$

样例

输入

```
N = 5
A = {10, 1, 2, 3, 4}
```

输出

```
1 10 2 4 3（分成{10, 1}, {2}, {3, 4}三段）
```

首先，确定第一段的分割位置是比较简单的。由于有A_1比其他数字都大这一条件，确定第一段的分割位置时只需考虑第一段就足够了。简单来讲，就是只需选择反转之后字典序最小的前缀就好了。这等价于求反转后的字符串中字典序最小的后缀，利用后缀数组即可轻松解决。

然后，把剩余部分分割成两段。不过这次这两部分不是独立的，不能光比较前半部分的字典序取其中最小者。不过，将序列分割成两段再分别反转得到的序列，可以看作是将两个原序列拼接得到的新序列中的某个子串反转后得到的序列，请参见下图。因此，只要计算新序列反转后的后缀数组，在从中选取字典序最小的合适的后缀就好了。

将两个原序列拼接得到的新序列

```
// 输入
int N, A[MAX_N];

int rev[MAX_N * 2], sa[MAX_N * 2];

void solve() {
  // 将A反转，并计算其后缀数组
  reverse_copy(A, A + N, rev);
  construct_sa(rev, N, sa);  // 计算整数序列后缀数组的函数

  // 确定第一段的分割位置
  int p1;
  for (int i = 0; i < N; i++) {
    p1 = N - sa[i];
    if (p1 >= 1 && N - p1 >= 2) break;
  }

  // 将p1之后的字符串反转并重复2次，再计算其后缀数组
  int m = N - p1;
  reverse_copy(A + p1, A + N, rev);
  reverse_copy(A + p1, A + N, rev + m);
  construct_sa(rev, m * 2, sa);

  // 确定后两段的分割位置
```

```
    int p2;
    for (int i = 0; i <= 2 * m; i++) {
      p2 = p1 + m - sa[i];
      if (p2 - p1 >= 1 && N - p2 >= 1) break;
    }

    reverse(A, A + p1);
    reverse(A + p1, A + p2);
    reverse(A + p2, A + N);
    for (int i = 0; i < N; i++) printf("%d\n", A[i]);
  }
```

4. 高度数组（LCP Array，Longest Common Prefix Array）

所谓高度数组，指的是由后缀数组中相邻两个后缀的最长公共前缀（LCP，Longest Common Prefix）的长度组成的数组。记后缀数组为sa，高度数组为lcp，则有后缀S[sa[i]...]与S[sa[i+1]...]的最长公共前缀的长度为lcp[i]。我们可以在$O(n)$时间内高效地求得高度数组，有了高度数组，后缀数组将成为一个更加有力的工具。高度数组的计算虽然简单，但非常巧妙，使用了类似尺取法的技巧。记rank[i]为位置i开始的后缀在后缀数组中的顺序，即有rank[sa[i]]=i。

abracadabra所对应的后缀数组sa和高度数组lcp

i	sa[i]	lcp[i]	S[sa[i]...]	i	sa[i]	lcp[i]	S[sa[i]...]
0	11	0	（空字符串）	6	8	3	bra
1	10	1	a	7	1	0	bracadabra
2	7	4	abra	8	4	0	cadabra
3	0	1	abracadabra	9	6	0	dabra
4	3	1	acadabra	10	9	2	ra
5	5	0	adabra	11	2	–	racadabra

我们从位置0的后缀开始，从前往后依次计算后缀S[i...]与后缀S[sa[rank[i]−1]...]（即后缀数组中的前一个后缀）的最长公共前缀的长度。此时，假设我们已经求得了位置i对应的高度h_i，那么我们可以证明位置i+1对应的高度不小于h_i−1。为什么呢？记k=sa[rank[i]−1]，已知后缀S[i...]和S[k...]的头部h_i个字符是相等的，那么后缀S[i+1...]和S[k+1...]分别是二者去除首个字符的结果，所以它们头部h_i−1个字符是相等的。虽然在后缀数组中，S[i+1...]前面一个元素未必就是S[k+1...]，但即便如此，公共前缀的长度也是只增不减的。因此，只要从h_i−1开始检查，计算最长公共前缀的长度就好了。

高度数组的计算

因为高度最多增加n次，所以总的复杂度是$O(n)$。如果把这个问题当作位置i对应的区间是[i, i+h_i)

的尺取法来看，就很容易理解。区间的左右端点始终不会向左移，并且是不超过n的整数。

```
int rank[MAX_N + 1];

// 传入字符串S和对应的后缀数组sa，计算高度数组lcp
void construct_lcp(string S, int *sa, int *lcp) {
  int n = S.length();
  for (int i = 0; i <= n; i++) rank[sa[i]] = i;

  int h = 0;
  lcp[0] = 0;
  for (int i = 0; i < n; i++) {
    // 计算字符串中从位置i开始的后缀及其在后缀数组中的前一个后缀的LCP
    int j = sa[rank[i] - 1];

    // 将h先减去首字母的1长度，在保持前缀相同前提下不断增加
    if (h > 0) h--;
    for (; j + h < n && i + h < n; h++) {
      if (S[j + h] != S[i + h]) break;
    }

    lcp[rank[i] - 1] = h;
  }
}
```

如果再将后缀数组与数据结构一节中介绍的Range Minimum Query相结合，我们就不光可以得到后缀数组内相邻两个后缀的最长公共前缀，还可以得到任意两个后缀的最长公共前缀。假设有rank[i]<rank[j]，那么从位置i和j开始的后缀的最长公共前缀的长度就是lcp[rank[i]], lcp[rank[i]+1],…, lcp[rank[j]−1]中的最小值。

5. 后缀数组与高度数组的应用

最长公共子串（POJ 2217）

给定两个字符串 S 和 T。请计算两个字符串最长的公共字符串子串的长度。

限制条件

- $1 \leqslant |S|, |T| \leqslant 10000$

样例

输入

```
S = ABRACADABRA
T = ECADADABRBCRDAR
```

输出

```
5（最长公共子串为ADABR）
```

注意本题和初级篇动态规划一节中所讲的最长公共子序列问题是不同的。这里的是子串而非子序列，子串要求是连续的，而子序列则不然[①]。利用后缀数组和高度数组，就可以高效地求解本问题。

首先来考虑一个简化的问题，计算一个字符串中至少出现两次的最长子串。答案一定会在后缀数组中相邻两个后缀的公共前缀之中，所以只要考虑它们就好了。这是因为子串的开始位置在后缀数组中相距越远，其公共前缀的长度也就越短。因此，高度数组的最大值其实就是答案。

再来考虑原问题的解法。因为对于两个字符串，不好直接运用后缀数组，所以我们可以把S和T，通过在中间插入一个不会出现的字符（例如'\0'）拼成一个字符串S'。然后，和刚才一样计算S'的后缀数组，检查后缀数组中的所有相邻后缀。其中，分属于S和T的不同字符串的后缀的lcp的最大值就是答案。而要知道后缀是属于S还是T，可以由其在S'中的位置直接判断。[②]

```
// 输入
string S, T;

int sa[MAX_L], lcp[MAX_L];

void solve() {
  int sl = S.length();
  S += '\0' + T;

  construct_sa(S, sa);
  construct_lcp(S, sa, lcp);

  int ans = 0;
  for (int i = 0; i < S.length(); i++) {
    if ((sa[i] < sl) != (sa[i + 1] < sl)) {
      ans = max(ans, lcp[i]);
    }
  }
  printf("%d\n", ans);
}
```

① 子序列（subsequence）指的是在不改变序列中元素顺序的前提下，删除一些元素后得到的新序列。子串（substring）则是原串中连续的一段，也可以定义为前缀的后缀或后缀的前缀。在这里，序列（sequence）和串（string）是同一概念，而不是狭义的数列和字符串。是否要求连续是子序列和子串唯一的区别。——译者注

② 这里，我们在中间插入了不会出现在字符串中的null字符('\0')。这是因为POJ 2217对输入的字符串中的字符没有特别限制。通常，多数情况有输入的字符串只包含字母之类的限制。由于使用null字符可能引入意想不到的bug，所以这时不用null字符，而是用'$'之类的字符比较好。

最长回文子串

给定字符串 S。请计算其中是回文的最长字符串子串的长度。所谓回文，指的是正着读和反着读都一样的字符串。

限制条件

- $1 \leq |S| \leq 100000$
- S 只包含小写英文字母

样例

输入

```
S = mississippi
```

输出

```
7（ississi是回文）
```

首先，看一下长度为奇数的回文。对于字符串上的每个位置i，如果知道从i开始的后缀和到i为止的前缀反转后的字符串的最长公共前缀的长度的话，也就知道了以第i个字符为对称中心的最长回文的长度了。因此，我们用在S中不会出现的字符（例如'$'）将$S$和$S$反转后的字符串拼接起来，得到字符串$S'$，再计算$S'$的后缀数组。于是，从$i$开始的后缀和到$i$为止的前缀反转后的字符串就都是$S'$中的后缀了，利用高度数组，就可以轻而易举地求得它们最长公共前缀的长度。对于长度为偶数的回文的处理也基本相同。[1]

```
// 输入
string S;

int sa[MAX_L + 1], lcp[MAX_L],  rank[MAX_L + 1];

void solve() {
  int n = S.length();
  string T = S;
  reverse(T.begin(), T.end());
  S += '$' + T;

  construct_sa(S, sa);
  construct_lcp(S, sa, lcp);
  for (int i = 0; i <= S.length(); i++) rank[sa[i]] = i;
  construct_rmq(lcp, S.length() + 1);  // 初始化RMQ

  int ans = 0;
```

① 查找回文，还有Manacher算法等更为高效简单的算法。

```
    // 以第i个字符对称的奇数长回文
    for (int i = 0; i < n; i++) {
      int j = n * 2 - i;
      int l = query_rmq(min(rank[i], rank[j]), max(rank[i], rank[j]));
      ans = max(ans, 2 * l - 1);
    }

    // 以第i-1和第i个字符对称的偶数长回文
    for (int i = 1; i < n; i++) {
      int j = n * 2 - i + 1;
      int l = query_rmq(min(rank[i], rank[j]), max(rank[i], rank[j]));
      ans = max(ans, 2 * l);
    }

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

4.8 一起来挑战 GCJ 的题目（3）

➠GCJ中设有一些需要灵活的想象力的难题。最后，我们就来挑战一下这样的问题吧。

4.8.1 Mine Layer

Mine Layer（2008 World Final C）

类似于扫雷游戏，在一些格子中散布着一些地雷，具体的埋藏位置并不清楚，但知道每个格子及其周围八个格子的地雷的总数。请问此时正中间那一行最多可能有多少地雷？（题目假定所有的输入都是奇数行的）

输入信息的例子和对应的地雷分布的例子

限制条件

- 输入有 R 行 C 列

Small

- R=3, 5
- $3 \leqslant C \leqslant 5$

Large

- $3 \leqslant R \leqslant 49$，$R$ 是奇数
- $3 \leqslant C \leqslant 49$

样例 1

输入

```
R = 3, C = 3
各个格子的信息如下
2 2 1
3 4 3
2 3 2
```

输出

```
1(地雷的分布只有右图一种)
```

样例1对应的地雷的分布

样例 2

输入

```
R = 3, C = 3
各个格子的信息如下
2 2 1
3 4 3
2 3 2
```

输出

```
1(对应题目描述中的例图)
```

1. 穷竭搜索

稍加思索便会发现要妥善处理地雷的影响并不容易，要应用动态规划之类的方法比较困难。

于是首先想到的解法便是穷竭搜索。利用递归函数，从左上角开始，枚举每个格子有地雷或者没有地雷。但是，Large的输入规模行列都达到了49，要用这个算法解决是很困难的。

2. 一维的情况

首先让我们从如下简化的一维版的问题开始考虑。每个格子内有一个数字，但具体数字不详。但知道该格子和左右两个相邻的格子内的数字之和（下面称这个和为信息）。

一维版问题的例子和格子中的数字的例子

那么，这时正中央格子中的数字最大是多少呢？

3. 按模3的余数分类讨论

首先让我们来考虑一下上面的样例。把下图灰色格子的信息加起来，就可以得到所有格子的数字之和。

计算全体之和

而把下图灰色格子的信息加起来，就得到了除正中央格子以外所有格子的数字之和。

计算正中央之外的数字之和

将这两个结果相减，就可以推出正中央的数字为(4+8+5)–(8+6)=3。

当长度模3余1时，我们都可以这样计算得到正中央的数字。例如，对于长度为13的情况，就可以像下图这样计算。注意比上面的例子更大的模3余1的长度应该是13，而不是10，因为长度应该是奇数，这样才存在正中央的格子。

长度为13的情况

同样地，当长度模3余2时，也可以像下图一样，求出两种和并相减，得到正中央的数字。

长度为5（模3余2）的情况

另一方面，当长度模3余0时，无法直接计算除正中央格子以外所有格子的数字之和。但是，可以像下图一样，求出重复计算了正中央格子两次的和，再通过与全体之和相减得到正中央的数字。

长度为9（模3余0）的情况

4. 推广到二维

原问题是二维的，但不会带来什么困难。只要先用前面介绍的方法对各行算出全体之和，我们就可得知每行及其上下两行所含的地雷总数。然后再对各行的全体之和运用一维版问题的方法，就可以求得正中央那一行地雷的个数了。

二维情况的例子

5. 本题的陷阱

本题要求最大值，但其实值是唯一的，这是本题设置的一个大陷阱。写成求最大值就是想引选手上当，而事实上值只有一个。尤其是对于那些知道扫雷这类游戏没有简单算法可解的选手来说，它们很容易想偏，从而忽略了这种简单的解法。

```
// 输入
int R, C;
int A[MAX_RC][MAX_RC];



// 计算长度为n的一维版问题的总和
int total(int *a, int n) {
  int res = 0;
  // 按模3的余数分类讨论
  if (n % 3 == 1 || n % 3 == 2) {
    for (int i = 0; i < n; i += 3) {
      res += a[i];
    }
  }
  else {
    for (int i = 1; i < n; i += 3) {
      res += a[i];
```

```
    }
  }
  return res;
}


// 计算长度为n的一维版问题正中央的数字
int center(int *a, int n) {
  int res;
  // 按模3的余数分类讨论
  if (n % 3 == 1) {
    res = total(a, n);
    for (int i = 1; i < n / 2; i += 3) {
      res -= a[i];
      res -= a[n - i - 1];
    }
  }
  else if (n % 3 == 2) {
    res = total(a, n);
    for (int i = 0; i < n / 2; i += 3) {
      res -= a[i];
      res -= a[n - i - 1];
    }
  }
  else {
    res = 0;
    for (int i = 0; i < n / 2; i += 3) {
      res += a[i];
      res += a[n - i - 1];
    }
    res -= total(a, n);
  }
  return res;
}



void solve() {
  // 计算各行的总和
  int rows[49];
  for (int i = 0; i < R; i++) {
    rows[i] = total(A[i], C);
  }

  // 求解一维版问题
  int ans = center(rows, R);
  printf("%d\n", ans);
}
```

4.8.2 Year of More Code Jam

Year of More Code Jam （2009 World Final A）

女孩 Sphinny 非常喜欢参加程序设计竞赛，她对参加今年的联赛非常有兴趣。今年共将举办 T 场联赛，其中第 i 场联赛包含 m_i 轮，其中的第 j 轮将在联赛开始后的第 $d_{i,j}$ 天举行（联赛开始的日期就是举行第 1 轮的日期，即总有 $d_{i,1}=1$）。各个联赛的开始日期还未确定，有可能会发生不同的联赛的一些轮次在同一天举行的情况。Sphinny 非常喜欢解题，同一天内举行的轮次越多，她的幸福感也越高。如果同一天内有 S 轮比赛的话，该天的幸福感就是 S^2。一年共有 N 天，各个联赛的开始日期在所有日期均匀分布。请计算她今年的幸福感的期望值，并以 $K+A/B$ 的形式输出（注：如果联赛在年底开始，其中一些轮次在第二年举行的话，这些幸福感是不属于今年的）。

限制条件

- $1 \leqslant N \leqslant 10^9$
- $2 \leqslant m_i \leqslant 50$
- $1 = d_{i,1} < d_{i,2} < d_{i,3} < \cdots < d_{i,m_i} \leqslant 10000$

Small

- $1 \leqslant T \leqslant 2$

Large

- $1 \leqslant T \leqslant 50$

样例 1

输入

```
N = 1
T = 1
m = {2}
d = {{1, 2}}
```

输出

```
1+0/1（由于N=1，所以联赛必定在第1天开始，只有第1轮会在今年举行）
```

样例 2

输入

```
N = 4
T = 2
m = {3, 2}
d = {{1, 2, 4}, {1, 3}}
```

输出

如果第1场联赛在第1天开始，第2场联赛在第2天开始的话，那么每天分别举办1，2，0，2轮比赛，总的幸福感为9。

首先，让我们先来推导期望值的计算公式。幸福感的定义是

$$\sum_{a=1}^{N}(\text{第 } a \text{ 天举办的轮数})^2$$

这样，幸福感的期望就是

$$E\left[\sum_{a=1}^{N}(\text{第}a\text{天举办的轮数})^2\right]=\sum_{a=1}^{N}E\left[(\text{第}a\text{天举办的轮数})^2\right]$$

于是每场联赛的开始日期都有从第1天到第N天共N种选择，一共就有N^T种，所以无法穷举所有方案。因此，为了高效地计算平方项的期望值，我们定义一个变量$X_{i,a}$。如果联赛i的某轮在第a天举办则为1，否则为0。就可以展开得到

$$E\left[(\text{第}a\text{天举办的轮数})^2\right]=E\left[(\sum_{i=1}^{T}X_{i,a})^2\right]$$

也许想把原式变形为

$$E\left[(\sum_{i=1}^{T}X_{i,a})^2\right]=(\sum_{i=1}^{T}E\left[X_{i,a}\right])^2$$

但要注意这个变形是不对的。因为虽然$i\neq j$时$X_{i,a}$和$X_{j,a}$是独立的，有$E[X_{i,a}X_{j,a}]=E[X_{i,a}]E[X_{j,a}]$成立，但$E[X_{i,a}{}^2]=E[X_{i,a}]E[X_{i,a}]$并不成立。我们单独计算这部分非独立的部分，就有

$$E\left[(\sum_{i=1}^{T}X_{i,a})^2\right]=E\left[\sum_{i=1}^{T}\sum_{j=1}^{T}X_{i,a}X_{j,a}\right]=\sum_{1\leqslant i,j\leqslant T,i\neq j}E\left[X_{i,a}\right]E\left[X_{j,a}\right]+\sum_{i=1}^{T}E\left[X_{i,a}^2\right]$$
$$=(\sum_{i=1}^{T}E\left[X_{i,a}\right])^2-\sum_{i=1}^{T}E\left[X_{i,a}\right]^2+\sum_{i=1}^{T}E\left[X_{i,a}^2\right]$$

又有这里 $X_{i,a}{}^2=X_{i,a}$，就得到

$$(\sum_{i=1}^{T}E\left[X_{i,a}\right])^2-\sum_{i=1}^{T}E\left[X_{i,a}\right]^2+\sum_{i=1}^{T}E\left[X_{i,a}\right]$$

像这样对式子进行变形之后，只要对每场联赛i独立地计算$E[X_{i,a}]$就可以求得全体的期望值了。又有

联赛i的某轮在第a天矩形 $\Leftrightarrow$ 存在j使得$s_i-1+d_{i,j}=a$

所以

$$E\left[X_{i,a}\right]=\frac{\left|\left\{j \mid d_{i,j} \leqslant a\right\}\right|}{N}$$

只要按顺序计算，就可以均摊$O(1)$地计算出每天的期望值。不过一共有N天，这样还是太大。但是注意到$d_{i,j}\leqslant 10000$的限制条件，对于$a>10000$总有$E[X_{i,a}]=E[X_{i,a-1}]$成立，于是只要对$1\leqslant a\leqslant 10000$计算就好了。另外，即便缺少了这个限制条件，也可以通过对概率保持不变的天一起求出期望值来进行优化，这样的区间个数为$O(Tm)$。

```
// 输入
int N, T;
int m[MAX_T];
int d[MAX_T][MAX_M];

// 期望值表（分母为N）
int E[MAX_T][MAX_D + 1];

void solve() {
  // 对每场联赛分别计算期望
  for (int i = 0; i < T; i++) {
    for (int j = 0; j < m[i]; j++) {
      E[i][d[i][j]]++;
    }
    for (int a = 1; a <= MAX_D; a++) {
      E[i][a] += E[i][a - 1];
    }
  }
  // 计算整体期望值
  // 以K+A/B的形式计算，计算过程中注意溢出问题
  long long K = 0, A = 0, B = (long long) N * N;
  for (int a = 1; a <= N && a <= MAX_D; a++) {
    long long sum = 0, sum2 = 0;
    for (int i = 0; i < T; i++) {
      sum += E[i][a];
      sum2 += E[i][a] * E[i][a];
    }
    if (a < MAX_D) {
      A += sum * sum - sum2 + sum * N;
      K += A / B;
      A %= B;
    } else {
      // 大于MAX_D的部分的期望值相同
      // 请注意直接计算可能溢出
      long long n = N - MAX_D + 1;
      K += sum * n / N;
      A += (sum * sum - sum2) * n + sum * n % N * N;
      K += A / B;
      A %= B;
      if (A < 0) {
        A += B;
        K--;
      }
```

```
    }
  }
  long long d = gcd(A, B);
  printf("%lld+%lld/%lld\n", K, A / d, B / d);
}
```

专栏 高精度运算

本题中的数据范围非常大，所以即便使用 long long 也有溢出的风险，需要注意。另外，偶尔也会遇到只有利用高精度运算才能求解的问题。遇到这类问题时，可以使用标准中支持高精度运算的 Java 等其他语言，或者预先准备好一些模块。

4.8.3 Football Team

Football Team （2009 Round 3 C）

N 名足球队员排成若干排拍照。给定每名队员的位置(x, y)，保证所有 x 均不相同。为了保证相片的美观，要让相邻的队员穿上不同颜色的衬衫。当位置为(x_1, y_1)的队员与位置为(x_2, y_2)的另一名队员满足如下条件时，我们称他们是相邻的。

- $y_1-1 \leqslant y_2 \leqslant y_1+1$
- 没有队员在满足 $x_1<x_3<x_2$ 的位置(x_3, y_2)上。

最少需要多少种不同颜色的衬衫？

限制条件

- $1 \leqslant x \leqslant 1000$

Small

- $1 \leqslant y \leqslant 15$
- $1 \leqslant N \leqslant 100$

Large

- $1 \leqslant y \leqslant 30$
- $1 \leqslant N \leqslant 1000$

样例 1

输入

```
N = 3
x = {10, 8, 12}
y = {10, 15, 7}
```

输出

```
1（所有人同样颜色）
```

样例 2

输入

```
N = 5
x = {1, 2, 3, 4, 5}
y = {1, 1, 1, 1, 1}
```

输出

```
1（第1、3、5个人和第2、4个人分别用不同颜色）
```

样例 3

输入

```
N = 3
x = {1, 2, 3}
y = {1, 2, 1}
```

输出

```
3（所有人的颜色互不相同）
```

首先把问题转化为图的模型，以队员为顶点，在应该穿不同颜色衬衫的队员之间连一条边。于是问题就转化成了图的着色问题。普通的着色问题是NP困难的，像本题这样规模的问题是无法求解的，所以这里的图应该隐含有一些特殊的性质。让我们来看看从题目的连边的条件推出的各种性质中哪些是有利于解题的。

1. 条件1：平面图

着色问题中有一个著名的四色定理，它指出任意平面图都可以用不超过四种颜色着色。实际上，本图就是平面图，适用这个定理。

假设实线的边发生交叉将产生矛盾（虚线处应该有边）

确定了图是平面图之后，根据四色定理，只要有至少四种颜色，就总能完成着色。色数为1的图

只有不含边的图，色数为2的图只有二分图。不过，只跟据平面图的性质，不能直接区分色数为3的图和色数为4的图，需要再寻找别的性质。

2. 条件2：内部面都是三角形

假设存在度数不小于4的内部面将产生矛盾（虚线处应该有边）

可以发现这里的图除了外部面外的所有面都是三角形。利用这一点可以高效地判断图的色数是否是3吗？首先，三角形的3个点应该要涂上不同的颜色。让我们来考虑两个三角形有一条公共边的情况。

有公共边时，确定了阴影三角形的着色也就唯一确定了其余三角形的着色

如上图所示，只要确定了一个三角形的着色，有公共边的三角形剩下的那个顶点的颜色也就唯一确定了。不断传递下去，所有通过边相连的三角形的颜色均可唯一确定。没有公共边，而只有公共点的情况又将如何呢？

有公共点时，确定了一个三角形的着色不能唯一确定其余点的着色

此时还有两个待着色的顶点，有两种着色方案，看似需要枚举两种可能。但是，由于所有的内部面都是三角形，删去公共点后，图就分成了两个独立部分，不论选择何种着色方案都是一样的。因此，要判断色数是否可能为3，只要考虑有公共边的三角形组成的子图就好了。

综上，我们能够高效地判断色数是否为3。

```
// 输入
int N;
int x[MAX_N], y[MAX_N];

bool g[MAX_N][MAX_N];     // 邻接矩阵
int color[MAX_N];         // 顶点的颜色
bool used[MAX_N][MAX_N]; // 边是否检查过的标记

// 判断色数是否为3
// 从确定2点v和u的颜色的状态开始，递归地给包含边v-u的三角形着色
bool rec(int v, int u) {
  used[v][u] = used[u][v] = true;
  int c = 3 - color[v] - color[u];  // 剩余的颜色
  for (int w = 0; w < N; w++) {
    if (g[v][w] && g[u][w]) {
      if (color[w] < 0) {
        color[w] = c;
        // 对三角形的其余两边递归处理
        if (!rec(v, w) || !rec(u, w)) {
          return false;
        }
      } else if (color[w] != c) {
        // 相邻顶点涂有相同颜色
        return false;
      }
    }
  }
  return true;
}

void solve() {
  // 建图
  for (int i = 0; i < N; i++) {
    int v[3] = {-1, -1, -1};
    for (int j = 0; j < N; j++) {
      if (x[i] < x[j]) {
        int k = y[j] - y[i] + 1;
        if (0 <= k && k < 3 && (v[k] < 0 || x[j] < x[v[k]])) {
          v[k] = j;
        }
      }
    }
    for (int k = 0; k < 3; k++) {
      if (v[k] >= 0) {
        g[i][v[k]] = g[v[k]][i] = true;
      }
    }
  }

  // 找出三角形并计算着色数
  int res = 1;

  // 虽然是三重循环，不过因为边数是O(N)，所以这部分的复杂度为O(N^2)
  for (int v = 0; v < N; v++) {
```

```
    for (int u = 0; u < N; u++) {
      if (g[v][u] && !used[v][u]) {
        // 如果有边则色数至少为2
        res = max(res, 2);
        for (int w = 0; w < N; w++) {
          if (g[v][w] && g[u][w]) {
            // 如果存在三角形，则着色数至少为3
            res = max(res, 3);
            memset(color, -1, sizeof(color));
            color[v] = 0;
            color[u] = 1;
            if (!rec(v, u)) {
              // 如果色数超过3则一定为4
              printf("4\n");
              return;
            }
            break;
          }
        }
      }
    }
  }

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

4.8.4 Endless Knight

Endless Knight （2008 Round3 D）

在 $H \times W$ 的棋盘上，马在移动过程中要保证 X 坐标和 Y 坐标同时增加。问从(1, 1)到(H, W)有多少不同的移动方案。输出 mod 10007 后的结果。此外，有 R 个格子被石头占据了，不能移动到这些格子上。

移动的方法

限制条件

- $0 \leqslant R \leqslant 10$

Small

- $1 \leqslant W \leqslant 100$
- $1 \leqslant H \leqslant 100$

Large

- $1 \leqslant W \leqslant 10^8$
- $1 \leqslant H \leqslant 10^8$

样例 1

输入

```
H = 4, W = 4, R = 1
石头的位置={2,1}
```

输出

```
2(1,1)->(2,3)->(4,4), (1,1)->(3,2)->(4,4)两种
```

样例 2

输入

```
H = 3, W = 3, R = 0
石头的位置{}
```

输出

```
0（无法从(1,1)移动到(3,3)，故0种）
```

样例 3

输入

```
H = 7, W = 10, R = 2
石头的位置={(1,2),(7,1)}
```

输出

```
5
```

解说

如果直接按马斜着移动处理有些不方便，所以我们对棋盘进行坐标变换，将其变换为水平和竖直方向的移动。

坐标变换

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} s + \begin{pmatrix} 2 \\ 1 \end{pmatrix} t = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} s \\ t \end{pmatrix}$$

$$\begin{pmatrix} s \\ t \end{pmatrix} = -\frac{1}{3} \begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

变换公式

如果变换后的坐标不是整数，那说明不存在满足条件的移动方案，答案为0。接下来，我们的讨论都是针对变换后的坐标的，依然记变换后的终点为(H, W)。首先考虑Small的解法。记走到(x, y)的方案数为$dp(x, y)$。那么走到$(x+1, y+1)$的方案只有先走到$(x, y+1)$再向下走一步，或先走到$(x+1, y)$再向右走一步这两种，所以$dp(x+1, y+1)=dp(x, y+1)+dp(x+1, y)$。当然，不允许移动到被石头占据的格子，所以对这些格子有$dp(x, y)=0$。于是，我们就能够在$O(H\times W)$的时间内计算得到$dp(H,W)$。

虽然对Small可以采用上述算法，但Large中的W和H都太大了，显然要用更为高效的解法。这里，注意到不能移动到的格子最多只有10个。记这些格子的集合为S，如果对任意的$S\supset T$，我们都能求得通过了所有属于T的格子的方案数的话，利用下面的容斥原理公式，就可以求得答案。

$$（不通过S移动到（H,W）的方案数）= \sum_{T\subset S}(-1)^{|T|}（通过T的所有格子移动到（H,W）的方案数）$$

考虑没有必须要通过的格子时的情况。此时从(0,0)移动到(H,W)的方案数就等于W个→和H个↓的排列的总数，也就是${}_{w+h}C_w$。

→和↓的排列与移动方案一一对应

当有若干个必须要通过的格子时，由于移动中x坐标和y坐标都是单调不减的，所以其通过的顺序是唯一的。也可能存在无法全部通过的情况，此时方案数显然为0。确定了通过顺序之后，那么两个相邻必须通过的格子之间就等价于没有必须通过的格子的情况，所以总的方案数就是至多R个${}_{w+h}C_w$形式的数的乘积。

就像4.1节所介绍的那样，通过预处理出$k!$ mod 10007的表，我们就能够在$O(\log n)$时间内计算组

合数。这样，本题能够在$O(2^R R \log(W+H))$时间内解决。

必须要通过×的情况

```
const int MOD = 10007;

typedef pair<int, int> P;

// 坐标变换为水平和竖直方向的移动
// 如果是无法移动到的点则返回false
bool normarize(int& x, int& y) {
  --x; --y;
  int xx = -x + 2 * y, yy = 2 * x - y;
  if (xx < 0 || yy < 0 || xx % 3 != 0 || yy % 3 != 0) return false;
  x = xx / 3; y = yy / 3;
  return true;
}

int count_bit(int a) {
  int res = 0;
  for (; a > 0; a >>= 1) res += a & 1;
  return res;
}


// 输入
int H, W, R;
P ps[MAX_R + 1];  // 石头的坐标

void solve() {
  int pn = 0;
  // 对石头进行坐标变换，预先将不可能到达的石头除去
  for (int i = 0; i < R; i++) {
    if (normarize(ps[i].first, ps[i].second)) {
      ps[pn++] = ps[i];
    }
  }

  // 如果不能移动到终点则答案是0
  ps[pn] = P(H, W);
  if (!normarize(ps[pn].first, ps[pn].second)) {
    printf("0\n");
    return ;
  }
  int res = 0;
  sort(ps, ps + pn);
```

```
    for (int i = 0; i < 1 << pn; i++) {
      int add = 1, prevx = 0, prevy = 0;
      for (int j = 0; j < pn + 1; j++) {
        if ((i >> j) % 2 == 1 || j == pn) {
          int mx = ps[j].first - prevx, my = ps[j].second - prevy;
          add = add * mod_comb(mx + my, mx, MOD) % MOD;
          prevx = ps[j].first;
          prevy = ps[j].second;
        }
      }
      if (count_bit(i) % 2 == 0) {
        res = (res + add) % MOD;
      } else {
        res = (res - add + MOD) % MOD;
      }
    }
    printf("%d\n", res);
  }
```

实际上，通过DP还可以将复杂度降到$O(R^2\log(W+H))$。有兴趣的读者可以思考一下。

4.8.5 The Year of Code Jam

The Year of Code Jam （2008 World Final E）

女孩 Sphinny 非常喜欢程序设计竞赛，她给今年的日历做了如下标记。

白：没有比赛的日子
蓝：参加比赛的日子
？：有比赛，但还在犹豫是否参加的日子

她的日历里有 N 个月，每个月各有 M 天。假定每一天都与其上个月的同一天、下个月的同一天、同个月的前一天、同个月的后一天是相邻的。现在她要决定每个？的日子是否参加比赛，来最大化她参加比赛所获得的幸福感。每参加一场比赛所获得的幸福感可以按照如下规则计算。

- 幸福感的初值为 4
- 每参加一场相邻日子里的比赛，幸福感减一

请计算她今年所能得到的最大幸福感。

限制条件

Small
- $1 \leq M, N \leq 15$

Large
- $1 \leq M, N \leq 50$

样例 1

输入

```
N = 3
M = 3
日历如下图（'.'代表白色，'#'代表蓝色）

.?.
.?.
.#.
```

输出

```
8（每月2日参加）
```

样例 2

输入

```
N = 5
M = 8

.#...##.
.##..?..
.###.#.#
??#..?..
###?#...
```

输出

```
42
```

本题是2008年的全球总决赛中通过人数最少的问题，不过利用目前已学的图论知识，只要稍加提示就能解出。首先考虑将日子表示为顶点，在相邻的日子之间连一条边建图。于是问题可以进行如下描述。

(1) 图中有三种顶点：蓝、白、？，？的顶点要变成蓝或白。
(2) 最大化(蓝点的个数)×4–(蓝点之间的边的条数)×2。

将图的顶点划分为两个集合并希望费用最小的问题，可以依据最大流最小割定理，通过最大流求解。这在以前的问题中已经实践过了。让我们来看一下本题应该如何转换。

1. 图中有三种顶点：蓝、白、?，？的顶点要变成蓝或白

假设得到的割中，和源点同侧的顶点是蓝色的，和汇点同侧的顶点是白色的。在以前的问题中，每个点属于任何一个集合都可以，但在本题中，其中一些点所在的集合一开始便已确定。不过，这只要通过将它属于另一个集合带来的费用设得足够大就能够简单处理了。

2. 最大化（蓝点的个数）×4 −（蓝点之间的边的条数）×2

因为我们要做的是最小化费用的问题，所以将之变形为最小化(白点的个数)×4+(蓝点之间的边的条数)×2。计算白点的费用虽然简单，但要计算蓝点之间的边的费用却不太容易。因为当两个顶点被分到不同的集合中时，它们之间所连的边的费用就包含在割中。所以，如果只有不同的集合间的边有费用的话，貌似可以转成最小化(白点的个数)×4–(白点和蓝点之间的边的个数)×2问题。

按照上面的方法变形后的图

但是，求包含负权边的图的最小割是NP困难的，尚未找到高效算法，所以这个方法也行不通。那么，该如何是好呢？因为只有连接两个蓝点的边上有费用，所以利用上原图还是一个二分图的性质，就能够很好地处理了。之前我们讨论的割，都是以和源点同侧的顶点为蓝色，和汇点同侧的顶点为白色的，而这里我们将二分图的其中一侧取反。这样，连接两个蓝点的边就属于割的一部分了。

按照上面的方法变形后的图

假设原图按二分图分为$U \cup V$，则我们按下面的规则给边赋权。

$u \in U$，u是蓝色的 $\rightarrow w(s, u) = \mathrm{INF}$
$u \in U$，u是白色的 $\rightarrow w(u, t) = \mathrm{INF}$
$u \in U$，u是？ $\rightarrow w(s, u) = 4$
$v \in V$，v是蓝色的 $\rightarrow w(v, t) = \mathrm{INF}$
$v \in V$，v是白色的 $\rightarrow w(s, v) = \mathrm{INF}$

$v \in V$，v是？ $\rightarrow w(v, t) = 4$

$u \in U$，$v \in V \rightarrow w(u, v) = 2$（反向的边则不加。因为只有蓝点之间的边有费用，而白点之间的边没有费用）

建好图之后，答案就是无视费用可能得到的最高分（=蓝与？的顶点个数×4）减去该图的最大s-t流流量。

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

// 输入
int N, M;
char cld[MAX_N][MAX_M + 1];  // 日历

void solve() {
  int res = 0;
  int s = N * M, t = s + 1;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      if ((i + j) % 2 == 0) {
        if (cld[i][j] == '#') {
          res += 4;
          add_edge(s, i * M + j, INF);
        } else if (cld[i][j] == '.') {
          add_edge(i * M + j, t, INF);
        } else {
          res += 4;
          add_edge(s, i * M + j, 4);
        }
        for (int k = 0; k < 4; k++) {
          int i2 = i + dx[k], j2 = j + dy[k];
          if (0 <= i2 && i2 < N && 0 <= j2 && j2 < M) {
            add_edge(i * M + j, i2 * M + j2, 2);
          }
        }
      } else {
        if (cld[i][j] == '#') {
          res += 4;
          add_edge(i * M + j, t, INF);
        } else if (cld[i][j] == '.') {
          add_edge(s, i * M + j, INF);
        } else {
          res += 4;
          add_edge(i * M + j, t, 4);
        }
      }
    }
  }
  res -= max_flow(s, t);
  printf("%d\n", res);
}
```

练 习 题

4.1 更加复杂的数学问题

■ 模运算的世界

POJ 1150: The Last Non-zero Digit
POJ 1284: Primitive Roots
POJ 2115: C Loooooops
POJ 3708: Recurrent Function
POJ 2720: Last Digits
GCJ Japan 2011 决赛 B：细菌繁殖

■ 矩阵

POJ 2345: Central heating
POJ 3532: Resistance
POJ 3526: The Teacher's Side of Math

■ 计数

POJ 2407: Relatives
POJ 1286: Necklace of Beads
POJ 2409: Let it Bead
AOJ 2164: Revenge of the Round Table
AOJ 2214: Warp Hall

4.2 找出游戏的必胜策略

■ 推理与动态规划算法

POJ 1082: Calendar Game
POJ 2068: Nim
POJ 3688: Cheat in the Game
POJ 1740: A New Stone Game

■ Nim与Grundy数

POJ 2975: Nim
POJ 3537: Crosses and Crosses
Codeforces 138D: World of Darkraft
POJ 2315: Football Game

4.3 成为图论大师之路

■ 强连通分量分解

POJ 3180: The Cow Prom
POJ 1236: Network of Schools

■ 2-SAT

POJ 3678: Katu Puzzle
POJ 2723: Get Luffy Out
POJ 2749: Building roads

■ LCA

POJ 1986: Distance Queries
POJ 3728: The merchant

4.4 常用技巧精选（二）

■ 栈

POJ 3250: Bad Hair Day
POJ 2082: Terrible Sets
POJ 3494: Largest Submatrix of All 1's

■ 双端队列

POJ 2823: Sliding Window
POJ 3260: The Fewest Coins
POJ 1180: Batch Scheduling
AOJ 1070: FIMO sequence

4.5 开动脑筋智慧搜索

■ 剪枝

POJ 1011: Sticks
POJ 2046: Gap
POJ 3134: Power Calculus

■ A*与IDA*

POJ 3523: The Morning after Halloween
POJ 2032: Square Carpets
UVA 10181: 15-Puzzle Problem

4.6 划分、解决、合并：分治法

■ 数列上的分治法

POJ 1854: Evil Straw Warts Live

■ 平面上的分治法

GCJ 2009 World Finals B: Min Perimeter
Codeforces 97B: Superset

■ 树上的分治法

POJ 2114: Boatherds
UVa 12161: Ironman Race in Treeland
SPOJ QTREE5: Query on a tree V

4.7 华丽地处理字符串

■ 动态规划算法

AOJ 2212: Stolen Jewel
Codeforces 86C: Genetic Engineering

■ 字符串匹配

Codeforces 25E: Test
AOJ 1312: Where's Wally

■ 后缀数组

POJ 1509: Glass Beads
POJ 3415: Common Substrings
POJ 3729: Facer's string
AOJ 2292: Common Palindromes
Codeforces 123D: String

本书中未涉及的拓展主题

图论、组合优化

■ 欧拉回路

欧拉回路指的是经过所有边恰好一次的回路。其存在性的判定和解的构造都可以在线性时间完成。

■ 最小树形图

最小树形图是推广到有向图上的最小生成树，目标是寻找一个边权和最小的生成子图，使得从给定的节点出发沿生成子图的边可以到达所有节点。存在通过反复强连通分量分解求解的$O(VE)$时间的算法。

■ 斯坦纳树

斯坦纳树问题是最小生成树问题的一般化，目标是寻找一个边权和最小的子图，使得指定的顶点子集连通，不过所构造子图中可以包含不属于给定顶点子集的顶点。该问题是NP-Hard问题，不过利用状态压缩DP或容斥原理，可以找到给定顶点个数的指数时间的算法。

■ 割点、割边

割点和割边分别指的是，将其删除后将导致图的连通分量个数增加的顶点和边。它们可以通过深度优先搜索在$O(E)$时间内求得。

■ 全局最小割

全局最小割指的是，为了破坏图的连通性，所需删除的权值和最小的边集。虽然可以通过固定源点s、枚举汇点t求s–t割的最小值求得，但是使用更为高效的Stoer-Wagner算法可以在$O(V^3)$时间内求得。

■ 单纯形法

单纯形法是通过不断进行诸如高斯消元法中所用的转轴操作，求解线性规划问题的算法。虽然该算法在某些刻意构造的数据上需要花费指数时间，但是它通常只需要约束条件个数的常数倍次迭代即可求得结果，非常实用。

■ **拟阵**

拟阵是对向量空间中的线性无关的一般化，与贪心算法关系密切。例如，求解最小生成树问题可以看作求解拟阵的最大权独立集问题，从而可以自然而然地推出Kruskal算法。又如，求解二分图最大匹配和最小树形图可以看作是求解两种拟阵的公共独立集，这称为拟阵交问题。拟阵交能够在多项式时间内求解，在求解多个互不相交的生成树等许多问题中会用到。

数值计算

■ **三分搜索、黄金分割搜索**

它们是计算拟凸函数的最大值和最小值的简单方法。与三分搜索相比，黄金分割搜索可以重复利用上一步的计算结果，减小计算量。

■ **Karatsuba法、快速傅里叶变换（FFT）**

高精度整数和多项式的乘法运算都是卷积运算。朴素的计算方法需要$O(N^2)$时间，而利用Karatsuba算法只需$O(N^{1.59})$时间，利用快速傅里叶变换则能够在$O(N\log N)$时间完成。

数论

■ **离散对数**

离散对数问题指的是，给定整数a, b, m，要求满足$a^x \equiv b \pmod m$的最小非负整数x。利用Baby-step Giant-step算法能够在$O(\sqrt{m})$时间内求得。取k为$\sqrt{m}$附近的整数，假设解$x=yk+z(0\leqslant z<k)$，如果我们预先计算好a^z的表，那么枚举y的时候就可以通过查表判断是否有对应的z。

■ **Stern-Brocot树**

Stern-Brocot树的每一个顶点代表一个有理数区间，可用于给出无理数的既约分数近似。它的根是(0/1, 1/0)，而顶点$(a/b, c/d)$有两个儿子$(a/b, (a+c)/(b+d))$和$((a+c)/(b+d), c/d)$，所有区间的端点都是既约分数。

搜索

■ **α-β剪枝**

α-β剪枝是将博弈类游戏最优策略的搜索进行有效优化的一种技术。游戏双发的目标都是令自己的得分尽量大，所以如果发现当前可能获得的分数比已经找到的最优解要差的时候，就可以剪枝。利用该剪枝，通常可以将搜索的节点数降到直接搜索的平方根规模。

数据结构

■ 平衡二叉树

平衡二叉树是不论进行何种操作，总能通过旋转调整等保持平衡的二叉搜索树。有时为了满足某些特殊需求，需要自己实现。在程序设计竞赛中，常用的有实现起来相对轻松的Treap、Splay和Scapegoat等。

■ 左偏树、斜堆

左偏树和斜堆都支持高效合并操作、且实现起来比较简单的堆。

■ 树链剖分

树链剖分是处理树上查询的有效方法。对于一颗给定的树，它可以将路径作为节点构造一棵高度不超过$O(\log n)$的新树，从而实现高效处理。

字符串算法

■ KMP算法、Aho-Corasick算法

二者都是模式匹配算法。KMP用于完成单串匹配；而Aho-Corasick能够在线性时间内完成多串匹配，其原理是通过预处理将模式串转成自动机。在程序设计竞赛中，常用到这类自动机。

■ Manacher算法

Manacher算法是寻找回文子串的算法。它可以在线性时间内求出以各个位置为中心的最长回文的长度。

■ 语法解析

程序设计竞赛中也经常出现要求解析满足给定语法规则的字符串的题目。相关方法有基于动态规划的CYK算法、按最低优先级的运算符分割处理的方法、构建递归下降语法解析器的方法等。

书中例题列表

抽签 2
三角形 16
Ants（POJ 1852） 18
部分和问题 30
Lake Counting（POJ 2386） 32
迷宫的最短路径 34
硬币问题 39
区间调度问题 40
Best Cow Line（POJ 3617） 43
Saruman's Army（POJ 3069） 45
Fence Repair（POJ 3253） 47
01背包问题 51
最长公共子序列问题 56
完全背包问题 57
01背包问题之2 60
多重部分和问题 62
最长上升子序列问题 64
划分数 66
多重集组合数 68
Expedition（POJ 2431） 74
食物链（POJ 1182） 88
二分图判定 97
Roadblocks（POJ 3255） 108
Conscription（POJ 3723） 109
Layout（POJ 3169） 110
线段上格点的个数 113
双六 115
素数判定 117
素数的个数 118
区间内素数的个数 120
Carmichael Numbers（UVa 10006） 122
Minimum Scalar Product（2008 Round 1A A） 125
Crazy Rows（2009 Round 2 A） 127
Bribe the Prisoners（2009 Round 1C C） 129
Millionaire（2008 APAC local onsites C） 132
lower_bound 138
Cable master（POJNO. 1064） 140
Aggressive cows（POJNO. 2456） 142
最大化平均值 143
Subsequence（POJNO. 3061） 146
Jessica's Reading Problem（POJNO. 3320） 149
Face The Right Way（POJNO. 3276） 150
Fliptile（POJNO. 3279） 153
Physics Experiment（POJNO. 3684） 158
4 Values whose Sum is 0（POJNO. 2785） 160
超大背包问题 162
区域的个数 164
Crane（POJ 2991） 170
冒泡排序的交换次数 178
A Simple Problem with Integers（POJ 3468） 179
K-th Number（POJ 2104） 185
旅行商问题 191
Travelling by Stagecoach（POJNO. 2686） 193
铺砖问题 196
斐波那契数列 199
Blocks（POJ 3734） 202
图中长度为k的路径的计数 203
Matrix Power Series（POJ 3233） 204
Minimizing Maximizer（POJ 1769） 206
最大传输量 209
指派问题 217
结对子 220
最小传输费用 222
Asteroids（POJNO. 3041） 228
Evacuation（POJNO. 3057） 230
Dining（POJNO. 3281） 234
Dual Core CPU（POJNO. 3469） 236
Fram Tour（POJNO. 2135） 238
Evacuation Plan（POJNO. 2175） 240
The Windy's（POJNO. 3686） 243

Intervals（POJNO. 3680）.....246
Jack Straws（POJ 1127）.....250
White Bird（AOJ 2308）.....255
Coneology（POJ 2932）.....258
Beauty Contest（POJ 2187）.....260
Intersection of Two Prisms（AOJ 1313）.....263
Numbers（2008 Round 1A C）.....267
No Cheating（2008 Round 3 C）.....269
Stock Charts（2009 Round 2 C）.....271
Watering Plants（2009 Round 2 D）.....273
Number Sets（2008 Round 1B B）.....278
Wi-fi Towers（2009 World Final D）.....280
Random Walk.....288
容斥原理.....295
没有周期性的字符串的计数.....297
石头染色方案计数问题.....301
硬币游戏1.....305
A Funny Game（POJ 2484）.....307
Euclid's Game（POJ 2348）.....309
Nim.....311
Georgia and Bob（POJ 1704）.....312
硬币游戏2.....315
Cutting Game（POJ 2311）.....318
Popular Cows（POJ 2186）.....322
Priest John's Busiest Day（POJ 3683）.....326
Housewife Wind（POJ 2763）.....331
Largest Rectangle in a Histogram（POJ 2559）.....335
滑动最小值.....337
多重背包问题.....339
K-Anonymous Sequence（POJ 3709）.....341
观看计划.....345
数独（POJ 2676, 2918, 3074, 3076）.....350
Square Destroyer（POJ 1084）.....352
Tree（POJ 1741）.....361
最近点对问题（UVa 10245）.....364
禁止字符串.....368
DNA Repair（POJ 3691）.....370
星座（POJ 3690）.....375
Sequence（POJ 3581）.....380
最长公共子串（POJ 2217）.....383
最长回文子串.....385
Mine Layer（2008 World Final C）.....387
Year of More Code Jam（2009 World Final A）.....392
Football Team（2009 Round 3 C）.....395
Endless Knight（2009 Round 3 D）.....399
The Year of Code Jam（2008 World Final E）.....403

参考文献

「アルゴリズムC」近代科学社
英文版名为 *Algorithms in C*
中文版名为《算法：C语言实现》（机械工业出版社）

「アルゴリズムC++」近代科学社
英文版名为 *Algorithms in C++*

「アルゴリズムイントロダクション」近代科学社
英文版名为 *Introduction To Algorithms*
中文版名为《算法导论》（有多个版本的中文译本）
该书是算法的标准教材。通过它可以学习到排序、搜索、字符串、计算几何、图论和数学等众多领域的基础算法。

「アルゴリズムデザイン」共立出版
英文版名为 *Algorithms Design*
中文版名为《算法设计》（清华大学出版社）
与《算法：C语言实现》和《算法导论》不同，该书着重介绍的不是算法，而是算法的设计方法，即如何推出正确的算法。

「組合せ最適化—理論とアルゴリズム」シュプリンガージャパン
英文版名为 *Combinatorial Optimization Theory and Algorithms*
该书涵盖了图论及离散数学相关的算法。

「コンピュータの数学」共立出版
英文版名为 *Concrete Mathematics: A Foundation for Computer Science*
中文版名为《具体数学：计算机科学基础》（有两个版本的中文译本）
该书介绍了求和、数论、数列和离散概率等数学知识。

「コンピュータ・ジオメトリ—計算幾何学：アルゴリズムと応用」近代科学社
英文版名为 *Computational Geometry: Algorithms and Applications*
该书涵盖了计算几何相关的数据结构与算法。

「目指せ！プログラミング世界一」近代科学社
《目标：程序设计世界冠军!》近代科学社
该书由ACM-ICPC日本赛区的主办方组织编写。包含ICPC的详细介绍和题目解析。