```
    while (orig[v] != a) {
      parent[v] = u;
      u = match[v];
      if (label[u] == 1) {  // 初始点设为「o」找增广路
        label[u] = 0;
        q.push(u);
      }
      orig[v] = orig[u] = a;  // 缩花
      v = parent[u];
    }
  };  // blossom

  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v];
      int next_v = match[pv];
      match[v] = pv;
      match[pv] = v;
      v = next_v;
    }
  };  // augment

  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    while (!q.empty()) {
      q.pop();
    }
    q.push(root);
    // 初始点设为「o」，这里以「0」代替「o」,「1」代替「i」
    label[root] = 0;
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int id : g.g[v]) {
        auto &e = g.edges[id];
        int u = e.from ^ e.to ^ v;
        if (label[u] == -1) {  // 找到未拜访点
          label[u] = 1;        // 标记「i」
          parent[u] = v;
          if (match[u] == -1) {  // 找到未匹配点
            augment(u);          // 寻找增广路径
            return true;
          }
          // 找到已匹配点 将与她匹配的点丢入queue 延伸交错树
          label[match[u]] = 0;
          q.push(match[u]);
          continue;
        } else if (label[u] == 0 && orig[v] != orig[u]) {
          // 找到已拜访点 且标记同为「o」代表找到「花」
          int a = lca(orig[v], orig[u]);
          // 找LCA 然后缩花
```

```
123           blossom(u, v, a);
124           blossom(v, u, a);
125         }
126       }
127     }
128     return false;
129   };  // bfs
130
131   auto greedy = [&]() {
132     vector<int> order(g.n);
133     // 随机打乱 order
134     iota(order.begin(), order.end(), 0);
135     shuffle(order.begin(), order.end(), rng);
136
137     // 将可以匹配的点匹配
138     for (int i : order) {
139       if (match[i] == -1) {
140         for (auto id : g.g[i]) {
141           auto &e = g.edges[id];
142           int to = e.from ^ e.to ^ i;
143           if (match[to] == -1) {
144             match[i] = to;
145             match[to] = i;
146             break;
147           }
148         }
149       }
150     }
151   };  // greedy
152
153   // 一开始先随机匹配
154   greedy();
155   // 对未匹配点找增广路
156   for (int i = 0; i < g.n; i++) {
157     if (match[i] == -1) {
158       bfs(i);
159     }
160   }
161   return match;
162 }
```

UOJ #79. 一般图最大匹配[1]

```
1 #include <algorithm>
2 #include <cassert>
3 #include <iostream>
4 #include <numeric>
5 #include <queue>
6 #include <random>
7 #include <vector>
8 using namespace std;
```

```
 9
10 // graph
11 template <typename T>
12 class graph {
13  public:
14   struct edge {
15     int from;
16     int to;
17     T cost;
18   };
19
20   vector<edge> edges;
21   vector<vector<int>> g;
22   int n;
23
24   graph(int _n) : n(_n) { g.resize(n); }
25
26   virtual int add(int from, int to, T cost) = 0;
27 };
28
29 // undirectedgraph
30 template <typename T>
31 class undirectedgraph : public graph<T> {
32  public:
33   using graph<T>::edges;
34   using graph<T>::g;
35   using graph<T>::n;
36
37   undirectedgraph(int _n) : graph<T>(_n) {}
38
39   int add(int from, int to, T cost = 1) {
40     assert(0 <= from && from < n && 0 <= to && to < n);
41     int id = (int)edges.size();
42     g[from].push_back(id);
43     g[to].push_back(id);
44     edges.push_back({from, to, cost});
45     return id;
46   }
47 };
48
49 // blossom / find_max_unweighted_matching
50 template <typename T>
51 vector<int> find_max_unweighted_matching(const undirectedgraph<T> &g) {
52   std::mt19937 rng(114514);  // 这里随机种子是无关紧要的
53   // 也可以用 chrono::steady_clock::now().time_since_epoch().count()
54   // 获取当前时间
55   vector<int> match(g.n, -1);   // 匹配
56   vector<int> aux(g.n, -1);     // 时间戳记
57   vector<int> label(g.n);       // "o" or "i"
58   vector<int> orig(g.n);        // 花根
59   vector<int> parent(g.n, -1);  // 父节点
60   queue<int> q;
```

```
  int aux_time = -1;

  auto lca = [&](int v, int u) {
    aux_time++;
    while (true) {
      if (v != -1) {
        if (aux[v] == aux_time) {  // 找到拜访过的点 也就是LCA
          return v;
        }
        aux[v] = aux_time;
        if (match[v] == -1) {
          v = -1;
        } else {
          v = orig[parent[match[v]]];  // 以匹配点的父节点继续寻找
        }
      }
      swap(v, u);
    }
  };  // lca

  auto blossom = [&](int v, int u, int a) {
    while (orig[v] != a) {
      parent[v] = u;
      u = match[v];
      if (label[u] == 1) {  // 初始点设为"o" 找增广路
        label[u] = 0;
        q.push(u);
      }
      orig[v] = orig[u] = a;  // 缩花
      v = parent[u];
    }
  };  // blossom

  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v];
      int next_v = match[pv];
      match[v] = pv;
      match[pv] = v;
      v = next_v;
    }
  };  // augment

  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    while (!q.empty()) {
      q.pop();
    }
    q.push(root);
    // 初始点设为 "o", 这里以"0"代替"o", "1"代替"i"
    label[root] = 0;
```

```
113     while (!q.empty()) {
114       int v = q.front();
115       q.pop();
116       for (int id : g.g[v]) {
117         auto &e = g.edges[id];
118         int u = e.from ^ e.to ^ v;
119         if (label[u] == -1) {  // 找到未拜访点
120           label[u] = 1;        // 标记 "i"
121           parent[u] = v;
122           if (match[u] == -1) {  // 找到未匹配点
123             augment(u);          // 寻找增广路径
124             return true;
125           }
126           // 找到已匹配点 将与她匹配的点丢入queue 延伸交错树
127           label[match[u]] = 0;
128           q.push(match[u]);
129           continue;
130         } else if (label[u] == 0 && orig[v] != orig[u]) {
131           // 找到已拜访点 且标记同为"o" 代表找到"花"
132           int a = lca(orig[v], orig[u]);
133           // 找LCA 然后缩花
134           blossom(u, v, a);
135           blossom(v, u, a);
136         }
137       }
138     }
139     return false;
140   };  // bfs
141
142   auto greedy = [&]() {
143     vector<int> order(g.n);
144     // 随机打乱 order
145     iota(order.begin(), order.end(), 0);
146     shuffle(order.begin(), order.end(), rng);
147
148     // 将可以匹配的点匹配
149     for (int i : order) {
150       if (match[i] == -1) {
151         for (auto id : g.g[i]) {
152           auto &e = g.edges[id];
153           int to = e.from ^ e.to ^ i;
154           if (match[to] == -1) {
155             match[i] = to;
156             match[to] = i;
157             break;
158           }
159         }
160       }
161     }
162   };  // greedy
163
164   // 一开始先随机匹配
```

```
  greedy();
  // 对未匹配点找增广路
  for (int i = 0; i < g.n; i++) {
    if (match[i] == -1) {
      bfs(i);
    }
  }
  return match;
}

int main() {
  ios::sync_with_stdio(false);

  int n, m;
  cin >> n >> m;

  undirectedgraph<int> g(n);

  while (m--) {
    int u, v;
    cin >> u >> v;
    g.add(u - 1, v - 1);  // 0-based
  }

  auto match = find_max_unweighted_matching(g);

  cout << count_if(match.begin(), match.end(), [](int x) { return x !=
-1; }) /
              2
       << endl;
  for (int i = 0; i < n; i++) cout << match[i] + 1 << " \n"[i == n - 1];

  return 0;
}
```

基于高斯消元的一般图匹配算法

提示

在阅读以下内容前，你可能需要先阅读「线性代数」部分中关于矩阵的内容：

- 矩阵→9.14.4@p1716
- 行列式→9.14.6@p1729
- 高斯消元→9.19.3@p1944

这一部分将介绍一种基于高斯消元的一般图匹配算法. 与传统的带花树算法相比, 它的优势在于更易于理解与编写, 同时便于解决「最大匹配中的必须点」等问题; 缺点在于常数比较大, 因为高斯消元的 $O(n^3)$ 基本是跑满的，而带花树一般跑不满.

前置知识：Tutte 矩阵

定义：对于一张 n 个点的无向图 $G=(V,E)$，其 Tutte 矩阵 $\tilde{A}(G)$ 为一个 $n\times n$ 的矩阵，其中：

$$\tilde{A}(G)_{i,j}=\begin{cases}x_{i,j}, & i<j,(v_i,v_j)\in E\\ -x_{i,j}, & i>j,(v_i,v_j)\in E\\ 0, & \text{otherwise}\end{cases}$$

其中 $x_{i,j}$ 是一个变量，因此 $\tilde{A}(G)$ 中共有 $|E|$ 个变量.

在无歧义的情况下，以下将 $\tilde{A}(G)$ 简写为 $\tilde{A}$.

定理（Tutte 定理）：G 存在完美匹配当且仅当 $\det\tilde{A}\neq 0$.

证明

这里引入「偶环覆盖」的概念：一个无向图 G 的偶环覆盖指用若干偶环（包括二元环）不重不漏地覆盖所有的点.

易证 G 存在完美匹配当且仅当 G 存在偶环覆盖.

- 如果 G 存在偶环覆盖，我们只需要在每个环都隔一条取一条边，就可以得到一个完美匹配.
- 如果 G 存在完美匹配，我们只需要将匹配边对应的二元环取出，就可以得到一个偶环覆盖.

然后证明 G 存在偶环覆盖当且仅当 $\tilde{A}\neq 0$.

考虑行列式的定义

$$\det A=\sum_{\pi}(-1)^{\pi}\prod_i A_{i,\pi_i}$$

其中 π 是任意排列，$(-1)^{\pi}$ 表示若 π 中的逆序对数为奇数，则取 -1，否则取 1.

不难看出每个排列都可以被看作 G 的一个环覆盖. 如果这个环覆盖中存在奇环，则将这个环翻转后的和一定为 0，因此只有偶环覆盖才能使行列式不为 0，证毕.

定理：$\operatorname{rank}\tilde{A}$ 一定为偶数，并且 G 的最大匹配的大小等于 $\operatorname{rank}\tilde{A}$ 的一半.

证明

反对称矩阵的秩只能是偶数；后者请读者自行思考.

实际应用中不可能带着 $|E|$ 个变量进行计算，不过可以取一个数域，例如取某个素数 p 的剩余系 $\mathcal{Z}_p$，将变量分别随机替换为 $\mathcal{Z}_p$ 中的数，再进行计算. 方便起见，在无歧义的情况下，以下用 $\tilde{A}$ 直接指代替换后的矩阵.

定理：$\operatorname{rank}\tilde{A}$ 至多为 G 的最大匹配大小的两倍，并且二者相等的概率至少为 $1-\frac{n}{p}$.

考虑到一般图最大匹配中 n 基本不会超过 10^3，实际中 p 取 10^9 数量级的素数就足够了.

由定理可知，如果只需要求最大匹配数，而无需匹配方案，那么只需要用一次高斯消元求出 $\operatorname{rank}\tilde{A}$ 即可，远比带花树简洁. 不过如果需要输出方案，会稍微复杂一些，需要用到下面介绍的算法.

构造完美匹配

由 Tutte 定理和上面的定理可知，如果 G 存在完美匹配，那么 $\tilde{A}$ 有很大概率满秩. 方便起见，以下叙述中均省略「有很大概率」.

记 G 中标号为 i 的点为 v_i，进一步地我们有如下定理：

定理：$\tilde{A}^{-1}_{j,i}\neq 0\iff G-\{v_i,v_j\}$ 有完美匹配.

逆矩阵与伴随矩阵

对任意 n 阶方阵 A，定义其伴随矩阵为 $A^*_{i,j}=(-1)^{i+j}M_{j,i}$，其中 $M_{j,i}$ 为删去第 j 行第 i 列的余子式. 换言之，设 A 的代数余子式矩阵为 M，则 $A^*=M^T$.

> **定理**：如果 A 可逆，那么 $A^{-1}=\frac{1}{\det A}A^*$.
>
> 所以这里的 $A^{-1}_{j,i}\neq 0\iff M_{i,j}\neq 0$，也就是 A 删去第 i 行第 j 列后的部分满秩.

换言之，如果 $(v_i,v_j)\in E$，并且 $\tilde{A}^{-1}_{j,i}\neq 0$，就表明存在一个完美匹配方案包含 (v_i,v_j) 这条边. 以下将这种边称为 **可行边**.

由如上定理，对于一个有完美匹配的无向图 G，我们可以得到一个比较显然的暴力算法来寻找一组完美匹配：每次枚举 i,j，如果 (v_i,v_j) 是一条可行边（连边存在，并且 $\tilde{A}^{-1}_{j,i}\neq 0$），就将 (v_i,v_j) 加入匹配方案，并在 G 中都删掉这两个点，再重新计算新的 $\tilde{A}^{-1}$.

总共要做 $\frac{n}{2}$ 轮，每轮都是 $O(n^3)$ 的，总的复杂度是 $O(n^4)$，有点慢了. 实际上我们在重新计算 $\tilde{A}^{-1}$ 时，不必每次都重新用高斯消元求逆矩阵，而是可以利用如下定理：

定理（消去定理）：令

$$A=\begin{bmatrix}a_{1,1} & v^T\\ u & B\end{bmatrix}\quad A^{-1}=\begin{bmatrix}\hat{a}^{1,1} & \hat{v}^T\\ \hat{u} & \hat{B}\end{bmatrix}$$

并且 $\hat{a}_{1,1}\neq 0$，那么就有

$$B^{-1}=\hat{B}-\frac{\hat{u}\hat{v}^T}{\hat{a}_{1,1}}$$

定理中描述的是消去第一行第一列的情况. 实际上，它可以非常显然地推广到消去任意一行一列的情况，因此我们只需在算法最开始计算一次 $\tilde{A}^{-1}$，后面每次删除两个点时，只需执行两次 $O(n^2)$ 的消去过程即可.

描述有些抽象，可以参考 C++ 代码

```cpp
void eliminate(int A[][MAXN], int r, int c) {  // 消去第 r 行第 c 列
  row_marked[r] = col_marked[c] = true;        // 已经被消掉

  int inv = quick_power(A[r][c], p - 2);  // 逆元

  for (int i = 1; i <= n; i++)
    if (!row_marked[i] && A[i][c]) {
      int tmp = (long long)A[i][c] * inv % p;

      for (int j = 1; j <= n; j++)
        if (!col_marked[j] && A[r][j])
          A[i][j] = (A[i][j] - (long long)tmp * A[r][j]) % p;
    }
}
```

总共要做 $\frac{n}{2}$ 轮，每轮复杂度为 $O(n^2)$，因此上述算法可以在 $O(n^3)$ 的时间内找到一组完美匹配.

构造最大匹配

我们刚刚已经解决了构造一组完美匹配的问题，但是求解问题时一般需要最大匹配.

前面已经提到，G 的最大匹配大小等于 $\operatorname{rank}\tilde{A}$ 的一半. 如果我们能找到 $\tilde{A}$ 的一个最大满秩子方阵，那么对子方阵对应的导出子图求出一组完美匹配，即可找到 G 的一组最大匹配.

换一个角度考虑，如果 G 有完美匹配，那么 $\tilde{A}$ 满秩，换言之，$\tilde{A}$ 是线性无关的. 那么如果 $\tilde{A}$ 不是满秩的，我们可以求出 $\tilde{A}$ 的一组线性基，然后只保留线性基对应的行列，就可以得到 $\tilde{A}$ 的一个最大满秩子方阵.

求出最大满秩子方阵之后，再用上面的算法找出导出子图的一组完美匹配，即可得到原图的一组最大匹配．注意由于高斯消元中可能会有行的交换，因此实现时要注意维护好点的编号．

UOJ #79. 一般图最大匹配[2]

```cpp
#include <cstring>
#include <iostream>
#include <random>
#include <utility>
using namespace std;

constexpr int MAXN = 505, p = (int)1e9 + 7;

int qpow(int a, int b) {
  int ans = 1;
  while (b) {
    if (b & 1) ans = (long long)ans * a % p;
    a = (long long)a * a % p;
    b >>= 1;
  }
  return ans;
}

int A[MAXN][MAXN], B[MAXN][MAXN], t[MAXN][MAXN], id[MAXN];

// 高斯消元 O(n^3)
// 在传入 B 时表示计算逆矩阵，传入 nullptr 则只需计算矩阵的秩
void Gauss(int A[][MAXN], int B[][MAXN], int n) {
  if (B) {
    memset(B, 0, sizeof(t));
    for (int i = 1; i <= n; i++) B[i][i] = 1;
  }

  for (int i = 1; i <= n; i++) {
    if (!A[i][i]) {
      for (int j = i + 1; j <= n; j++)
        if (A[j][i]) {
          swap(id[i], id[j]);
          for (int k = i; k <= n; k++) swap(A[i][k], A[j][k]);

          if (B)
            for (int k = 1; k <= n; k++) swap(B[i][k], B[j][k]);
          break;
        }

      if (!A[i][i]) continue;
    }

    int inv = qpow(A[i][i], p - 2);

    for (int j = 1; j <= n; j++)
      if (i != j && A[j][i]) {
        int t = (long long)A[j][i] * inv % p;
```

```
49
50          for (int k = i; k <= n; k++)
51            if (A[i][k]) A[j][k] = (A[j][k] - (long long)t * A[i][k]) % p;
52
53          if (B) {
54            for (int k = 1; k <= n; k++)
55              if (B[i][k]) B[j][k] = (B[j][k] - (long long)t * B[i][k]) % p;
56          }
57        }
58    }
59
60    if (B)
61      for (int i = 1; i <= n; i++) {
62        int inv = qpow(A[i][i], p - 2);
63
64        for (int j = 1; j <= n; j++)
65          if (B[i][j]) B[i][j] = (long long)B[i][j] * inv % p;
66      }
67  }
68
69  bool row_marked[MAXN] = {false}, col_marked[MAXN] = {false};
70
71  int sub_n;  // 极大满秩子矩阵的大小
72
73  // 消去一行一列 O(n^2)
74  void eliminate(int r, int c) {
75    row_marked[r] = col_marked[c] = true;  // 已经被消掉
76
77    int inv = qpow(B[r][c], p - 2);
78
79    for (int i = 1; i <= sub_n; i++)
80      if (!row_marked[i] && B[i][c]) {
81        int t = (long long)B[i][c] * inv % p;
82
83        for (int j = 1; j <= sub_n; j++)
84          if (!col_marked[j] && B[r][j])
85            B[i][j] = (B[i][j] - (long long)t * B[r][j]) % p;
86      }
87  }
88
89  int vertices[MAXN], girl[MAXN];  // girl 是匹配点，用来输出方案
90
91  int main() {
92    cin.tie(nullptr)->sync_with_stdio(false);
93    auto rng = mt19937(random_device{}());
94
95    int n, m;
96    cin >> n >> m;  // 点数和边数
97
98    while (m--) {
99      int x, y;
100     cin >> x >> y;
```

```
    A[x][y] = rng() % p;
    A[y][x] = -A[x][y];  // Tutte 矩阵
  }

  for (int i = 1; i <= n; i++)
    id[i] = i;  // 输出方案用的，因为高斯消元的时候会交换列
  memcpy(t, A, sizeof(t));

  Gauss(A, nullptr, n);

  for (int i = 1; i <= n; i++)
    if (A[id[i]][id[i]]) vertices[++sub_n] = i;  // 找出一个极大满秩子矩阵

  for (int i = 1; i <= sub_n; i++)
    for (int j = 1; j <= sub_n; j++) A[i][j] = t[vertices[i]][vertices[j]];

  Gauss(A, B, sub_n);

  for (int i = 1; i <= sub_n; i++)
    if (!girl[vertices[i]])
      for (int j = i + 1; j <= sub_n; j++)
        if (!girl[vertices[j]] && t[vertices[i]][vertices[j]] && B[j][i]) {
          // 注意上面那句 if 的写法，现在 t 是邻接矩阵的备份，
          // 逆矩阵 j 行 i 列不为 0 当且仅当这条边可行
          girl[vertices[i]] = vertices[j];
          girl[vertices[j]] = vertices[i];

          eliminate(i, j);
          eliminate(j, i);
          break;
        }

  cout << sub_n / 2 << '\n';
  for (int i = 1; i <= n; i++) cout << girl[i] << ' ';

  return 0;
}
```

习题

- UOJ #79. 一般图最大匹配[3]
- UOJ#171.【WC2016】挑战 NPC[4]

参考资料

1. Mucha M, Sankowski P.Maximum matchings via Gaussian elimination[5]
2. 周子鑫，杨家齐《基于线性代数的一般图匹配》
3. ZYQN 《基于线性代数的一般图匹配算法》[6]

外部参考资料

[1] UOJ #79. 一般图最大匹配 [2] UOJ #79. 一般图最大匹配 [3] UOJ #79. 一般图最大匹配

[4] UOJ#171.【WC2016】挑战NPC  [5] Maximum matchings via Gaussian elimination  [6] 《基于线性代数的一般图匹配算法》 

11.24.5 一般图最大权匹配

Authors: accelsao, Henry-ZHR, yuhuoji

本页从一般图最大权完美匹配到一般图最大权匹配（最大权匹配可以通过增加零边变成最大权完美匹配）.

预备知识

花（blossom）

一般图匹配和二分图匹配不同的是，图可能存在奇环. 可以将偶环视为二分图.

带花树算法（Blossom Algorithm）的处理方式时是遇到奇环就把它缩成一个 **花（Blossom）**，并把花中所有的点设为偶点. 既然花上的点都可以成为偶点，那么可以把整个花直接缩成一个偶点. 注意，一个花可以包含其它花.

这也可以变成线性规划和对偶问题，但是要对花进行一些处理.

顶标（vertex labeling）和等边（Equality Edge）

定义 z_u 是点 u 的顶标（vertex labeling），与 KM 算法中定义的顶标含义相同. 定义边 $e(u,v)$ 为 "等边" 当且仅当点 u 和点 v 的标号和等于边 e 的权值（$z_u+z_v=w(e)$），此时边的标号 $z_e=z_u+z_v-w(e)=0$.

一般图最大权完美匹配的线性规划

定义

因为一朵花最少有三个点，缩花后成为一个点. 设 O 为大小为 ≥ 3 奇数的集合的集合（包含所有花），$\gamma(S)$ 表示 S 集合中的边.

$$
\begin{aligned}
&\text{设} S \subseteq V \\
&\gamma(S) = \{(u,v) \in E : u \in S, v \in S\} \\
&O = \{B \subseteq V : |\ B\ | \text{是奇数且} |\ B\ | \geq 3\}
\end{aligned}
$$

对偶问题

原问题

$$
\begin{aligned}
&\max \sum_{e \in E} w(e) x_e \\
&\text{限制:} \\
&x(\delta(u)) = 1 : \forall u \in V \\
&x(\gamma(B)) \leq \left\lfloor \frac{|\ B\ |}{2} \right\rfloor : \forall B \in O \\
&x_e \geq 0 : \forall e \in E
\end{aligned}
$$

然后通过原始对偶（Primal-Dual）将问题转换为对偶问题.

对偶问题

$$\min \sum_{u\in V} z_u + \sum_{B\in O} \left\lfloor \frac{|B|}{2} \right\rfloor z_B$$

限制：

$z_B \geq 0 : \forall B \in O$

$z_e \geq 0 : \forall e \in E$

设$e=(u,v)$，这里

$$z_e = z_u + z_v - w(e) + \sum_{\substack{B\in O \\ u,v\in\gamma(B)}} z_B$$

$x_e = 1$ 的边是匹配边，$x_e = 0$ 的边是非匹配边．和二分图一样，我们必须满足 $x_e \in \{0,1\} : \forall e \in E$. 因此必须在最大权完美匹配的时候，让所有匹配边都是 **等边** 的.

和二分图不同的是，一般图多了 z_B 要处理．下面考虑 z_B 什么时候大于 0.

可以看出，尽量使 $z_B = 0$ 是最好的做法，但在不得已时还是要让 $z_B > 0$．在 $x(\gamma(B)) = \left\lfloor \frac{|B|}{2} \right\rfloor$ 且 $x(\delta(B)) = 1$ 时，让 $z_B > 0$ 即可．因为除了在这种情况下，$z_B > 0$ 是无意义的.

根据互补松弛条件，有以下的对应关系：

- 对于选中的边 e，必有 $z_e = 0$.
  $x_e > 0 \longrightarrow z_e = 0, \quad \forall e \in E$
- 对于选中的集合 B，$z_B > 0 \longrightarrow x(\gamma(B)) = \left\lfloor \frac{|B|}{2} \right\rfloor$，即所有 $z_B > 0$ 的集合 B，都被选了集合大小一半的边，也即集合 B 是一朵花，选中花中的一条边进行增广．同时，我们加入一个条件：$x(\delta(B)) = 1$，即只有花 B 向外连了一条边的时候，$z_B > 0$ 才是有意义的.
  $z_B > 0 \longrightarrow x(\gamma(B)) = \left\lfloor \frac{|B|}{2} \right\rfloor, x(\delta(B)) = 1 \quad \forall B \in O$

以「**等边**」的概念，结合之前的带花树算法：用「等边」构成的增广路不断进行扩充，由于用来扩充的边全是「等边」，最后得到的最大权完美匹配仍然全是「等边」.

处理花的问题

当遇到花的时候，要将它缩成一个偶点．将花中所有点都设为偶点，并让它的 $z_B = 0$.

由于缩花后会把花保存起来，直到满足某些条件才会拆开，所以不能用之前的方法记录花.

如果没有特殊说明，之前提到的点，都包含缩花形成的偶点.

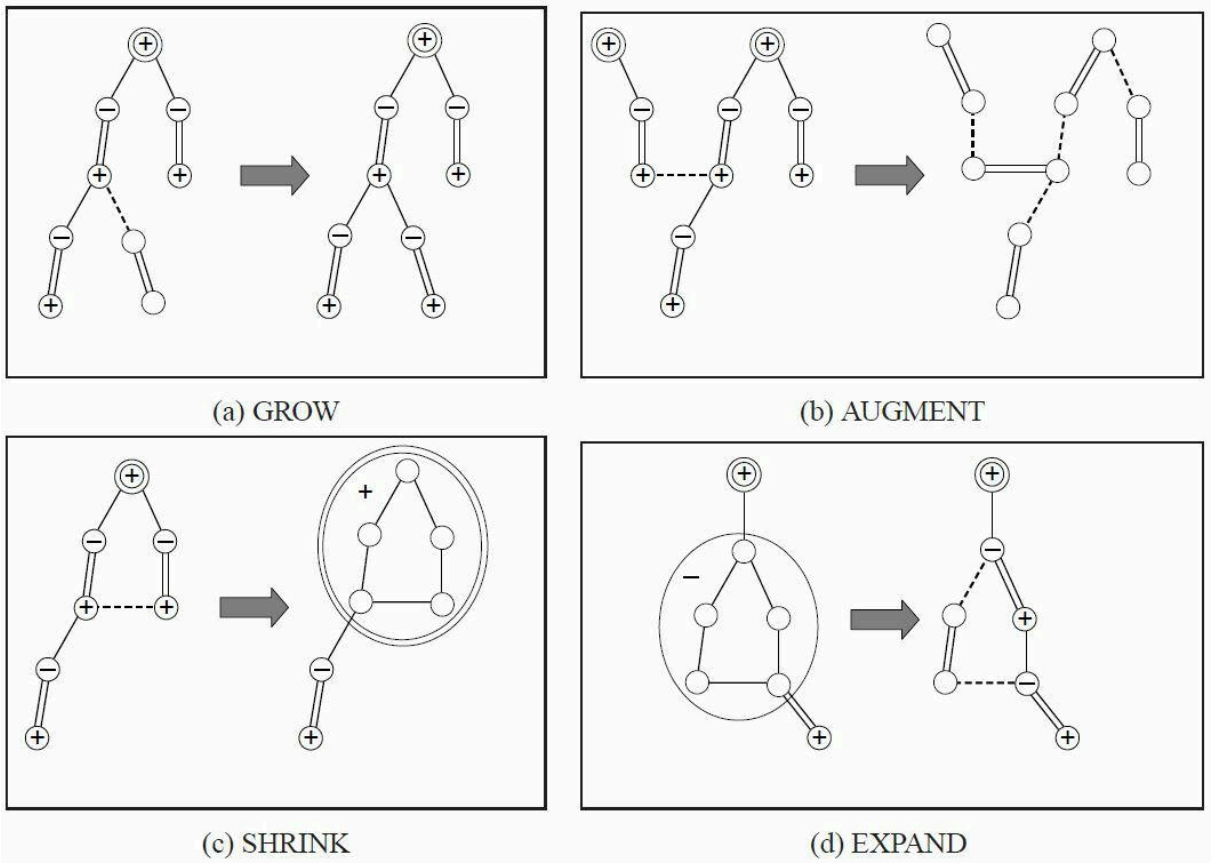
由于花也有可能缩成点被加入队列中，并且花的数量是不固定的，因此不能像之前一样枚举每个点来检查是否有增广路．因此，在进行广度优先搜索（BFS）时，必须将所有未匹配的点都放入队列中.

这样会同时产生很多棵交错树.

算法的四个步骤

这个算法可以分成四个步骤.

1. GROW（等边）：用 "等边" 构成交错树.
2. AUGMENT（增广）：找出增广路并扩充匹配.
3. SHRINK（缩花）：把花缩成一个点.
4. EXPAND（展开）：把花拆开.



在 AUGMENT 阶段时，因为所有未匹配点都会在不同的交错树上，所以当增广时两棵交错树的偶点连在一起，就表示找到了一条增广路.

找不到等边扩充

和二分图一样，也会有找不到「等边」扩充的问题. 这时就需要调整 vertex labeling.

调整 VERTEX LABELING

vertex labeling 仍要维持大于等于的性质，而且既有的「等边」不能被改变，还要让 z_B 尽量的小.

定义符号 奇偶点

以 u^- 来表示 u 在交错树上为奇点.

以 u^+ 来表示 u 在交错树上为偶点.

以 $u^\emptyset$ 来表示 u 不在任何一棵交错树上.

之后所有提到的 B 预设都是花，并同时代表缩花之后的点.

花也可以有奇花偶花之分，因此也适用 B^+、B^-、$B^\emptyset$ 等符号.

设目前有 r 棵交错树 $T_i = (U_{t_i}, V_{t_i}) : 1 \le i \le r$，令

$$d1 = \min(\{z_e : e = (u^+, v^\emptyset)\})$$

$$d2 = \min(\{z_e : e = (u^+, v^+),\ u^+ \in T_i,\ v^+ \in T_j,\ i \ne j\})/2$$

$$d3 = \min(\{z_{B^-} : B^- \in O\})/2$$

注意这里 B 是缩花之后的点，所以可以有奇偶性.

设 $d = min(d1, d2, d3)$，让

$$z_{u^+} -= d$$
$$z_{v^-} += d$$
$$z_{B^+} += 2d$$
$$z_{B^-} -= 2d$$

如果出现 $z_B = 0(d = d3)$，为了防止 $z_B < 0$ 的情况，所以要把这朵花拆了 (EXPAND)． 拆花后只留下花里的交替路径，并把花里不在交替路径上的点设为未走访 ($\emptyset$).

如此便制造了一条（以上）的等边，既有等边保持不动，并维持了 $z_e \geq 0 : \forall e \in E$ 的性质，且最低限度增加了 z_B，可以继续找增广路了.

一般图最大权匹配

以上求的是最大权完美匹配，求最大权匹配需要在 vertex labeling 额外增加一个限制：对于所有匹配点 u，$z_u > 0$.

开始时先设所有的 $z_u = max(\{w(e) : e \in E\})/2$.

vertex labeling 为 0 的点最后将成为未匹配点.

参考代码

这里为了方便实现，使用边权乘 2 来计算 z_e 的值，这样就不会出现浮点数误差了.

存储

```cpp
constexpr int INF = INT_MAX;
constexpr int MAXN = 400;

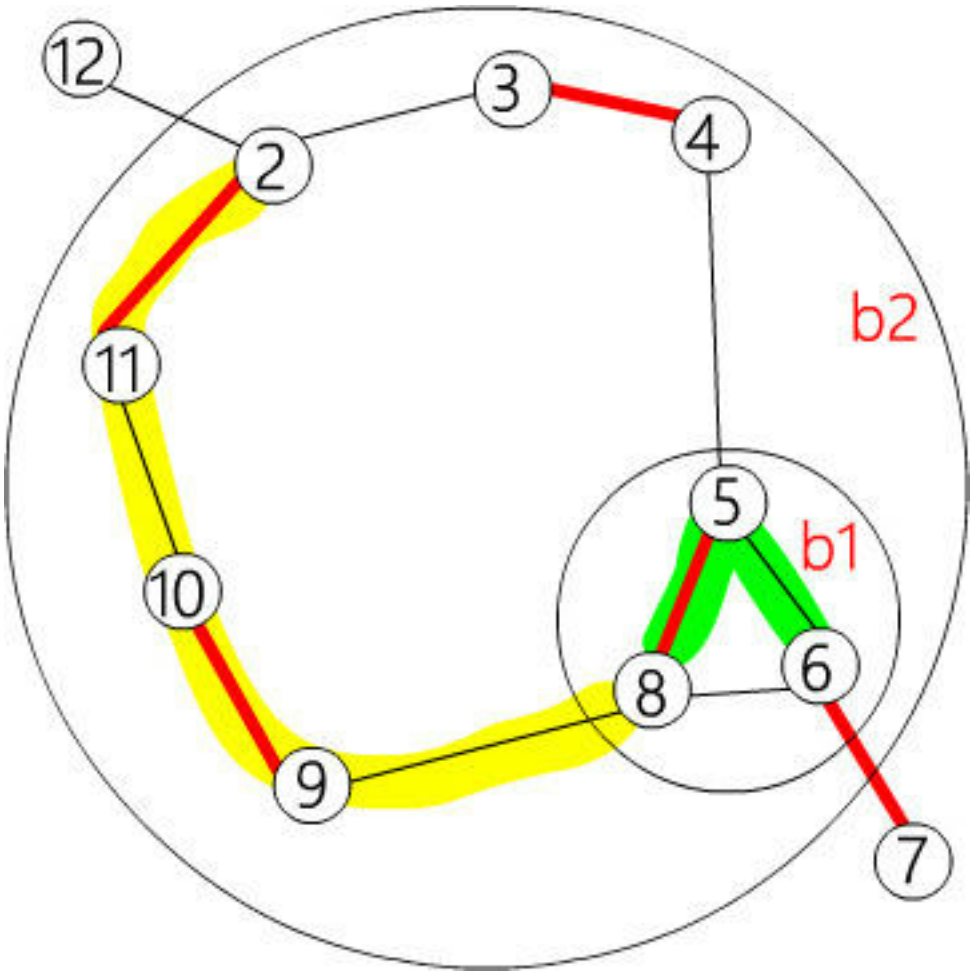
struct edge {
  int u, v, w;

  // 表示(u,v)为一条边其权重为w
  edge() {}

  edge(int u, int v, int w) : u(u), v(v), w(w) {}
};

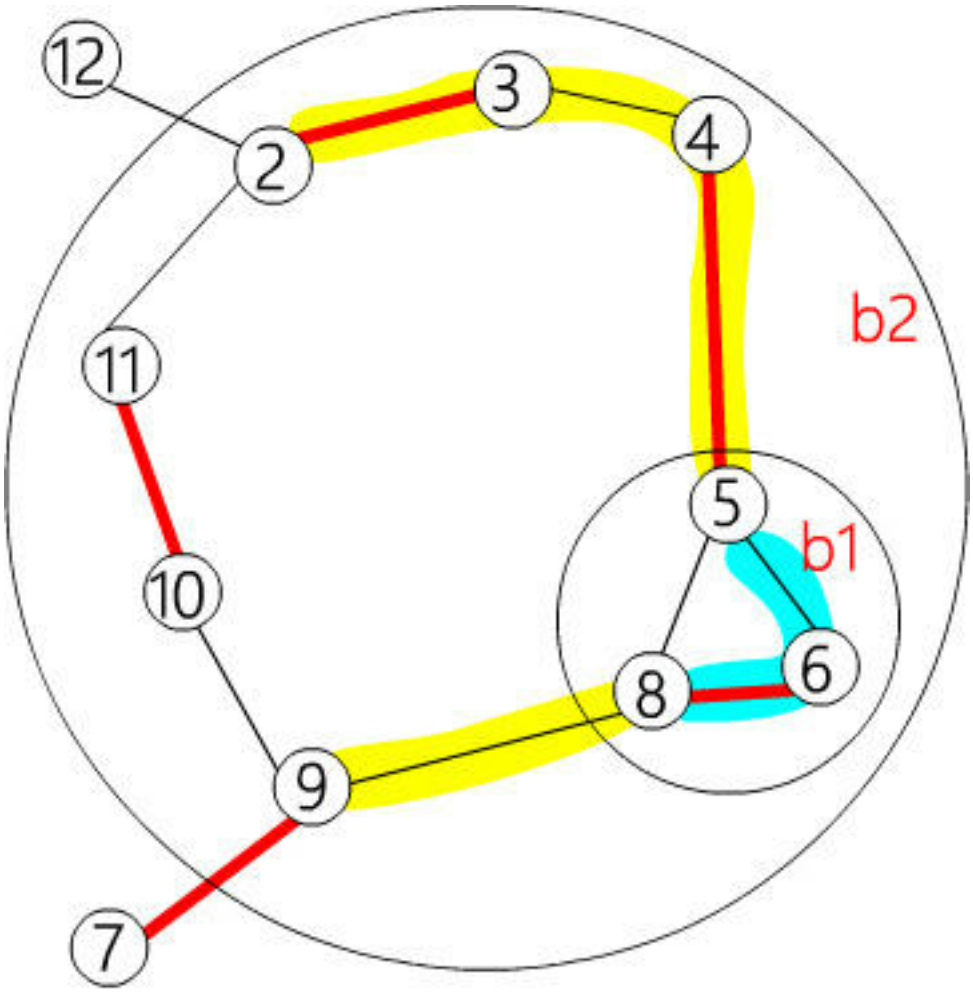
int n, n_x;
// 有n个点，编号为 1 ~ n
// n_x表示当前点加上花的数量，编号从n+1到n_x为花的节点
edge g[MAXN * 2 + 1][MAXN * 2 + 1];
// 图用邻接矩阵存储，因为最多有n-1朵花，所以大小为MAXN*
vector<int> flower[MAXN * 2 + 1];
// flower[b]记录了花b中有哪些点
// 我们记录花中的点的方式是只记录花里面的最外层花
```

下面是嵌套花的例子.



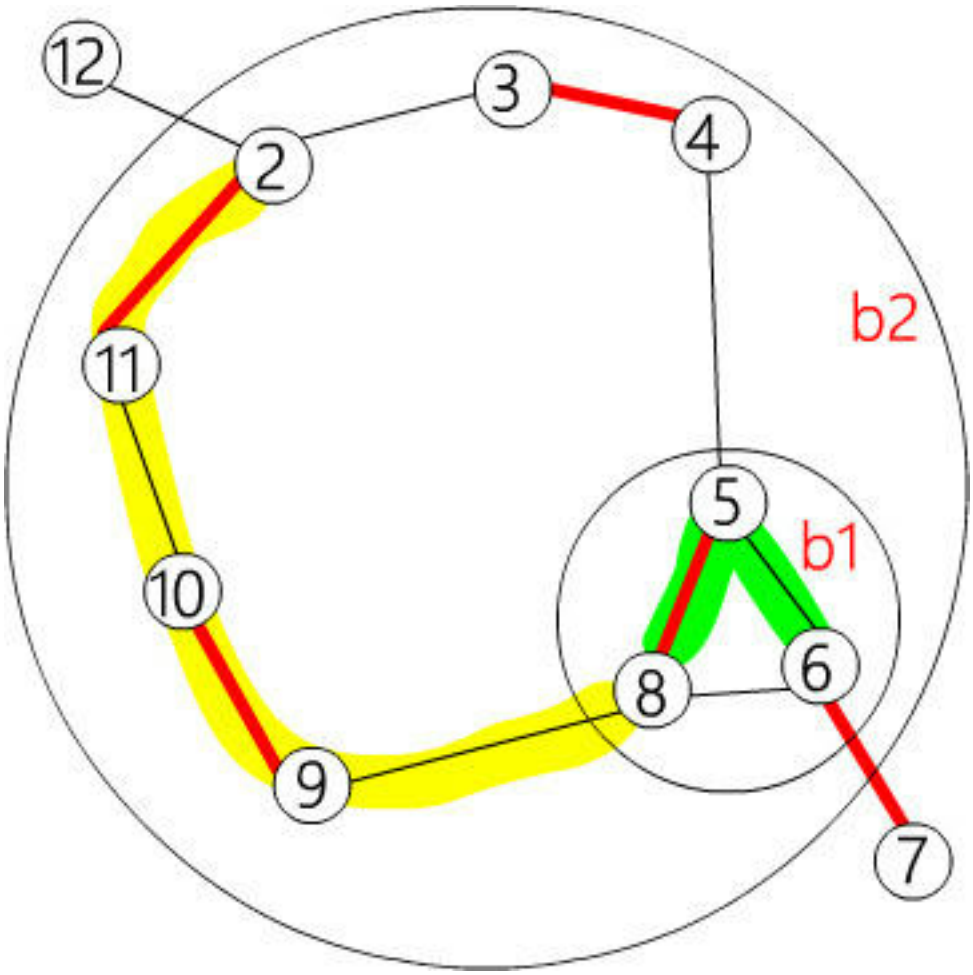
其中 $\{6,5,8\} \in b1, \{b1,4,3,2,11,10,9\} \in b2$. 存储为:

```
flower[b2] = {b1, 4, 3, 2, 11, 10, 9}
flower[b1] = {6, 5, 8}
```




```
flower[b2] = {9, b1, 4, 3, 2, 11, 10}
flower[b1] = {5, 8, 6}
```

```
int lab[MAXN * 2 + 1];
// lab[u]用来记录z_u, lab[b]用来记录z_B
int match[MAXN * 2 + 1], slack[MAXN * 2 + 1], st[MAXN * 2 + 1],
    pa[MAXN * 2 + 1];
// match[x]=y表示(x,y)是匹配，这里x、y可能是花
// slack[x]=u表示z(x,u)是所有和x相邻的边中最小的那条边
// 表示节点 x 所在的花是 b. 如果 x=b 且 b<=n，则表示 x
// 是一个普通节点（不属于任何花） 表示在交错树中，节点 v 的父节点是 u
int flower_from[MAXN * 2 + 1][MAXN + 1], S[MAXN * 2 + 1], vis[MAXN * 2 + 1];
/*
flower_from[b][x]=xs表示最大的包含x的b的子花是xs
x是b里面的一个点，xs是b里面的一朵花或一个点，同时x=xs或x是xs的其中一个点
*/
// S[u]={-1:没走过 0:偶点 1:奇点}
// vis只用在找lca的时候检查是不是走过了
queue<int> q;
// BFS找增广路用的queue
```




```
flower_from[b2][6] = b1
flower_from[b2][5] = b1
flower_from[b2][9] = 9
flower_from[b1][6] = 6
以此类推
```

```
int e_delta(const edge &e) {
  // 计算ze，为了方便起见先把所有边的权重乘二
  // 在花里面直接计算 e_delta 值会导致错误
  return lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2;
}

void update_slack(int u, int x) {
  // 以u更新slack[x]的值
  if (!slack[x] || e_delta(g[u][x]) < e_delta(g[slack[x]][x])) {
    slack[x] = u;
  }
}

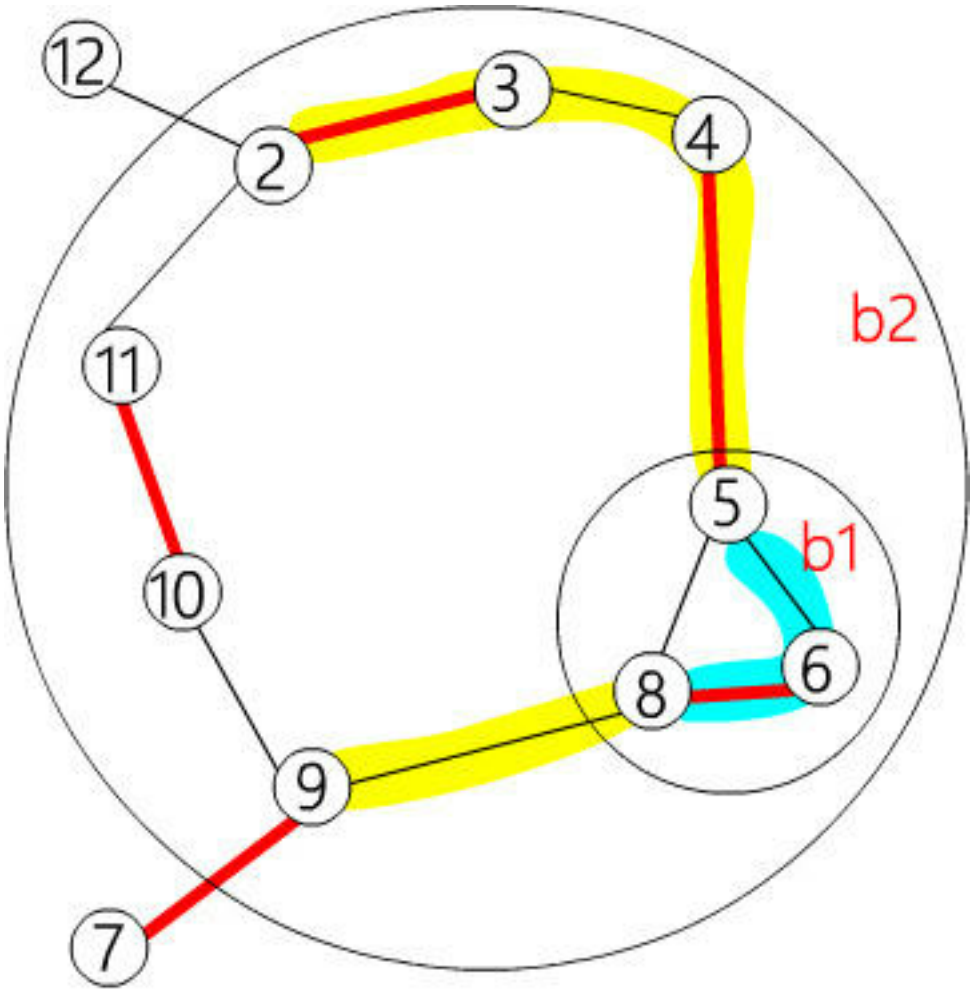
void set_slack(int x) {
  // 算出slack[x]的值，slack[x]=0表示x是交错树中的节点
  slack[x] = 0;
  for (int u = 1; u <= n; ++u) {
    if (g[u][x].w > 0 && st[u] != x && S[st[u]] == 0) {
      update_slack(u, x);
```

```
    }
  }
}
```

```
void q_push(int x) {
  // 把x丢到queue里面，我们设定queue不能直接push一朵花
  if (x <= n)
    q.push(x);
  else {
    // 若要push花必须将花里面原图的点都添加到queue中
    for (size_t i = 0; i < flower[x].size(); i++) {
      q_push(flower[x][i]);
    }
  }
}

void set_st(int x, int b) {
  // 将x所在的花设为b
  st[x] = b;
  if (x > n) {
    // 若x也是花的话，就必须要把x里面的点其所在的花也设为b
    for (size_t i = 0; i < flower[x].size(); ++i) {
      set_st(flower[x][i], b);
    }
  }
}
```

```
int get_pr(int b, int xr) {
  // xr是flower[b]中的一个点，返回值pr是它的位置
  // 为了方便程序运行，我们让 flower[b][0]~flower[b][pr]为花里的交替路
  int pr = find(flower[b].begin(), flower[b].end(), xr) - flower[b].begin();
  if (pr % 2 == 1) {
    // 检查他在花里的位置，如果 flower[b][0]~flower[b][pr] 不是交替路
    // 就把整朵花反转，重新计算 pr
    // 让 flower[b][0]~flower[b][pr] 为花里的交替路
    reverse(flower[b].begin() + 1, flower[b].end());
    return (int)flower[b].size() - pr;
  } else
    return pr;
}
```



如果使用 `get_pr(b2,11)`，`flower[b2]` 会变成 `{9,10,11,2,3,4,b1}`，并返回 2.

如果使用 `get_pr(b2,2)`，`flower[b2]` 会变成 `{9,b1,4,3,2,11,10}`，并返回 4.

```
void set_match(int u, int v) {
  // 设置u和v为匹配边，u和v有可能是花
  match[u] = g[u][v].v;
  if (u > n) {
    // 如果u是花的话
    edge e = g[u][v];
    int xr = flower_from[u][e.u];  // 找出e.u在flower[u]里的哪朵花上
    int pr = get_pr(u, xr);  // 找出xr的位置并让0~pr为花里的交替路径
    for (int i = 0; i < pr; ++i) {  // 把花里的交替路上的匹配边和非匹配边反转
      set_match(flower[u][i], flower[u][i ^ 1]);
    }
    set_match(xr, v);  // 设置(xr,v)为匹配边
    rotate(flower[u].begin(), flower[u].begin() + pr, flower[u].end());
    // 最后把pr设为花托，因为花的存法是flower[u][0]会是u的花托
    // 所以要把flower[u][pr] rotate 到最前面
  }
}

void augment(int u, int v) {
  // 把u和u的祖先全部增广，并设(u,v)为匹配边
  for (;;) {
    int xnv = st[match[u]];
```

```
    set_match(u, v);
    if (!xnv) return;
    set_match(xnv, st[pa[xnv]]);
    u = st[pa[xnv]];
    v = xnv;
  }
}

int get_lca(int u, int v) {
  // 找出u,v在交错树上的lca
  static int t = 0;
  for (++t; u || v; swap(u, v)) {
    if (u == 0) continue;
    if (vis[u] == t) return u;
    vis[u] = t;  // 这种方法可以不用清空vis数组
    u = st[match[u]];
    if (u) u = st[pa[u]];
  }
  return 0;
}
```

增加一朵奇花

```
void add_blossom(int u, int lca, int v) {
  // 将u,v,lca这朵花缩成一个点 b
  // 交错树上u,v的lca即为花托
  int b = n + 1;
  while (b <= n_x && st[b]) ++b;
  if (b > n_x) ++n_x;
  // 找出目前未使用的花的编号
  lab[b] = 0;              // 设置zB=0
  S[b] = 0;                // 整朵花为一个偶点
  match[b] = match[lca];   // 设置花的匹配边为花托的匹配边
  flower[b].clear();
  flower[b].push_back(lca);
  for (int x = u, y; x != lca; x = st[pa[y]]) {
    flower[b].push_back(x);
    y = st[match[x]];
    flower[b].push_back(y);
    q_push(y);
  }
  reverse(flower[b].begin() + 1, flower[b].end());
  for (int x = v, y; x != lca; x = st[pa[y]]) {
    flower[b].push_back(x);
    y = st[match[x]];
    flower[b].push_back(y);
    q_push(y);
  }
  // b中所有点以环形的方式加入flower[b]，并设花托为首个元素
  set_st(b, b);  // 把整朵花里所有的元素其所在的花设为b
  for (int x = 1; x <= n_x; ++x) {
    g[b][x].w = 0;
    g[x][b].w = 0;
```

```
  }
  for (int x = 1; x <= n; ++x) {
    flower_from[b][x] = 0;
  }
  for (size_t i = 0; i < flower[b].size(); ++i) {
    int xs = flower[b][i];
    for (int x = 1; x <= n_x; ++x) {
      // 设置b和x相邻的边为b里面和x相邻的边e_delta最小的那条
      if (g[b][x].w == 0 || e_delta(g[xs][x]) < e_delta(g[b][x])) {
        g[b][x] = g[xs][x];
        g[x][b] = g[x][xs];
      }
    }
    for (int x = 1; x <= n; ++x) {
      if (flower_from[xs][x]) {
        // 如果b里面的点xs有包含x
        // 那flower_from[b][x]就会是xs
        flower_from[b][x] = xs;
      }
    }
  }
  set_slack(b);
  // 最后必须要设置b的slack值
}
```

拆花

```
void expand_blossom(int b) {
  // b是奇花且zB=0时，必须要把b拆开
  // 因为只拆开b而已，所以如果b里面有包含其他的花
  // 不需要把他们拆开
  for (size_t i = 0; i < flower[b].size(); ++i) {
    set_st(flower[b][i], flower[b][i]);
    // 先把flower[b]里每个元素所在的花设为自己
  }
  int xr = flower_from[b][g[b][pa[b]].u];
  // xr表示交错路上b的父母节点在flower[b]里的哪朵花上
  int pr = get_pr(b, xr);  // 找出xr的位置并让0~pr为花里的交替路径
  for (int i = 0; i < pr; i += 2) {
    // 把交替路径拆开到交错树中
    // 并把交替路中的偶点丢到queue里
    int xs = flower[b][i];
    int xns = flower[b][i + 1];
    pa[xs] = g[xns][xs].u;
    S[xs] = 1;
    S[xns] = 0;
    slack[xs] = 0;
    set_slack(xns);
    q_push(xns);
  }
  S[xr] = 1;  // 这时xr会是奇点或奇花
```

```
  pa[xr] = pa[b];
  for (size_t i = pr + 1; i < flower[b].size(); ++i) {
    // 把花中所有不再交替路径上的点设为未走访
    int xs = flower[b][i];
    S[xs] = -1;
    set_slack(xs);
  }
  st[b] = 0;
}
```

尝试增广一条等边

```
bool on_found_edge(const edge &e) {
  // BFS时找到一条等边e
  // 要对它进行以下的处理
  // 这里u一定是偶点
  int u = st[e.u], v = st[e.v];
  if (S[v] == -1) {
    // v是未走访节点
    pa[v] = e.u;
    S[v] = 1;
    int nu = st[match[v]];
    slack[v] = 0;
    slack[nu] = 0;
    S[nu] = 0;
    q_push(nu);
  } else if (S[v] == 0) {
    // v是偶点
    int lca = get_lca(u, v);
    if (!lca) {  // lca=0表示u,v在不同的交错树上，有增广路
      augment(u, v);
      augment(v, u);
      return true;  // 找到增广路
    } else
      add_blossom(u, lca, v);
    // 否则u,v在同棵树上就会是一朵花，要缩花
  }
  return false;
}
```

增广

```
bool matching() {
  memset(S + 1, -1, sizeof(int) * n_x);
  memset(slack + 1, 0, sizeof(int) * n_x);
  q = queue<int>();  // 把queue清空
  for (int x = 1; x <= n_x; ++x) {
    if (st[x] == x && !match[x]) {
      // 把所有非匹配点加入queue里面，并设为偶点
      pa[x] = 0;
```

```
      S[x] = 0;
      q_push(x);
    }
  }
  if (q.empty()) return false;  // 所有点都有匹配了
  for (;;) {
    while (q.size()) {
      // BFS
      int u = q.front();
      q.pop();
      if (S[st[u]] == 1) continue;
      for (int v = 1; v <= n; ++v) {
        if (g[u][v].w > 0 && st[u] != st[v]) {
          if (e_delta(g[u][v]) == 0) {
            if (on_found_edge(g[u][v])) return true;
          } else
            update_slack(u, st[v]);
        }
      }
    }
    // 修改lab值
    int d = INF;
    for (int u = 1; u <= n; ++u) {
      // 这是为了防止出现lab<0的情况发生
      // 只要有任何一个lab[u]=0就结束程序
      if (S[st[u]] == 0) d = min(d, lab[u]);
    }
    for (int b = n + 1; b <= n_x; ++b) {
      if (st[b] == b && S[b] == 1) d = min(d, lab[b] / 2);
    }
    for (int x = 1; x <= n_x; ++x)
      if (st[x] == x && slack[x]) {
        if (S[x] == -1)
          d = min(d, e_delta(g[slack[x]][x]));
        else if (S[x] == 0)
          d = min(d, e_delta(g[slack[x]][x]) / 2);
      }
    for (int u = 1; u <= n; ++u) {
      if (S[st[u]] == 0) {
        if (lab[u] == d) return false;
        // 如果lab[u]=0就直接结束程序
        lab[u] -= d;
      } else if (S[st[u]] == 1)
        lab[u] += d;
    }
    for (int b = n + 1; b <= n_x; ++b) {
      if (st[b] == b) {
        if (S[st[b]] == 0)
          lab[b] += d * 2;
        else if (S[st[b]] == 1)
          lab[b] -= d * 2;
      }
```

```
    }
    q = queue<int>();  // 把queue清空
    for (int x = 1; x <= n_x; ++x) {
      // 检查看看有没有增广路径产生
      if (st[x] == x && slack[x] && st[slack[x]] != x &&
          e_delta(g[slack[x]][x]) == 0)
        if (on_found_edge(g[slack[x]][x])) return true;
    }
    for (int b = n + 1; b <= n_x; ++b) {
      // EXPAND的操作，把所有lab[b]=0的奇花拆开
      if (st[b] == b && S[b] == 1 && lab[b] == 0) expand_blossom(b);
    }
  }
  return false;
}
```

主函数

```
pair<long long, int> weight_blossom() {
  // 主函数，一开始先初始化
  memset(match + 1, 0, sizeof(int) * n);
  n_x = n;  // 一开始没有花
  int n_matches = 0;
  long long tot_weight = 0;
  for (int u = 0; u <= n; ++u) {
    // 先把自己所在的花设为自己
    st[u] = u;
    flower[u].clear();
  }
  int w_max = 0;
  for (int u = 1; u <= n; ++u)
    for (int v = 1; v <= n; ++v) {
      // u是一个点时，里面所包含的点只有自己
      flower_from[u][v] = (u == v ? u : 0);
      w_max = max(w_max, g[u][v].w);
      // 找出最大的边权
    }
  for (int u = 1; u <= n; ++u) lab[u] = w_max;
  // 让所有的lab=最大的边权
  // 因为这里实现是用边权乘二来计算ze的值所以不用除以二
  while (matching()) ++n_matches;
  for (int u = 1; u <= n; ++u)
    if (match[u] && match[u] < u) tot_weight += g[u][match[u]].w;
  return make_pair(tot_weight, n_matches);
}
```

初始化

很重要 使用前一定要初始化

```
void init_weight_graph() {
```

```
  // 在把边输入到图里面前必须要初始化
  // 因为是最大权匹配所以把不存在的边设为0
  for (int u = 1; u <= n; ++u)
    for (int v = 1; v <= n; ++v) g[u][v] = edge(u, v, 0);
}
```

复杂度分析

每朵花在一次 BFS 中只会被缩花或拆花一次．每次缩花或拆花的时间复杂度为 $O(|V|)$．最多总共有 $O(|V|)$ 朵花，所以花的处理花费 $O(|V|^2)$ 的时间．而 BFS 花费 $O(|V|+|E|)$ 的时间复杂度．因此，找增广路花费 $O(|V|+|E|)+O(|V|^2)=O(|V|^2)$ 的时间复杂度．

最多做 $|V|$ 次 BFS．所以，总时间复杂度为 $O(|V|^3)$．

习题

- UOJ #81. 一般图最大权匹配[1]

参考资料

1. Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm"[2]
2. 从匈牙利算法到带权带花树——详解对偶问题在图匹配上的应用[3]

外部参考资料

[1] UOJ #81. 一般图最大权匹配

[2] Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm"

[3] 从匈牙利算法到带权带花树——详解对偶问题在图匹配上的应用

11.24.6 稳定匹配

引入

稳定匹配问题（stable matching problem）是组合优化和合作博弈论中的经典问题．相较于传统的图论匹配问题，稳定匹配引入了个体偏好和稳定性的限制，这使得算法设计更多地依赖于偏好顺序而非单纯的图结构．稳定匹配问题的模型中，每个个体对潜在的匹配对象具有偏好，而稳定匹配问题希望能在它们之间建立一种稳定的匹配关系．一个稳定的匹配中，不存在任何一组个体，会因为能得到更优的选择而合谋偏离当前的匹配结果．稳定匹配及其相关问题广泛地应用于劳动力市场、学校录取、医疗资源分配等场景中．

算法竞赛中最常出现的稳定匹配问题是双边市场的一对一匹配，即稳定婚姻问题．本文将重点介绍稳定婚姻问题及其算法．

稳定婚姻问题

稳定婚姻问题是最早研究的稳定匹配问题．类似于二分图匹配，它可以描述为婚恋市场上的匹配问题：假设有男士和女士若干，每个人都对异性有一组偏好顺序，目标是找到一种匹配方式，使得没有一对男女更愿意抛弃各自的匹配对象而选择彼此．

问题描述

匹配市场由若干男士 M 和若干女士 W 构成．每个人都对异性有严格的偏好顺序：

- 对于每位男士 $m\in M$，都存在集合 $W\cup\{m\}$ 上一个严格的全序 $\underset{m}{\preceq}$；
- 对于每位女士 $w\in W$，都存在集合 $M\cup\{w\}$ 上一个严格的全序 $\underset{w}{\preceq}$．

除了在异性之间相互比较之外，每个人还会将自身加入到这个偏好顺序中．这表示，这个人只会接受比单身更偏好的异性；这些异性称为 **可接受的**（acceptable）．显然，不可接受的异性的偏好顺序是无足轻

重的；原则上，只需要给出可接受的异性之间的偏好顺序即可．所以，这些存在不可接受异性的偏好也称为列表不完整的偏好（preferences with incomplete lists）．

例子

假设 m 是一位男士，w_1, w_2, w_3 是三位女士，且有偏好关系 $w_1 \underset{m}{\prec} m \underset{m}{\prec} w_2 \underset{m}{\prec} w_3$ 成立．那么，男士 m 相对于和女士 w_1 匹配，更喜欢单身；相对于单身，更喜欢和女士 w_2 匹配；相对于和女士 w_2 匹配，更喜欢和女士 w_3 匹配．对于男士 m，女士 w_1 就是不可接受的，女士 w_2, w_3 就是可接受的．

市场上的一个 **匹配** $\mu : M \cup W \to M \cup W$ 需要满足如下性质：

- 每个人只能匹配异性或其自身，即对所有 $m \in M$ 都有 $\mu(m) \in W \cup \{m\}$ 且对所有 $w \in W$ 都有 $\mu(w) \in M \cup \{w\}$．
- 匹配是相互的，即对所有 $i \in M \cup W$ 都有 $i = \mu(\mu(i))$．

一个匹配 μ 中可能存在两种不稳定因素：

- 如果存在个体 $i \in M \cup W$ 使得 $\mu(i) \underset{i}{\prec} i$，也就是说，相对于当前的匹配对象，个体 i 宁愿单身，那么，就称 i 是匹配 μ 的 **阻塞个体**（blocking individual）．
- 如果存在一对异性 $m \in M$ 和 $w \in W$ 使得 $\mu(m) \underset{m}{\prec} w$ 且 $\mu(w) \underset{w}{\prec} m$，也就是说，相对于当前各自的匹配对象，男士 m 和女士 w 更希望和对方在一起，那么，就称 (m, w) 是匹配 μ 的 **阻塞对**（blocking pair）．

如果一个匹配 μ 既不存在阻塞个体，也不存在阻塞对，就称匹配 μ 是 **稳定的**（stable）．稳定匹配中，所有人都无法破坏当前的局面：单身的人找不到愿意同他在一起的人；结婚的人既不愿意离婚单身，也找不到愿意同他私奔的人．

稳定匹配问题就是在问：对于任意给定的一组偏好顺序，是否都存在一个稳定匹配？如果是，如何求出这样的稳定匹配？

Gale–Shapley 算法

Gale 和 Shapley 在 1962 年提出了 **延迟接受算法**（deferred acceptance algorithm），可以对任意给定的一组偏好顺序求出一个稳定匹配．因此，稳定匹配一定是存在的．

Gale–Shapley 算法有两个对称的版本，分别由男士求婚和女士求婚．以男士求婚的 Gale–Shapley 算法为例，算法流程如下：

1. 算法开始时，每位女士都视为保留着她对其自身的求婚请求，每位男士都标记为活跃的．
2. 活跃的男士会向他可接受但是尚未求婚过的女士中最喜欢的那位求婚；如果这样的女士不存在，就无需进行任何操作．无论求婚与否，将所有男士都标记为不活跃的．
3. 收到新的求婚请求的女士，会将它们与之前保留的求婚请求比较，只保留其中最喜欢的那一个（可能是她自身），并拒绝所有其他的求婚请求．将遭到拒绝的男士恢复标记为活跃的．
4. 重复前两个步骤，直到没有活跃的男士为止．此时，女士接受她们当前保留的求婚请求．这样得到的匹配结果，就是一个稳定匹配．

由于每位男士向每位女士至多求婚一次，算法在 $O(|M||W|)$ 时间内一定会结束．

参考实现如下：

模板题 SPOJ STABLEMP - Stable Marriage Problem[1] 参考实现

```cpp
#include <iostream>
#include <queue>
#include <vector>

// Solver for stable marriage problems.
```

```cpp
// Assume strict preferences with incomplete lists.
struct StableMatching {
  int nx, ny;
  std::vector<std::vector<int>> pref_x,
      pref_y;  // Preferences: preferred first, only acceptable.
  std::vector<int> match_x, match_y;  // Matching: -1 means unmatched.

  StableMatching(int nx, int ny)
      : nx(nx),
        ny(ny),
        pref_x(nx),
        pref_y(ny),
        match_x(nx, -1),
        match_y(ny, -1) {}

  // Gale-Shapley algorithm.
  // Complexity: O(nx * ny).
  void solve() {
    // Compute Y's ranks over X.
    std::vector<std::vector<int>> ranks(ny, std::vector<int>(nx));
    for (int j = 0; j != ny; ++j) {
      for (int i = 0; i != (int)pref_y[j].size(); ++i) {
        ranks[j][pref_y[j][i]] = nx - i;
      }
    }
    // Initialize.
    std::vector<int> waitlist(ny);  // Best proposal rank for j in Y.
    std::vector<int> ids(nx);       // Next j in Y for i in X to propose to.
    std::queue<int> q;              // Currently active i's in X.
    for (int i = 0; i != nx; ++i) q.push(i);
    // Loop.
    while (!q.empty()) {
      auto i = q.front();
      q.pop();
      if (ids[i] == (int)pref_x[i].size()) continue;  // List exhausted.
      auto j = pref_x[i][ids[i]++];
      if (ranks[j][i] > waitlist[j]) {
        if (waitlist[j]) q.push(pref_y[j][nx - waitlist[j]]);
        waitlist[j] = ranks[j][i];
      } else {
        q.push(i);
      }
    }
    // Output.
    for (int j = 0; j != ny; ++j) {
      if (waitlist[j]) {
        int i = pref_y[j][nx - waitlist[j]];
        match_x[i] = j;
        match_y[j] = i;
      }
    }
  }
```

```
};

void solve() {
  // Input.
  int n;
  std::cin >> n;
  StableMatching solver(n, n);
  for (int j = 0, x; j < n; ++j) {
    auto& cur = solver.pref_y[j];
    std::cin >> x;
    for (int i = 0; i < n; ++i) {
      std::cin >> x;
      cur.push_back(x - 1);
    }
  }
  for (int i = 0, y; i < n; ++i) {
    auto& cur = solver.pref_x[i];
    std::cin >> y;
    for (int j = 0; j < n; ++j) {
      std::cin >> y;
      cur.push_back(y - 1);
    }
  }
  // Solve the problem.
  solver.solve();
  // Output.
  for (int i = 0; i < n; ++i) {
    std::cout << (i + 1) << ' ' << (solver.match_x[i] + 1) << '\n';
  }
}

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int t;
  std::cin >> t;
  for (; t; --t) {
    solve();
  }
  return 0;
}

```

稳定匹配的性质

稳定匹配有着良好的理论性质．首先，Gale–Shapley 算法构造性地证明，稳定匹配一定存在．

定理 1（Gale and Shapley, 1962）

Gale–Shapley 算法得到的是一个稳定匹配．因此，稳定匹配存在．

证明

男士不会对他不接受的女士求婚，女士也会立即拒绝她不接受的男士的求婚．因此，最终互相匹配的男士和女士一定是彼此接受的，不可能存在阻塞个体．要证明它是稳定匹配，只需要说明不存在阻塞对．

反证法．假设 (m,w) 是一个阻塞对．那么，由于 $\mu(m) \underset{m}{\prec} w$，男士 m 一定已经向 w 求过婚．但是，既然女士 w 拒绝了 m，她一定是收到了她更喜欢的人 m' 的求婚请求．如果 $m' \neq \mu(w)$，那么，女士 w 相对于 m' 只会更喜欢 $\mu(w)$．由此，相对于 m，女士 w 一定更喜欢最终的匹配对象 $\mu(w)$．这与 (m,w) 是阻塞对矛盾．所以，匹配是稳定的．

推论

如果 $|M| = |W|$ 且所有异性都是可接受的，那么，存在一个稳定的完美匹配．

Gale–Shapley 算法中，可以由男士求婚，也可以由女士求婚．一般情况下，这两个版本的 Gale–Shapley 算法得到的稳定匹配并不相同．事实上，由男士求婚的 Gale–Shapley 算法得到的稳定匹配是所有稳定匹配中，对于男士最有利的；反之亦然．

定理 2（Gale and Shapley, 1962）

设 μ_M 和 μ_W 分别是由男士和女士求婚的 Gale–Shapley 算法得到的稳定匹配．对于任何稳定匹配 μ，都有 $\mu(m) \underset{m}{\preceq} \mu_M(m)$ 对于所有 $m \in M$ 成立，且 $\mu(w) \underset{w}{\preceq} \mu_W(w)$ 对所有 $w \in W$ 成立．

证明

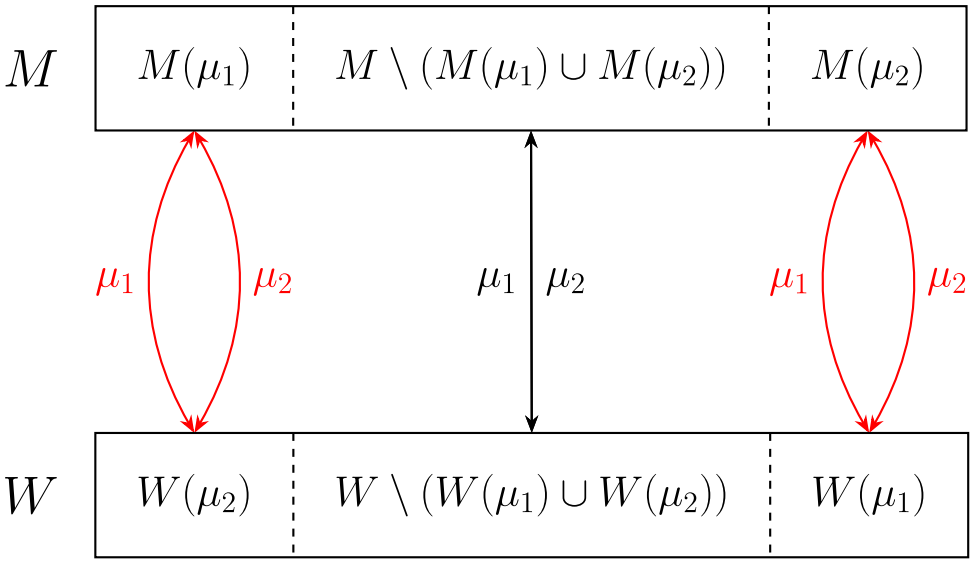
根据对称性，只需要证明 $\mu(m) \underset{m}{\preceq} \mu_M(m)$ 对于所有 $m \in M$ 成立．为此，仍然考虑由男士求婚的 Gale–Shapley 算法，并记 $k(m,w)$ 为女士 w 拒绝男士 m 的求婚时，算法进行到的轮次．这个轮次对于所有满足 $\mu_M(m) \underset{m}{\prec} w$ 的 (m,w) 都是良定义的．

假设 μ_M 并非对所有男士都最有利的，也就是说，存在稳定匹配 μ 和男士 $m \in M$ 使得 $\mu_M(m) \underset{m}{\prec} \mu(m)$ 成立．由于匹配 μ_M 是稳定的，就有 $m \underset{m}{\preceq} \mu_M(m) \underset{m}{\prec} \mu(m)$，所以 $\mu(m)$ 是女士，一定有良定义的 $k(m,\mu(m))$．于是，不妨设 m 恰为所有这样的男士中，$k(m,\mu(m))$ 最小的那个．设算法过程中，女士 $w = \mu(m)$ 拒绝男士 m 时保留的是男士 m' 的求婚请求，也就是说，$m = \mu(w) \underset{w}{\prec} m'$．由于 μ 是稳定匹配，(m',w) 不能是阻塞对，又有 $\mu(m') \neq w$，所以，$w \underset{m'}{\prec} \mu(m')$．由于 Gale–Shapley 算法过程中，女士 w 未必会保留 m' 的求婚请求到最后，所以 $\mu_M(m') \underset{m'}{\preceq} w \underset{m'}{\prec} \mu(m')$．此时，$k(m',\mu(m'))$ 是良定义的．而且，由于 $w \underset{m'}{\prec} \mu(m')$，所以，女士 $\mu(m')$ 拒绝 m' 的求婚请求之后，才会有 w 保留 m' 的求婚请求，也就是说，$k(m',\mu(m')) < k(m,\mu(m))$．这与 m 的选取相矛盾．所以，根据反证法，μ_M 是对所有男士最有利的稳定匹配．

一个匹配市场可能存在指数级数量的稳定匹配．设 $\mathcal{S}$ 为全体稳定匹配的集合．在这个集合上，可以定义两个偏序：

- $\mu_1 \underset{M}{\preceq} \mu_2$，当且仅当 $\mu_1(m) \underset{m}{\preceq} \mu_2(m)$ 对所有 $m \in M$ 都成立；
- $\mu_1 \underset{W}{\preceq} \mu_2$，当且仅当 $\mu_1(w) \underset{w}{\preceq} \mu_2(w)$ 对所有 $w \in W$ 都成立．

这两个偏序分别表示匹配结果对于所有男士和所有女士都更优．一般地，两个稳定匹配未必是可比的．但是，任意两个稳定匹配都诱导如图所示的分解，使得分解所得的三个部分中，分别成立 $\mu_1 \underset{M}{\preceq} \mu_2$，$\mu_1 = \mu_2$ 和 $\mu_2 \underset{M}{\preceq} \mu_1$．注意，尽管没有直接绘制出，但是 $\mu_1 = \mu_2$ 的那一部分其实包含了匹配到自身（即未匹配）的情形．



这一分解依赖于如下的引理：

引理（Knuth, 1976）

设 μ_1 和 μ_2 是两个稳定匹配．设 $M(\mu_i)=\{m\in M:\mu_j(m)\underset{m}{\prec}\mu_i(m)\}$ 和 $W(\mu_i)=\{w\in W:\mu_j(w)\underset{w}{\prec}\mu_i(w)\}$ 分别为更偏好 μ_i 中的匹配结果的男士和女士的集合，其中，$i,j=1,2$ 且 $i\neq j$．那么，μ_1 和 μ_2 都是 $M(\mu_1)$ 与 $W(\mu_2)$ 之间的双射，也都是 $M(\mu_2)$ 与 $W(\mu_1)$ 之间的双射．

证明

设 $m\in M(\mu_1)$．由于 $m\underset{m}{\preceq}\mu_2(m)\underset{m}{\prec}\mu_1(m)$，所以 $\mu_1(m)\in W$．令 $w=\mu_1(m)$．因为 $\mu_2(w)\neq m$，而 $\mu_2(w)\underset{w}{\prec}m$ 又意味着 (m,w) 是 μ_2 的阻塞对，所以，$\mu_1(w)=m\underset{w}{\prec}\mu_2(w)$．也就是说，$w\in W(\mu_2)$．这说明，$\mu_1(M(\mu_1))\subseteq W(\mu_2)$．由对称性，还可以建立 $\mu_2(W(\mu_2))\subseteq M(\mu_1)$．由于 μ_1 和 μ_2 都是单射，所以，$|M(\mu_1)|=|W(\mu_2)|$ 且这两个映射都是满射．这就说明，μ_1 和 μ_2 都是 $M(\mu_1)$ 与 $W(\mu_2)$ 之间的双射．同理，它们也都是 $M(\mu_2)$ 与 $W(\mu_1)$ 之间的双射．

这一引理说明，偏序集 $(\mathcal{S},\underset{M}{\preceq})$ 和 $(\mathcal{S},\underset{W}{\preceq})$ 互为 对偶→9.20@p1951．而且，在每个偏序下，集合 $\mathcal{S}$ 都构成一个 格→9.20@p1951．因为 $\mathcal{S}$ 是有限的，这两个格一定存在最大元和最小元．这两个最值元素，分别就是前文提到的两个版本的 Gale–Shapley 算法所得到的稳定匹配．

定理 3（Conway and Knuth, 1976）

偏序集 $(\mathcal{S},\underset{M}{\preceq})$ 和 $(\mathcal{S},\underset{W}{\preceq})$ 是相互对偶的格．而且，μ_M 和 μ_W 分别是 $(\mathcal{S},\underset{M}{\preceq})$ 的最大元和最小元，也分别是 $(\mathcal{S},\underset{W}{\preceq})$ 的最小元和最大元．

证明

根据引理，容易说明两个偏序集是对偶的．如果 $\mu_1\underset{M}{\preceq}\mu_2$，这说明 $M(\mu_1)=\emptyset$；由引理，$W(\mu_2)=\emptyset$，此即 $\mu_2\underset{W}{\preceq}\mu_1$．反之亦然．这就说明两者互为对偶．再结合前文的定理 2，就得到 μ_M 和 μ_W 是两个偏序集的最值元素．命题中还需要证明的是，两个偏序集是格．由对称性，只需要证明 $(\mathcal{S},\underset{M}{\preceq})$ 是格．再根据交和并运算的对称性，只需要证明稳定匹配的并仍然是稳定匹配．形式化地，对于任意 $\mu_1,\mu_2\in\mathcal{S}$，需要证明对于所有 $m\in M$ 都满足 $\mu(m)=\mu_1(m)\vee_m\mu_2(m)$ 的匹配 $\mu=\mu_1\vee_M\mu_2$ 是稳定匹配，其中，$\vee_m$ 是全序 $\underset{m}{\preceq}$ 下的并运算（即两者中 m 更喜欢的那个）．

仍采用引理中的记号．对于 $i \in M(\mu_1) \cup W(\mu_2)$，令 $\mu(i) = \mu_1(i)$；否则，令 $\mu(i) = \mu_2(i)$．由引理，μ_1 和 μ_2 都将集合 $M(\mu_1) \cup W(\mu_2)$ 映射到其自身，所以 μ 是良定义的匹配．由于 μ_1 和 μ_2 都是稳定的，不存在阻塞个体，μ 也同样如此．假设 (m, w) 是 μ 的阻塞对．如果 $m \in M(\mu_1)$，那么，$\mu_2(m) \underset{m}{\prec} \mu_1(m) = \mu(m) \underset{m}{\prec} w$．此时，如果 $w \in W(\mu_2)$，那么，$\mu_1(w) = \mu(w) \underset{w}{\prec} m$，所以，$(m, w)$ 是 μ_1 的阻塞对，矛盾；否则，$w \in W \setminus W(\mu_2)$，有 $\mu_2(w) = \mu(w) \underset{w}{\prec} m$，所以，$(m, w)$ 是 μ_2 的阻塞对，也矛盾．类似地，$m \in M \setminus M(\mu_1)$ 的情形也只能导出矛盾．由反证法可知，这样的阻塞对不存在．所以，$\mu_1 \vee_M \mu_2$ 是稳定匹配．命题得证．

最后，在所有稳定匹配中，未匹配的男士和女士的集合都是固定的．

定理 4（McVitie and Wilson, 1970）

设 μ_1 和 μ_2 是两个稳定匹配．那么，μ_1 和 μ_2 的不动点集合相同．

证明

假设存在 $m \in M$ 使得 $\mu_1(m) = m$ 且 $\mu_2(m) \neq m$ 对某组 $\mu_1, \mu_2 \in \mathcal{S}$ 成立．此时，有 $m \in M(\mu_2)$．由引理可知，$m = \mu_1(m) \in W(\mu_1)$，这与 $m \in M$ 矛盾．所以，不存在这样的 $m \in M$．同理，也不存在这样的 $w \in W$．所以，任意两个稳定匹配的不动点集合必然相同．

除了本节讨论的这些性质外，稳定匹配还有一些良好的策略性质．关于这些内容，可以参见文末提供的文献．

相关问题

稳定匹配及其类似问题还出现在许多其他的情境中．

学院招生问题

如果将稳定婚姻问题中的一对一匹配的限制放宽，允许多对一匹配，就得到了 **学院招生问题**（college admissions problem）．此时，一个学院可以招收多名学生，只要不超过招生限额；但是，一名学生仍然只允许进入至多一个学院学习．类似的情景还出现在公司招聘、医院招收实习医生等场景中．

对于这类问题，Gale–Shapley 算法仍然适用．例如，由学生申请的 Gale–Shapley 算法中，学院可以维持一个不超过限额长度的候选名单（waitlist），每次只要在申请数量超过限额时，拒绝最差学生的申请即可．前文关于稳定匹配性质的讨论对于这一场景仍然适用．特别地，定理 4 对应的版本是，在所有稳定匹配中，学院能够招到的学生人数是固定的．这也称为 **乡村医院定理**（rural hospitals theorem）．因为它意味着，无论如何更改匹配机制，只要得到的结果是稳定的，那些招不满医生的乡村医院永远也招不满．

稳定室友问题

如果将稳定婚姻问题中，只能匹配异性的条件放宽，就得到了 **稳定室友问题**（stable roommates problem）．此时，初始只有若干名学生，需要两两结对成为室友．对于这类问题，稳定匹配未必存在．Irving 在 1985 年提出了可以在 $O(n^2)$ 时间内解决该问题的算法．

住房市场问题

稳定婚姻问题中，两组个体互相有偏好，所以是双边匹配问题．除此之外，还可以考虑单边匹配问题．一个常见的场景是 **住房市场问题**（housing market problem）．有 n 名居民，各自拥有一套住房．每人对所有住房有一个严格偏好．现在，要将这些住房重新分配给这些居民，要求每名居民都不能分配到比初始更差的住房，且不存在任何数量的居民，可以私自交换房产，得到更满意的结局．对于这一问题，可以通过 Top Trading Cycle 算法在 $O(n^2)$ 时间内解决．这类问题还出现在肾移植等场景中．

习题

- UOJ 41.【清华集训 2014】矩阵变换[2]
- Codeforces 1147 F. Zigzag Game[3]

参考资料与注释

- 什么是算法：如何寻找稳定的婚姻搭配 - Matrix67[4]
- Gale–Shapley 算法：在二分图中寻找稳定匹配[5]
- Stable matching problem - Wikipedia[6]
- Lattice of stable matchings - Wikipedia[7]
- Stable roommates problem - Wikipedia[8]
- Top trading cycle - Wikipedia[9]
- Stable matching: Theory, evidence, and practical design - the 2012 Nobel Prize in Economics[10]
- Notes on Matching and Market Design by Xiang Sun[11]
- Gale, David, and Lloyd S. Shapley. "College admissions and the stability of marriage." The American Mathematical Monthly 69, no. 1 (1962): 9-15.
- Irving, Robert W. "An efficient algorithm for the stable roommates problem." Journal of Algorithms 6, no. 4 (1985): 577-595.
- Knuth, Donald Ervin. "Mariages stables et leurs relations avec d'autres problèmes combinatoires." Les Presses de l'Université de Montréal (1976).
- McVitie, David G., and Leslie B. Wilson. "Stable marriage assignment for unequal sets." BIT Numerical Mathematics 10, no. 3 (1970): 295-309.
- Roth, Alvin E., and Marilda Sotomayor. "Two-sided matching." Handbook of game theory with economic applications 1 (1992): 485-541.
- Roth, Alvin E. "Deferred acceptance algorithms: History, theory, practice, and open questions." International Journal of Game Theory 36, no. 3-4 (2008): 537-569.

外部参考资料

[1] SPOJ STABLEMP - Stable Marriage Problem

[2] UOJ 41.【清华集训 2014】矩阵变换

[3] Codeforces 1147 F. Zigzag Game

[4] 什么是算法：如何寻找稳定的婚姻搭配 - Matrix67

[5] Gale–Shapley 算法：在二分图中寻找稳定匹配

[6] Stable matching problem - Wikipedia

[7] Lattice of stable matchings - Wikipedia

[8] Stable roommates problem - Wikipedia

[9] Top trading cycle - Wikipedia

[10] Stable matching: Theory, evidence, and practical design - the 2012 Nobel Prize in Economics

[11] Notes on Matching and Market Design by Xiang Sun

11.25 Prüfer 序列

Note

本文翻译自 e-maxx Prüfer Code[1]．另外解释一下，原文的结点是从 0 开始标号的，本文按照大多数人的习惯改成了从 1 标号．

这篇文章介绍 Prüfer 序列 (Prüfer code)，这是一种将带标号的树用一个唯一的整数序列表示的方法．

使用 Prüfer 序列可以证明 凯莱公式→11@p2597(Cayley's formula)．并且我们也会讲解如何计算在一个图中加边使图连通的方案数．

注意：我们不考虑含有 1 个结点的树．

Prüfer 序列

引入

Prüfer 序列可以将一个带标号 n 个结点的树用 $[1,n]$ 中的 $n-2$ 个整数表示．你也可以把它理解为完全图的生成树与数列之间的双射．常用组合计数问题中．

Heinz Prüfer 于 1918 年发明这个序列来证明 凯莱公式→11@p2597.

对树建立 Prüfer 序列

Prüfer 是这样建立的：每次选择一个编号最小的叶结点并删掉它，然后在序列中记录下它连接到的那个结点．重复 $n-2$ 次后就只剩下两个结点，算法结束．

显然使用堆可以做到 $O(n\log n)$ 的复杂度

实现

- C++

```cpp
// 代码摘自原文，结点是从 0 标号的
vector<vector<int>> adj;

vector<int> pruefer_code() {
  int n = adj.size();
  set<int> leafs;
  vector<int> degree(n);
  vector<bool> killed(n);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1) leafs.insert(i);
  }

  vector<int> code(n - 2);
  for (int i = 0; i < n - 2; i++) {
    int leaf = *leafs.begin();
    leafs.erase(leafs.begin());
    killed[leaf] = true;
    int v;
    for (int u : adj[leaf])
      if (!killed[u]) v = u;
    code[i] = v;
    if (--degree[v] == 1) leafs.insert(v);
  }
  return code;
}
```

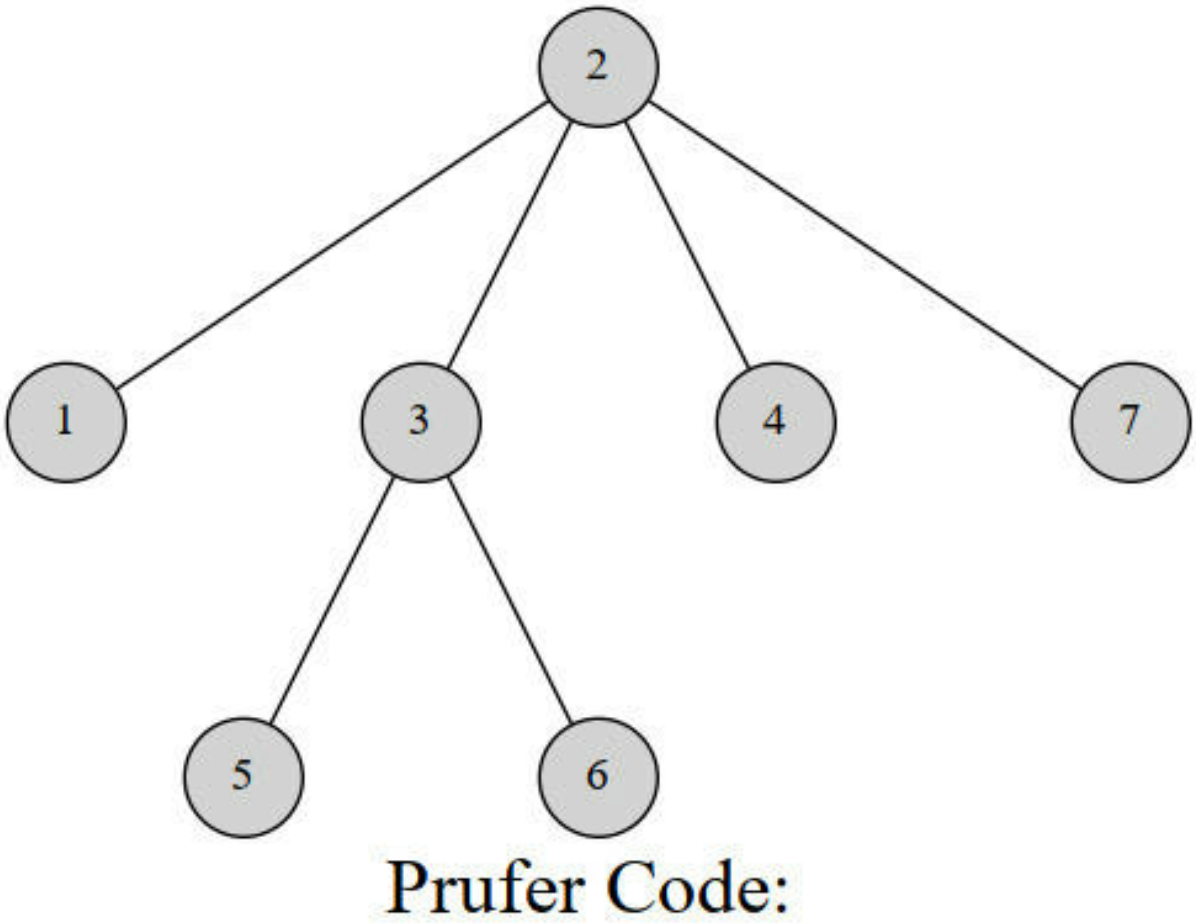
- Python

```python
# 结点是从 0 标号的
adj = [[]]


def pruefer_code():
    n = len(adj)
    leafs = set()
    degree = [0] * n
```

```
    killed = [False] * n
    for i in range(1, n):
        degree[i] = len(adj[i])
        if degree[i] == 1:
            leafs.intersection(i)
    code = [0] * (n - 2)
    for i in range(1, n - 2):
        leaf = leafs[0]
        leafs.pop()
        killed[leaf] = True
        for u in adj[leaf]:
            if killed[u] == False:
                v = u
        code[i] = v
        if degree[v] == 1:
            degree[v] = degree[v] - 1
            leafs.intersection(v)
    return code
```

例如，这是一棵 7 个结点的树的 Prüfer 序列构建过程：



最终的序列就是 $2, 2, 3, 3, 2$.

当然，也有一个线性的构造算法.

Prüfer 序列的线性构造算法

线性构造的本质就是维护一个指针指向我们将要删除的结点. 首先发现, 叶结点数是非严格单调递减的, 删去一个叶结点, 叶结点总数要么不变要么减 1.

于是我们考虑这样一个过程: 维护一个指针 p. 初始时 p 指向编号最小的叶结点. 同时我们维护每个结点的度数, 方便我们知道在删除结点的时候是否产生新的叶结点. 操作如下:

1. 删除 p 指向的结点, 并检查是否产生新的叶结点.
2. 如果产生新的叶结点, 假设编号为 x, 我们比较 p, x 的大小关系. 如果 $x > p$, 那么不做其他操作; 否则就立刻删除 x, 然后检查删除 x 后是否产生新的叶结点, 重复 2 步骤, 直到未产生新节点或者新节点的编号 $> p$.
3. 让指针 p 自增直到遇到一个未被删除叶结点为止;

正确性

循环上述操作 $n-2$ 次, 就完成了序列的构造. 接下来考虑算法的正确性.

p 是当前编号最小的叶结点, 若删除 p 后未产生叶结点, 我们就只能去寻找下一个叶结点; 若产生了叶结点 x:

- 如果 $x > p$, 则反正 p 往后扫描都会扫到它, 于是不做操作;
- 如果 $x < p$, 因为 p 原本就是编号最小的, 而 x 比 p 还小, 所以 x 就是当前编号最小的叶结点, 优先删除. 删除 x 继续这样的考虑直到没有更小的叶结点.

算法复杂度分析, 发现每条边最多被访问一次 (在删度数的时候), 而指针最多遍历每个结点一次, 因此复杂度是 $O(n)$ 的.

实现

- C++

```cpp
// 从原文摘的代码，同样以 0 为起点
vector<vector<int>> adj;
vector<int> parent;

void dfs(int v) {
  for (int u : adj[v]) {
    if (u != parent[v]) parent[u] = v, dfs(u);
  }
}

vector<int> pruefer_code() {
  int n = adj.size();
  parent.resize(n), parent[n - 1] = -1;
  dfs(n - 1);

  int ptr = -1;
  vector<int> degree(n);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1 && ptr == -1) ptr = i;
  }

  vector<int> code(n - 2);
  int leaf = ptr;
  for (int i = 0; i < n - 2; i++) {
    int next = parent[leaf];
    code[i] = next;
```

```
    if (--degree[next] == 1 && next < ptr) {
      leaf = next;
    } else {
      ptr++;
      while (degree[ptr] != 1) ptr++;
      leaf = ptr;
    }
  }
  return code;
}
```

- Python

```
# 同样以 0 为起点
adj = [[]]
parent = [0] * n


def dfs(v):
    for u in adj[v]:
        if u != parent[v]:
            parent[u] = v
            dfs(u)


def pruefer_code():
    n = len(adj)
    parent[n - 1] = -1
    dfs(n - 1)

    ptr = -1
    degree = [0] * n
    for i in range(0, n):
        degree[i] = len(adj[i])
        if degree[i] == 1 and ptr == -1:
            ptr = i

    code = [0] * (n - 2)
    leaf = ptr
    for i in range(0, n - 2):
        next = parent[leaf]
        code[i] = next
        if degree[next] == 1 and next < ptr:
            degree[next] = degree[next] - 1
            leaf = next
        else:
            ptr = ptr + 1
            while degree[ptr] != 1:
                ptr = ptr + 1
            leaf = ptr
    return code
```

Prüfer 序列的性质

1. 在构造完 Prüfer 序列后原树中会剩下两个结点，其中一个一定是编号最大的点 n.

2. 每个结点在序列中出现的次数是其度数减 1.（没有出现的就是叶结点）

用 Prüfer 序列重建树

重建树的方法是类似的. 根据 Prüfer 序列的性质，我们可以得到原树上每个点的度数. 然后也可以得到编号最小的叶结点，而这个结点一定与 Prüfer 序列的第一个数所对应的点连接. 然后我们同时将这两个结点的度数减一.

讲到这里也许你已经知道该怎么做了. 每次我们选择一个度数为 1 的编号最小的结点，与当前枚举到的 Prüfer 序列的点连接，然后同时减掉两个点的度. 到最后我们剩下两个度数为 1 的点，其中一个是结点 n. 把它们连上. 使用堆维护这个过程，在节点度数下降的过程中如果发现度数减到 1 就把这个结点添加到堆中，这样做的复杂度是 $O(n \log n)$ 的.

实现

```cpp
// 原文摘代码
vector<pair<int, int>> pruefer_decode(vector<int> const& code) {
  int n = code.size() + 2;
  vector<int> degree(n, 1);
  for (int i : code) degree[i]++;

  set<int> leaves;
  for (int i = 0; i < n; i++)
    if (degree[i] == 1) leaves.insert(i);

  vector<pair<int, int>> edges;
  for (int v : code) {
    int leaf = *leaves.begin();
    leaves.erase(leaves.begin());

    edges.emplace_back(leaf, v);
    if (--degree[v] == 1) leaves.insert(v);
  }
  edges.emplace_back(*leaves.begin(), n - 1);
  return edges;
}
```

线性时间重建树

同线性构造 Prüfer 序列的方法. 在删度数的时候会产生新的叶结点，于是判断这个叶结点与指针 p 的大小关系，如果更小就优先考虑它.

实现

```cpp
// 原文摘代码
vector<pair<int, int>> pruefer_decode(vector<int> const& code) {
  int n = code.size() + 2;
  vector<int> degree(n, 1);
  for (int i : code) degree[i]++;

  int ptr = 0;
  while (degree[ptr] != 1) ptr++;
  int leaf = ptr;

```

```
  vector<pair<int, int>> edges;
  for (int v : code) {
    edges.emplace_back(leaf, v);
    if (--degree[v] == 1 && v < ptr) {
      leaf = v;
    } else {
      ptr++;
      while (degree[ptr] != 1) ptr++;
      leaf = ptr;
    }
  }
  edges.emplace_back(leaf, n - 1);
  return edges;
}
```

通过这些过程其实可以理解，Prüfer 序列与带标号无根树建立了双射关系.

Cayley 公式 (Cayley's formula)

完全图 K_n 有 n^{n-2} 棵生成树.

怎么证明？方法很多，但是用 Prüfer 序列证是很简单的．任意一个长度为 $n-2$ 的值域 $[1,n]$ 的整数序列都可以通过 Prüfer 序列双射对应一个生成树，于是方案数就是 n^{n-2}.

图连通方案数

Prüfer 序列可能比你想得还强大．它能创造比 凯莱公式→11@p2597 更通用的公式．比如以下问题：

> 一个 n 个点 m 条边的带标号无向图有 k 个连通块．我们希望添加 $k-1$ 条边使得整个图连通．求方案数.

证明

设 s_i 表示第 i 个连通块内点的数量．我们考虑对 k 个连通块构造 Prüfer 序列．由于两个连通块之间的连接方法很多，这并不是普通的 Prüfer 序列．于是不妨假设 d_i 为第 i 个连通块的度数．由于度数之和是边数的两倍，于是 $\sum_{i=1}^{k} d_i = 2k-2$．则对于给定的 d 序列构造 Prüfer 序列的方案数是

$$\binom{k-2}{d_1-1, d_2-1, \cdots, d_k-1} = \frac{(k-2)!}{(d_1-1)!(d_2-1)!\cdots(d_k-1)!}$$

对于第 i 个连通块，它的连接方式有 $s_i^{d_i}$ 种，因此对于给定 d 序列使图连通的方案数是

$$\binom{k-2}{d_1-1, d_2-1, \cdots, d_k-1} \cdot \prod_{i=1}^{k} s_i^{d_i}$$

现在我们要枚举 d 序列，式子变成

$$\sum_{d_i \geq 1,\ \sum_{i=1}^{k} d_i = 2k-2} \binom{k-2}{d_1-1, d_2-1, \cdots, d_k-1} \cdot \prod_{i=1}^{k} s_i^{d_i}$$

好的这是一个非常不喜闻乐见的式子．但是别慌！我们有多元二项式定理：

$$(x_1 + \ldots + x_m)^p = \sum_{c_i \geq 0,\ \sum_{i=1}^{m} c_i = p} \binom{p}{c_1, c_2, \cdots, c_m} \cdot \prod_{i=1}^{m} x_i^{c_i}$$

那么我们对原式做一下换元，设 $e_i = d_i - 1$，显然 $\sum_{i=1}^{k} e_i = k-2$，于是原式变成

$$\sum_{e_i \geq 0,\ \sum_{i=1}^{k} e_i = k-2} \binom{k-2}{e_1, e_2, \cdots, e_k} \cdot \prod_{i=1}^{k} s_i^{e_i+1}$$

化简得到

$$(s_1+s_2+\cdots+s_k)^{k-2}\cdot\prod_{i=1}^{k}s_i$$

即

$$n^{k-2}\cdot\prod_{i=1}^{k}s_i$$

为答案.

习题

- Luogu P6086【模板】Prüfer 序列[2]（模板题）
- Luogu P11039【MX-X3-T6】「RiOI-4」TECHNOPOLIS 2085[3]
- UVa #10843 - Anne's game[4]
- Timus #1069 - Prufer Code[5]
- Codeforces - Clues[6]
- Topcoder - TheCitiesAndRoadsDivTwo[7]

外部参考资料

[1] e-maxx Prüfer Code

[2] Luogu P6086【模板】Prüfer 序列

[3] Luogu P11039【MX-X3-T6】「RiOI-4」TECHNOPOLIS 2085

[4] UVa #10843 - Anne's game

[5] Timus #1069 - Prufer Code

[6] Codeforces - Clues

[7] Topcoder - TheCitiesAndRoadsDivTwo

11.26 矩阵树定理

Authors: pw384, s0cks5, Watersail2005, Xeonacid

矩阵树定理解决了一张图的生成树个数计数问题.

本篇记号声明

本篇中的图，无论无向还是有向，都允许重边，但是默认没有自环.

有自环的情形

自环不影响生成树的个数，计算 Laplace 矩阵时不计入自环即可．但根据 BEST 定理统计欧拉回路个数时，度数需计入自环.

无向图情况

设 G 是一个有 n 个顶点的无向图．定义度数矩阵 $D(G)$ 为

$$D_{ii}(G)=\deg(i),\ D_{ij}=0,\ i\neq j.$$

设 $\#e(i,j)$ 为点 i 与点 j 相连的边数，并定义邻接矩阵 A 为

$$A_{ij}(G)=A_{ji}(G)=\#e(i,j),\ i\neq j.$$

定义 Laplace 矩阵（亦称 Kirchhoff 矩阵）L 为

$$L(G)=D(G)-A(G).$$

记图 G 的所有生成树个数为 $t(G)$.

有向图情况

设 G 是一个有 n 个顶点的有向图．定义出度矩阵 $D^{\text{out}}(G)$ 为

$$D^{\text{out}}{}_{ii}(G) = \deg^{\text{out}}(i),\ D^{\text{out}}{}_{ij} = 0,\ i \neq j.$$

类似地定义入度矩阵 $D^{\text{in}}(G)$.

设 $\#e(i,j)$ 为点 i 指向点 j 的有向边数，并定义邻接矩阵 A 为

$$A_{ij}(G) = \#e(i,j),\ i \neq j.$$

定义出度 Laplace 矩阵 L^{out} 为

$$L^{\text{out}}(G) = D^{\text{out}}(G) - A(G).$$

定义入度 Laplace 矩阵 L^{in} 为

$$L^{\text{in}}(G) = D^{\text{in}}(G) - A(G).$$

记图 G 的以 k 为根的所有根向树形图个数为 $t^{\text{root}}(G,k)$．所谓根向树形图，是说这张图的基图是一棵树，所有的边全部指向父亲.

记图 G 的以 k 为根的所有叶向树形图个数为 $t^{\text{leaf}}(G,k)$．所谓叶向树形图，是说这张图的基图是一棵树，所有的边全部指向儿子.

定理叙述

矩阵树定理具有多种形式.

定义 $[n] = \{1,2,\cdots,n\}$，矩阵 A 的子矩阵 $A_{S,T}$ 为选取 $A_{i,j}\quad (i \in S, j \in T)$ 的元素得到的子矩阵.

定理 1（矩阵树定理，无向图，行列式形式）

对于无向图 G 和任意的 k，都有

$$t(G) = \det L(G)_{[n]\setminus\{k\},[n]\setminus\{k\}}.$$

也就是说，无向图的 Laplace 矩阵所有 $n-1$ 阶主子式都相等，且都等于图的生成树的个数.

推论 1（矩阵树定理，无向图，特征值形式）

设 $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{n-1} \geq \lambda_n = 0$ 为 $L(G)$ 的 n 个特征值，那么有

$$t(G) = \frac{1}{n}\lambda_1\lambda_2\cdots\lambda_{n-1}.$$

定理 2（矩阵树定理，有向图根向树，行列式形式）

对于有向图 G 和任意的 k，都有

$$t^{\text{root}}(G,k) = \det L^{\text{out}}(G)_{[n]\setminus\{k\},[n]\setminus\{k\}}.$$

也就是说，有向图的出度 Laplace 矩阵删去第 k 行第 k 列得到的主子式等于以 k 为根的根向树形图的个数.

因此如果要统计一张图所有的根向树形图，只要枚举所有的根 k 并对 $t^{\text{root}}(G,k)$ 求和即可.

定理 3（矩阵树定理，有向图叶向树，行列式形式）

对于有向图 G 和任意的 k，都有

$$t^{\text{leaf}}(G,k) = \det L^{\text{in}}(G)_{[n]\setminus\{k\},[n]\setminus\{k\}}.$$

也就是说，有向图的入度 Laplace 矩阵删去第 k 行第 k 列得到的主子式等于以 k 为根的叶向树形图的个数.

因此如果要统计一张图所有的叶向树形图，只要枚举所有的根 k 并对 $t^{\text{leaf}}(G,k)$ 求和即可.

注

根向树形图也被称为内向树形图，但因为计算内向树形图用的是出度，为了不引起 in 和 out 的混淆，所以采用了根向这一说法.

定理证明

注意到上述定理的形式极为相似，这里给出一种统一的证明方式，并且将之前的结论拓展到带权的图上.

证明的大致思路如下：

- 首先，所有情形都可以转化为计数有向图上根向树形图的情形；
- 利用矩阵语言给出选出的若干边可以构成根向树形图的充要条件；
- 将选边的操作利用 Cauchy–Binet 公式和 Laplace 矩阵的行列式联系起来；
- 最后，将行列式形式的结论转化为特征值形式的结论.

引理：Cauchy–Binet 公式

引理 1（Cauchy–Binet）

给定 $n\times m$ 的矩阵 A 和 $m\times n$ 的矩阵 B，则有

$$\det(AB)=\sum_{S\subset[m];\ |S|=n}\det A_{[n],S}\det B_{S,[n]},$$

这里求和记号的含义是，S 取遍所有 $[m]$ 中大小为 n 的子集. 如果 $n>m$，必然有 $\det(AB)=0$.

证明（组合视角）

参考「NOI2021」路径交点[1] 的模型，首先考虑行列式的如下组合意义. 对于 $n\times n$ 阶矩阵 C，建立有向无环图 $G=(V,E)$. 其中，顶点集为 $V=[2]\times[n]\subset\mathbb{R}^2$，亦即平面上的两列点. 记左侧一列点为 $L=\{l_i=(1,i):i\in[n]\}$，右侧的一列点为 $R=\{r_i=(2,i):i\in[n]\}$；而有向边集为 $E=\{(l_i,r_j):i,j\in[n]\}$，并赋有边权 $w(l_i,r_j)=C_{i,j}$. 在图中，称大小为 n 的边的子集 $E^\sigma\subset E$ 为一个路径组，如果它的起点互不相同，且终点也互不相同. 显然，路径组 E^σ 和 $[n]$ 上的置换 σ 可以一一对应. 注意到，如果将一个路径组在平面上画出，这些边之间可能会两两相交，而这些交点的数目（计重数）就等于 σ 的逆序数. 这是因为边 $(l_i,r_{\sigma(i)})$ 和边 $(l_j,r_{\sigma(j)})$ 相交，当且仅当 $(i-j)(\sigma(i)-\sigma(j))<0$，即这是一个逆序对. 为方便，称对应置换的逆序数的奇偶性，亦即该路径组交点个数的奇偶性，为该路径组的奇偶性. 所以，如果将这些路径组按照权重计数，且用偶数交点的路径组数减去奇数交点的路径组数，就会得到行列式的 Leibniz 展开：

$$\det(C)=\sum_{\sigma\in S_n}\operatorname{sgn}(\sigma)\prod_{i\in[n]}C_{i,\sigma(i)},$$

其中，S_n 为 $[n]$ 上的置换群，而 $\operatorname{sgn}(\sigma)$ 为置换 σ 的符号（当逆序数为偶数时，它等于 1；当逆序数为奇数时，它等于 -1）.

在理解行列式的组合意义后，可以利用如下的组合模型证明 Cauchy–Binet 公式. 对于 $n\times m$ 阶矩阵 A 和 $m\times n$ 阶矩阵 B，建立有向无环图 $G=(V,E)$. 其中，顶点集为 $V=L\cup D\cup R$，这里，$L=\{l_i=(1,i):i\in[n]\}$，$D=\{d_i=(2,i):i\in[m]\}$ 和 $R=\{r_i=(3,i):i\in[n]\}$；而有向边集为 $E=E_L\cup E_R$，其中，$E_L=\{(l_i,d_j):i\in[n],j\in[m]\}$ 和 $E_R=\{(d_j,r_i):j\in[m],i\in[n]\}$，分别赋以边权 $w(l_i,d_j)=A_{i,j}$ 和 $w(d_j,r_i)=B_{j,i}$. 同样考虑自 L 经 D 到 R 的路径组（路径间两两不共用顶点），按

照权重计数，并用偶数交点的路径组数减去奇数交点的路径组数．下面说明，Cauchy–Binet 公式的左右两侧分别用两种方式计算了这一数目．

对于左侧，基于上面描述的图 G，建立新图 G'，其顶点集为 $V'=L\cup R$，边集为 $E'=\{(l_i,r_j):i,j\in[n]\}$，且对于边 (l_i,r_j) 赋以边权 $\sum_{k\in[m]}A_{i,k}B_{k,j}$，即在原图 G 中自 l_i 到 r_j 的简单路径的加权计数．这一边权正是 $(AB)_{i,j}$．这相当于把上述的三层图简化成了两层图．但是，两层图 G' 中的路径组（按权重计）并非和三层图 G 中的路径组一一对应．由于在两层图中，每个路径都对应三层图中若干条简单路径，在对两层图进行路径组的计数时，需要将权重相乘，这相当于对它们对应的三层图中的路径集合两两组合，这必然会造成出现共用中间经停点的情形．但是，这些共用中间经停点的路径对并不会对最后的答案有贡献，因为对于 $i_1<i_2$ 和 $j_1<j_2$ 和任意中间点 d，都存在两种简单路径对 $(l_{i_1}\to d\to r_{j_1},l_{i_2}\to d\to r_{j_2})$ 和 $(l_{i_1}\to d\to r_{j_2},l_{i_2}\to d\to r_{j_1})$，但是这两组路径在三层图中的交点数目奇偶性必然相反，因为如果只看起点和终点，两组路径交换了终点．所以，这些共用中间经停点的路径在简化后的两层图计数时，贡献会两两抵消．对于剩下的情形，如果给定两条路径的起点和终点，那么无论中间的点如何选取（只要不选择同一个点），则这两条路径的交点个数的奇偶性不会变．故而，G' 中每一个路径组对应的所有原图 G 中的路径组都具有相同的奇偶性．因而，$\det(AB)$ 提供了前文所述路径组数差值的一种计算方式．

对于右侧，它相当于枚举了所有可能的中间点的组合．给定任何中间点集合 $S\subset D=[m]$ 且 $|S|=n$，分别考虑自 L 到 S 的路径组和自 S 到 R 的路径组，可以连接得到 L 到 R 的路径组，且前两个路径组对应的置换的复合就等于之后的路径组对应的置换，故而前两个路径组的奇偶性的乘积等于之后的路径组的奇偶性．所以，所有中间点集合为 S 的路径组的计数的差值正等于自 L 到 S 的路径组的计数的差值和自 S 到 R 的路径组的计数的差值的乘积．对所有可能的 S 求和，即得到右式，故而它正是前文所述路径组数差值．

证明（代数视角）

上述组合证明其实可以逐字逐句地翻译成代数证明．这里转而提供另一种技巧性较强的代数证明，但用到了几个常见结论．当 $m<n$ 时，行列式为零，因为

$$\operatorname{rank}(AB)\le\min\{\operatorname{rank}(A),\operatorname{rank}(B)\}\le m<n.$$

当 $m=n$ 时，Cauchy–Binet 公式就是，方阵的积的行列式等于方阵的行列式的积．

当 $m>n$ 时，注意到

$$x^{m-n}\det(xI_n+AB)=\det(xI_m+BA).$$

又已知结论，$\det(xI_n+C)$ 中 x^{n-k} 的系数是 C 的所有 k 阶主子式的和．故而，比较上式中两侧系数，有

$$\det(AB)=\sum_{S\subset[m];\ |S|=n}\det(BA)_{S,S}=\sum_{S\subset[m];\ |S|=n}\det B_{S,[n]}\det A_{[n],S}=\sum_{S\subset[m];\ |S|=n}\det A_{[n],S}\det B_{S,[n]}$$

这里，第二个等号用到了 $m=n$ 的情形的结论．

用关联矩阵刻画图的结构

对于有向图 $G=(V,E)$，顶点数为 n，边数为 m，且边 e 赋有边权 $w(e)$．由此，可以定义 $m\times n$ 阶出度关联矩阵

$$M^{\text{out}}{}_{ij}=\begin{cases}\sqrt{w(e_i)},&\exists u(e_i=(v_j,u)),\\0,&\text{otherwise},\end{cases}$$

和 $m\times n$ 阶入度关联矩阵

$$M^{\text{in}}{}_{ij} = \begin{cases} \sqrt{w(e_i)}, \exists u(e_i = (u, v_j)), \\ 0, \qquad \text{otherwise.} \end{cases}$$

它们每行都记录了一条边：出度关联矩阵 M^{out} 记录了边的起点，入度关联矩阵 M^{in} 记录了边的终点.

简单计算可知

$$D^{\text{out}}(G) = (M^{\text{out}})^T M^{\text{out}},\ A(G) = (M^{\text{out}})^T M^{\text{in}},\ D^{\text{in}}(G) = (M^{\text{in}})^T M^{\text{in}}.$$

进而有

$$L^{\text{out}}(G) = (M^{\text{out}})^T (M^{\text{out}} - M^{\text{in}}),\ L^{\text{in}}(G) = (M^{\text{in}} - M^{\text{out}})^T M^{\text{in}}.$$

前文的 Cauchy–Binet 公式表明，Laplace 矩阵的主子式其实是一系列子结构的和. 每个子结构都反映了对应的子图的性质.

引理 2

对于顶点子集 $W \subseteq V$ 和边子集 $S \subseteq E$，若它们满足 $|W| = |S| \leq n$，则当子图 $T = (V, S)$ 是一个以 $V \setminus W$ 为根的根向森林时，对应的算式

$$\det(M^{\text{out}}{}_{S,W}) \det(M^{\text{out}}{}_{S,W} - M^{\text{in}}{}_{S,W})$$

等于 $\prod_{e \in S} w(e)$，记作 $w(T)$；否则，该式等于零.

证明

不妨设 $w(e) = 1$. 这是根据行列式的多重线性，每个行列式的每行都可以提取因子 $\sqrt{w(e)}$，这些因子的乘积为 $w(T)$.

首先分析两个因子等于零的条件. 前一个因子 $\det(M^{\text{out}}{}_{S,W})$ 每行至多一个不为零的数字，即 $+1$. 如果有任何一行全为零，或者有两行的 $+1$ 位于同一列，则该行列式必然为零. 所以，该行列式不为零，当且仅当每行每列都恰好一个 $+1$，亦即 S 中每条边的起点都在 W 中，且 W 中每个点都恰好是 S 中一条边的起点. 已知 T 成为以 $V \setminus W$ 为根的根向森林，一个必要条件就是除了根之外，所有顶点有且只有一个父结点，这必然使得该因子不为零；但反过来并不一定成立，因为不能保证不存在环，所以还需要考察第二个因子. 注意，S 的终点未必在 W 中.

假定前一个因子不为零，则此时子图 T 成为根向森林，当且仅当 T 中没有环. 此时，后一项 $\det(M^{\text{out}}{}_{S,W} - M^{\text{in}}{}_{S,W})$ 每行中都有一个 $+1$，但可能有一个或零个 -1. 对于终点也在 W 中的边，如果 e_i 的终点是 e_j 的起点，则将 e_i 对应的行加上 e_j 对应的行，可以消去 e_i 行中的 -1. 可以想象，此时该行描述的是 e_i 和 e_j 首尾相接的简单路径. 如果该行出现了新的 -1，那么说明 e_j 的终点也在 W 内，-1 的位置就是 e_j 的终点，于是，可以继续找到以 e_j 的终点为起点的边，再次加到该行上. 这样的边总是存在的，因为上一段论述说明，W 中每个点都恰好是 S 中一条边的起点. 这一过程一直持续到该行不再出现 -1 为止，相当于不断添加新的边到简单路径 $e_i \to e_j \to \cdots \to e_k$ 中. 此时，如果该行只剩下一个 $+1$，那么说明 e_k 的终点不在所选顶点 W 中，过程终止；如果上次加入的边恰巧抵消了现有的 $+1$，即该行只剩下零，那么说明新边 e_k 的终点就是最开始的边 e_i 的起点，即出现了一个环；如果这一过程始终不终止，那么路径进入了一个环，环上的边对应的行同样会变为零. 所以，没有环的充要条件是该行列式经上述操作可以变形成每行都恰好只有一个 $+1$ 的形式. 由于这些 $+1$ 的位置是各行对应边的起点，此时得到的矩阵实际上就是 $M^{\text{out}}{}_{S,W}$，故两者的行列式相等.

综上所述，如果 T 不是根向森林，则要么 $\det(M^{\text{out}}{}_{S,W}) = 0$，要么 $\det(M^{\text{out}}{}_{S,W} - M^{\text{in}}{}_{S,W}) = 0$；否则，两者均不为零，且乘积等于 $\left(\det(M^{\text{out}}{}_{S,W})\right)^2 = 1$.

带权有向图的矩阵树定理

现在可以证明本文的主要结果. 前文所述矩阵树定理均为该定理的特殊情形.

定理 4（矩阵树定理，带权有向图根向树，行列式形式）

对于任意的 k，都有

$$\sum_{T\in\mathcal{T}^{\text{root}}(G,k)} w(T)=\det L^{\text{out}}(G)_{[n]\setminus\{k\},[n]\setminus\{k\}}.$$

这里，$\mathcal{T}^{\text{root}}(G,k)$ 是 G 的以 k 为根的根向树形图的集合.

证明

记 $W=[n]\setminus\{k\}$ 为除去 k 点外的剩余顶点的集合. 那么，根据 Cauchy–Binet 公式，右式可以写作

$$\det L^{\text{out}}(G)_{W,W}=\sum_{S\subset[m];\ |S|=n-1}\det({M^{\text{out}}}_{S,W})\det({M^{\text{out}}}_{S,W}-{M^{\text{in}}}_{S,W}).$$

遍历所有的 S，由引理 2，当且仅当 $T=(V,S)$ 构成一个以 $V\setminus W=\{k\}$ 为根的根向森林时，亦即 T 是一个以 k 为根的根向树形图时，右侧累加一个 $w(T)$.

当 $w(e)=1$ 时，每个树的权值都是 1，则左侧就是所有树的计数，即 $t^{\text{root}}(G,k)$，这就得到定理 2. 类比上文，可以将结论直接推广于叶向树形图，这就得到定理 3. 最后，要得到无向图上的生成树计数，可以应用如下推论.

推论 2（矩阵树定理，带权无向图，行列式形式）

对于无向图 G 和任意的 k，都有

$$\sum_{T\in\mathcal{T}(G)} w(T)=\det L(G)_{[n]\setminus\{k\},[n]\setminus\{k\}}.$$

这里，$\mathcal{T}(G)$ 是 G 的生成树的集合. 这也说明，$L(G)$ 的所有 $(n-1)$ 阶主子式都相等.

证明

对于无向图 $G=(V,E)$，可以构建有向图 $G'=(V,E')$，其中，$E'=\{(v_i,v_j):(v_i,v_j)\in E\}\cup\{(v_j,v_i):(v_i,v_j)\in E\}$，即每条 G 中的无向边都拆成有向图中方向相反的两条有向边. 任取 k，则 G' 中以 k 为根的根向树形图和 G 中的生成树一一对应. 由前者向后者，只需要移除边的定向和根的选取；由后者向前者，只需要从选定的根 k 开始逐边选取根向作为边的定向. 所以，此时有

$$\sum_{T\in\mathcal{T}(G)} w(T)=\sum_{T\in\mathcal{T}^{\text{root}}(G',k)} w(T)=\det L^{\text{out}}(G')_{[n]\setminus\{k\},[n]\setminus\{k\}}=\det L(G)_{[n]\setminus\{k\},[n]\setminus\{k\}}.$$

此处用到了结论 $L^{\text{out}}(G')=L(G)$，这容易直接验证.

特征值形式

仍然首先考虑有向图上的结论.

定理 5

对于有向图 G，定义多元多项式

$$\chi(x_1,\cdots,x_n)=\det(\operatorname{diag}(x_1,\cdots,x_n)-L^{\text{out}}(G)).$$

这里，$\operatorname{diag}(x_1,\cdots,x_n)$ 是指以 $x_1,\cdots,x_n$ 为对角线元素的对角矩阵. 那么，

$$(-1)^{n-r}[x_{k_1}\cdots x_{k_r}]\chi(x_1,\cdots,x_n)$$

就等于 G 的以 $\{k_1,\cdots,k_r\}$ 为根的根向森林的（带权的）计数.

证明

仿照定理 4 的证明，注意到如果令 $W=[n]\setminus\{k_1,\cdots,k_r\}$，那么，定理中的式子就是 $\det L^{\text{out}}(G)_{W,W}$（这一点不妨直接观察行列式的 Leibniz 展开式）．根据 Cauchy–Binet 公式，它等于

$$\det L^{\text{out}}(G)_{W,W}=\sum_{S\subset[m];\ |S|=n-r}\det({M^{\text{out}}}_{S,W})\det({M^{\text{out}}}_{S,W}-{M^{\text{in}}}_{S,W}).$$

遍历所有的 S，由引理 2，当且仅当 $T=(V,S)$ 构成一个以 $V\setminus W=\{k_1,\cdots,k_r\}$ 为根的根向森林时，右侧累加一个 $w(T)$.

将 x 代入所有的未知元，得到 Laplace 矩阵的特征多项式

$$P(x)=\det(xI-L^{\text{out}}(G))=\chi(x,\cdots,x).$$

引理 3

Laplace 矩阵 $L^{\text{out}}(G)$ 至少有一个特征值为零.

证明

只要证明它的行列式为零即可．仿照定理 4 和 5 的证明，取 $W=[n]$（即根的集合为空），则这个行列式的值应该等于有零棵树的根向森林的数目．这并不存在，所以该行列式等于零.

推论 3

对于有向图 G，所有由 k 棵树构成的根向森林的权值的总和等于系数

$$(-1)^{n-k}[x^k]P(x).$$

证明

对所有可能的 k 个根的选择求和即可.

定义 k- 生成森林 是图的一个生成子图，使得这个子图有 k 个连通分量且无环.

推论 4

记无向图 G 的 k- 生成森林 的集合为 $\mathcal{T}_k(G)$，$P(x)=\det(xI-L(G))$ 为其 Laplace 矩阵的特征多项式，则

$$\sum_{T\in\mathcal{T}_k(G)}w(T)Q(T)=(-1)^{n-k}[x^k]P(x).$$

这里，$Q(T)$ 为森林 T 中每个连通分量的顶点数目的乘积．特别地，当 $k=1$ 时，有 $Q(T)=n$，故而

$$n\sum_{T\in\mathcal{T}(G)}w(T)=\lambda_1\lambda_2\cdots\lambda_{n-1}.$$

证明

仿照推论 2 的证明，可以直接利用推论 3 的结论．有向图中每一个由 k 棵树构成的根向森林都对应一个无向图中的 k- 生成森林．但是，由于每个 k- 生成森林 T 有 $Q(T)$ 种选择根的方法，它会出现在 $Q(T)$ 个有向图的根向森林中.

应用

Cayley 公式

推论 5（Cayley）

大小为 n 的带标号的无根树有 n^{n-2} 个.

证明

等价地，只要求得 n 个顶点的完全图的生成树的数目为 n^{n-2} 即可．为此，写出 Laplace 矩阵

$$L(G)=\begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix}_{n\times n}.$$

计算它的任意主子式，有

$$\det(nI_{n-1}-\mathbf{1}\mathbf{1}^T)=n^{n-1}\det(I_{n-1}-n^{-1}\mathbf{1}\mathbf{1}^T)=n^{n-1}(1-n^{-1}\mathbf{1}^T\mathbf{1})=n^{n-1}(1-(n-1)/n)=n^{n-2}.$$

应用定理 1 即得到结论.

BEST 定理

前置知识：欧拉图→11.17@p2871

这一定理将有向欧拉图中欧拉回路的数目和该图的根向树形图的数目联系起来，从而解决了有向图中的欧拉回路的计数问题．注意，任意无向图中的欧拉回路的计数问题是 #P 完全的.

在实现该算法时，应当首先判定给定图是否是欧拉图，移除所有零度顶点，然后建图计算根向树形图的个数，并由 BEST 定理得到欧拉回路的计数．注意，如果所求欧拉回路个数要求以给定点作为起点，需要将答案再乘上该点出度，相当于枚举回路中首条边.

在证明 BEST 定理之前，需要知道如下结论.

性质（有向图具有欧拉回路的判定）

一个有向图具有欧拉回路，当且仅当非零度顶点是强连通的，且所有顶点的出度和入度相等.

对于欧拉图，因为出度和入度相等，可以将它们略去上标，记作 $\deg(v)$．BEST 定理可以叙述如下.

定理 6（BEST 定理）

设 G 是有向欧拉图，k 为任意顶点，那么 G 的不同欧拉回路总数 $\mathrm{ec}(G)$ 是

$$\mathrm{ec}(G)=t^{\mathrm{root}}(G,k)\prod_{v\in V}(\deg(v)-1)!.$$

这也说明，对欧拉图 G 的任意两个顶点 k,k'，都有 $t^{\mathrm{root}}(G,k)=t^{\mathrm{root}}(G,k')$.

证明

证明的大致思路是建立以 k 为起点的欧拉回路和以 k 为根的根向树形图以及各个顶点处出边的排列的对应关系．在指定欧拉回路自 k 出发后，需要证明的计数应当等于

$$\deg(k)\mathrm{ec}(G)=t^{\mathrm{root}}(G,k)\deg(k)!\prod_{v\neq k}(\deg(v)-1)!.$$

这一计数的组合含义对应的构造如下．对于起点为 k 的欧拉回路，根据回路中每条边的出现顺序，可以构造出

- 一个以 k 为根的根向树形图，由所有非根顶点处的最后一条出边组成，即 $t^{\text{root}}(G,k)$，
- 根 k 处所有出边的排列顺序，即 $\deg(k)!$，和
- 非根顶点 $v \neq k$ 处除去最后一条出边之外的其他所有出边的排列顺序，即 $(\deg(v)-1)!$.

下面说明，这样的构造得到的映射是双射.

一方面，给定欧拉回路，要证明所有非根顶点处的最后一条出边组成了一个根向树形图. 根据构造，树中每个非根顶点的确只有一条出边，所以只需要证明这些出边不会成环. 注意到，如果将所有顶点根据它在欧拉回路最后一次出现的顺序排序，那么非根顶点的最后一次出边必然指向顺序严格更靠后的顶点. 如果存在环，那么环中就有一个顺序最靠后的顶点，因为它在环中，所以它指向了一个顺序并不靠后的点，这与上文矛盾. 所以，非根顶点的最后一次出边必然构成根向树形图.

另一方面，给定任意根向树形图和其余出边的排列顺序，可以复原出一条欧拉回路，使得该欧拉回路经上述构造后可以得到给定的根向树形图和其余出边的排列顺序. 对此，只需要从根 k 出发，每当到达一个顶点时，都根据给定的该顶点的出边排列顺序，选择顺序最靠前的、尚未经过的出边作为欧拉回路中本次的出边；如果该顶点处的排列中所有出边都已经经过了，就选择根向树形图中该顶点的出边作为欧拉回路中本次的出边. 因为图是欧拉图，每个顶点的入度都等于出度，所以，这一过程不会在非根顶点处终止，即所得路径的确是回路. 要证明所得路径是合法的欧拉回路，只需要证明这一过程能够遍历所有边就可以.

如果不能，则必然有某个顶点 v 的某个出边没有遍历到. 考察顶点 v. 顶点 v 不能是根，因为最后会终止在根，如果根仍有出边剩余，这与过程终止矛盾. 所以，v 必然不是根. 根据前文描述的过程，只要非根顶点 v 有任何出边剩余，那么非根顶点在树中的出边 e 必然剩余. 记 $e=(v,u)$. 因为 u 的某个入边没有遍历到，根据 u 的出度等于入度，必然有 u 的某条出边没有遍历到. 然后，可以类似地考察顶点 u. 这些推理将考察的顶点从 v 移动到了 u，即沿着根向树形图向树的根移动了一步. 可以归纳地证明，此时必有根 k 的某个出边没有遍历到. 前文已经说明这不可能，故得到矛盾. 这说明，上一段所得路径的确是合法的欧拉回路.

可以验证这些映射都是单射，则必然同为双射. 原命题得证.

实现

根据图写出 Laplace 矩阵，删去一行一列，求所得矩阵的行列式即可. 求行列式可以使用 Gauss 消元法.

例如，一个正方形图的生成树个数

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & -1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

$$\begin{vmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{vmatrix} = 4$$

可以用 Gauss 消元解决，时间复杂度为 $O(n^3)$.

实现

```cpp
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <iostream>
```

```cpp
using namespace std;
constexpr double eps = 1e-7;

struct matrix {
  static constexpr int MAXN = 20;
  int n, m;
  double mat[MAXN][MAXN];

  matrix() { memset(mat, 0, sizeof(mat)); }

  void print() {
    cout << "MATRIX " << n << " " << m << endl;
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        cout << mat[i][j] << "\t";
      }
      cout << endl;
    }
  }

  void random(int n) {
    this->n = n;
    this->m = n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) mat[i][j] = rand() % 100;
  }

  void initSquare() {
    this->n = 4;
    this->m = 4;
    memset(mat, 0, sizeof(mat));
    mat[0][1] = mat[0][3] = -1;
    mat[1][0] = mat[1][2] = -1;
    mat[2][1] = mat[2][3] = -1;
    mat[3][0] = mat[3][2] = -1;
    mat[0][0] = mat[1][1] = mat[2][2] = mat[3][3] = 2;
    this->n--;  // 去一行
    this->m--;  // 去一列
  }

  double gauss() {
    double ans = 1;
    for (int i = 0; i < n; i++) {
      int sid = i;
      for (int j = i + 1; j < n; j++)
        if (abs(mat[j][i]) > abs(mat[sid][i])) sid = j;
      if (abs(mat[sid][i]) <= eps) return 0;
      if (sid != i) {
        for (int j = 0; j < n; j++) swap(mat[sid][j], mat[i][j]);
        ans = -ans;
      }
      for (int j = i + 1; j < n; j++) {
```

```
58          double ratio = mat[j][i] / mat[i][i];
59          for (int k = 0; k < n; k++) {
60            mat[j][k] -= mat[i][k] * ratio;
61          }
62        }
63      }
64      for (int i = 0; i < n; i++) ans *= mat[i][i];
65      return ans;
66    }
67 };
68
69 int main() {
70   srand(1);
71   matrix T;
72   // T.random(2);
73   T.initSquare();
74   T.print();
75   double ans = T.gauss();
76   T.print();
77   cout << ans << endl;
78 }
```

例题

例题 1：「HEOI2015」小 Z 的房间[2]

解 矩阵树定理的裸题．将每个空房间看作一个结点，根据输入的信息建图，得到 Laplace 矩阵后，任意删掉 L 的第 i 行第 i 列，求这个子式的行列式即可．求行列式的方法就是高斯消元成上三角阵然后算对角线积．另外本题需要在 $\mathbb{Z}/10^9\mathbb{Z}$ 上进行高斯消元，采用辗转相除法即可．

例题 2：「FJOI2007」轮状病毒[3]

解 本题的解法很多，这里用矩阵树定理是最直接的解法．当输入为 n 时，容易写出其 $n+1$ 阶的 Laplace 矩阵为：

$$L_n = \begin{bmatrix} n & -1 & -1 & -1 & \cdots & -1 & -1 \\ -1 & 3 & -1 & 0 & \cdots & 0 & -1 \\ -1 & -1 & 3 & -1 & \cdots & 0 & 0 \\ -1 & 0 & -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & 0 & 0 & 0 & \cdots & 3 & -1 \\ -1 & -1 & 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_{n+1}$$

求出它的 n 阶子式的行列式即可，剩下的只有高精度计算了．

例题 2+

将例题 2 的数据加强，要求 $n \leq 100000$，但是答案对 1000007 取模．（本题求解需要一些线性代数知识）

解 推导递推式后利用矩阵快速幂即可求得．

推导递推式的过程：

注意到 L_n 删掉第 1 行第 1 列以后得到的矩阵很有规律，因此其实就是在求矩阵

$$M_n = \begin{bmatrix} 3 & -1 & 0 & \cdots & 0 & -1 \\ -1 & 3 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 3 & -1 \\ -1 & 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_n$$

的行列式．对 M_n 的行列式按第一列展开，得到

$$\det M_n = 3\det \begin{bmatrix} 3 & -1 & \cdots & 0 & 0 \\ -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 3 & -1 \\ 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_{n-1} + \det \begin{bmatrix} -1 & 0 & \cdots & 0 & -1 \\ -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 3 & -1 \\ 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_{n-1} + (-1)^n \det \begin{bmatrix} -1 & 0 & \cdots & 0 & -1 \\ 3 & -1 & \cdots & 0 & 0 \\ -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 3 & -1 \end{bmatrix}_{n-1}$$

上述三个矩阵的行列式记为 $d_{n-1}, a_{n-1}, b_{n-1}$.

注意到 d_n 是三对角行列式，采用类似的展开的方法可以得到 d_n 具有递推公式 $d_n = 3d_{n-1} - d_{n-2}$. 类似地，采用展开的方法可以得到 $a_{n-1} = -d_{n-2} - 1$，以及 $(-1)^n b_{n-1} = -d_{n-2} - 1$.

将这些递推公式代入上式，得到：

$$\det M_n = 3d_{n-1} - 2d_{n-2} - 2$$

$$d_n = 3d_{n-1} - d_{n-2}$$

于是猜测 $\det M_n$ 也是非齐次的二阶线性递推．采用待定系数法可以得到最终的递推公式为

$$\det M_n = 3\det M_{n-1} - \det M_{n-2} + 2$$

改写成 $(\det M_n + 2) = 3(\det M_{n-1} + 2) - (\det M_{n-2} + 2)$ 后，采用矩阵快速幂即可求出答案.

例题 3：「BZOJ3659」WHICH DREAMED IT[4]

解 本题是 BEST 定理的直接应用，但是要注意，由于题目规定「两种完成任务的方式算作不同当且仅当使用钥匙的顺序不同」，对每个欧拉回路，1 号房间可以沿着任意一条出边出发，从而答案还要乘以 1 号房间的出度.

例题 4：「联合省选 2020 A」作业题[5]

解 首先需要用莫比乌斯反演转化成计算所有生成树的边权和，因为与本文关系不大所以略去.

将行列式的项写成 $w_i x + 1$，最后答案是行列式的一次项系数，因为答案实际上是钦定一条边之后的生成树个数 $\times$ 这条边的边权之和，那么被乘上一次项系数的边就是被钦定的边．此时可以把高于一次的项忽略掉，复杂度 $O(n^3)$.

「北京省选集训 2019」生成树计数[6] 是较为一般化的情况：计算生成树权值之和的 k 次方之和，用类似方法构造行列式的项即可，具体见洛谷题解.

例题 5：AGC051D C4[7]

解 无向图欧拉回路计数是 #P 完全问题，但这题的图较为简单，确定了 $S - T$ 的边中从 S 指向 T 的有多少条，就可以确定其他三条边的定向方案，然后直接套用 BEST 定理就得到 $O(a + b + c + d)$ 的做法.

外部参考资料

[1] 「NOI2021」路径交点 

[2] 「HEOI2015」小 Z 的房间 

[3] 「FJOI2007」轮状病毒 

[4] 「BZOJ3659」WHICH DREAMED IT 

[5]「联合省选 2020 A」作业题 

[6] 「北京省选集训 2019」生成树计数 

[7] AGC051D C4 

11.27 LGV 引理

简介

Lindström–Gessel–Viennot lemma，即 LGV 引理，可以用来处理有向无环图上不相交路径计数等问题.

前置知识：图论相关概念→11.2@p2598 中的基础部分、矩阵→9.14.4@p1716、高斯消元求行列式→9.19.3@p1944.

LGV 引理仅适用于 **有向无环图**.

定义

$\omega(P)$ 表示 P 这条路径上所有边的边权之积.（路径计数时，可以将边权都设为 1）（事实上，边权可以为生成函数）

$e(u,v)$ 表示 u 到 v 的 **每一条** 路径 P 的 $\omega(P)$ 之和，即 $e(u,v)=\sum\limits_{P:u\to v}\omega(P)$.

起点集合 A，是有向无环图点集的一个子集，大小为 n.

终点集合 B，也是有向无环图点集的一个子集，大小也为 n.

一组 $A\to B$ 的不相交路径 S：S_i 是一条从 A_i 到 $B_{\sigma(S)_i}$ 的路径（$\sigma(S)$ 是一个排列），对于任何 $i\neq j$，S_i 和 S_j 没有公共顶点.

$t(\sigma)$ 表示排列 σ 的逆序对个数.

引理

$$M=\begin{bmatrix} e(A_1,B_1) & e(A_1,B_2) & \cdots & e(A_1,B_n)\\ e(A_2,B_1) & e(A_2,B_2) & \cdots & e(A_2,B_n)\\ \vdots & \vdots & \ddots & \vdots\\ e(A_n,B_1) & e(A_n,B_2) & \cdots & e(A_n,B_n)\end{bmatrix}$$

$$\det(M)=\sum_{S:A\to B}(-1)^{t(\sigma(S))}\prod_{i=1}^{n}\omega(S_i)$$

其中 $\sum\limits_{S:A\to B}$ 表示满足上文要求的 $A\to B$ 的每一组不相交路径 S.

证明

由行列式定义可得

$$\begin{aligned}\det(M)&=\sum_{\sigma}(-1)^{t(\sigma)}\prod_{i=1}^{n}e(a_i,b_{\sigma(i)})\\&=\sum_{\sigma}(-1)^{t(\sigma)}\prod_{i=1}^{n}\sum_{P:a_i\to b_{\sigma(i)}}\omega(P)\end{aligned}$$

观察到 $\prod_{i=1}^{n}\sum_{P:a_i\to b_{\sigma(i)}}\omega(P)$，实际上是所有从 A 到 B 排列为 σ 的路径组 P 的 $\omega(P)$ 之和.

$$\begin{aligned}&\sum_{\sigma}(-1)^{t(\sigma)}\prod_{i=1}^{n}\sum_{P:a_i\to b_{\sigma(i)}}\omega(P)\\=&\sum_{\sigma}(-1)^{t(\sigma)}\sum_{P=\sigma}\omega(P)\\=&\sum_{P:A\to B}(-1)^{t(\sigma)}\prod_{i=1}^{n}\omega(P_i)\end{aligned}$$

此处 P 为任意路径组.

设 U 为不相交路径组，V 为相交路径组，

$$\begin{aligned}&\sum_{P:A\to B}(-1)^{t(\sigma)}\prod_{i=1}^{n}\omega(P_i)\\=&\sum_{U:A\to B}(-1)^{t(U)}\prod_{i=1}^{n}\omega(U_i)+\sum_{V:A\to B}(-1)^{t(V)}\prod_{i=1}^{n}\omega(V_i)\end{aligned}$$

设 P 中存在一个相交路径组 $P_i:a_1\to u\to b_1, P_j:a_2\to u\to b_2$，则必然存在和它相对的一个相交路径组 $P_{(i)'}=a_1\to u\to b_2, P_{(j)'}=a_2\to u\to b_1$，$P'$ 的其他路径与 P 相同. 可得 $\omega(P)=\omega(P'), t(P)=t(P')\pm 1$.

因此我们有 $\sum_{V:A\to B}(-1)^{t(\sigma)}\prod_{i=1}^{n}\omega(V_i)=0$.

则 $\det(M)=\sum_{U:A\to B}(-1)^{t(U)}\prod_{i=1}^{n}\omega(U_i)$.

证毕[1].

例题

例 1 CF348D Turtles[2]

题意：有一个 $n\times m$ 的格点棋盘，其中某些格子可走，某些格子不可走. 有一只海龟从 (x,y) 只能走到 $(x+1,y)$ 和 $(x,y+1)$ 的位置，求海龟从 $(1,1)$ 到 (n,m) 的不相交路径数对 10^9+7 取模之后的结果. $2\le n,m\le 3000$.

比较直接的 LGV 引理的应用. 考虑所有合法路径，发现从 $(1,1)$ 出发一定要经过 $A=\{(1,2),(2,1)\}$，而到达终点一定要经过 $B=\{(n-1,m),(n,m-1)\}$，则 A,B 可立即选定. 应用 LGV 引理可得答案为：

$$\begin{vmatrix}f(a_1,b_1) & f(a_1,b_2)\\ f(a_2,b_1) & f(a_2,b_2)\end{vmatrix}=f(a_1,b_1)\times f(a_2,b_2)-f(a_1,b_2)\times f(a_2,b_1)$$

其中 $f(a,b)$ 为图上 $a\to b$ 的路径数，带有障碍格点的路径计数问题可以直接做一个 $O(nm)$ 的 dp，则 f 易求. 最终复杂度 $O(nm)$.

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <string>

using namespace std;

using ll = long long;
```

[1]证明来源于 知乎 - LGV 引理证明[1]

```cpp
constexpr int MOD = 1e9 + 7;
constexpr int SIZE = 3010;

string board[SIZE];
int dp[SIZE][SIZE];

int f(int x1, int y1, int x2, int y2) {
  memset(dp, 0, sizeof dp);

  dp[x1][y1] = board[x1][y1] == '.';
  for (int i = 1; i <= x2; i++) {
    for (int j = 1; j <= y2; j++) {
      if (board[i][j] == '#') {
        continue;
      }
      dp[i][j] = (dp[i][j] + dp[i - 1][j]) % MOD;
      dp[i][j] = (dp[i][j] + dp[i][j - 1]) % MOD;
    }
  }
  return dp[x2][y2] % MOD;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  int n, m;
  cin >> n >> m;

  for (int i = 1; i <= n; i++) {
    cin >> board[i];
    board[i] = " " + board[i];
  }

  ll f11 = f(1, 2, n - 1, m);
  ll f12 = f(1, 2, n, m - 1);
  ll f21 = f(2, 1, n - 1, m);
  ll f22 = f(2, 1, n, m - 1);

  ll ans = ((f11 * f22) % MOD - (f12 * f21) % MOD + MOD) % MOD;
  cout << ans << '\n';

  return 0;
}
```

例 2 HDU 5852 Intersection is not allowed![3]

题意：有一个 $n \times n$ 的棋盘，一个棋子从 (x, y) 只能走到 $(x, y+1)$ 或 $(x+1, y)$，有 k 个棋子，一开始第 i 个棋子放在 $(1, a_i)$，最终要到 (n, b_i)，路径要两两不相交，求方案数对 10^9+7 取模．$1 \le n \le 10^5$，$1 \le k \le 100$，保证 $1 \le a_1 < a_2 < ... < a_n \le n$，$1 \le b_1 < b_2 < ... < b_n \le n$.

观察到如果路径不相交就一定是 a_i 到 b_i，因此 LGV 引理中一定有 $\sigma(S)_i = i$，不需要考虑符号问题. 边权设为 1，直接套用引理即可.

从 $(1, a_i)$ 到 (n, b_j) 的路径条数相当于从 $n-1+b_j-a_i$ 步中选 $n-1$ 步向下走，所以 $e(A_i, B_j) = \binom{n-1+b_j-a_i}{n-1}$.

行列式可以使用高斯消元求.

复杂度为 $O(n + k(k^2 + \log p))$，其中 $\log p$ 是求逆元复杂度.

参考代码

```cpp
#include <algorithm>
#include <iostream>

using ll = long long;

constexpr int K = 105;
constexpr int N = 100005;
constexpr int mod = 1e9 + 7;

int T, n, k, a[K], b[K], fact[N << 1], m[K][K];

int qpow(int x, int y) {
  int out = 1;
  while (y) {
    if (y & 1) out = (ll)out * x % mod;
    x = (ll)x * x % mod;
    y >>= 1;
  }
  return out;
}

int c(int x, int y) {
  return (ll)fact[x] * qpow(fact[y], mod - 2) % mod *
         qpow(fact[x - y], mod - 2) % mod;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  fact[0] = 1;
  for (int i = 1; i < N * 2; ++i) fact[i] = (ll)fact[i - 1] * i % mod;

  cin >> T;

  while (T--) {
    cin >> n >> k;

    for (int i = 1; i <= k; ++i) cin >> a[i];
    for (int i = 1; i <= k; ++i) cin >> b[i];

    for (int i = 1; i <= k; ++i) {
```

```
      for (int j = 1; j <= k; ++j) {
        if (a[i] <= b[j])
          m[i][j] = c(b[j] - a[i] + n - 1, n - 1);
        else
          m[i][j] = 0;
      }
    }

    for (int i = 1; i < k; ++i) {
      if (!m[i][i]) {
        for (int j = i + 1; j <= k; ++j) {
          if (m[j][i]) {
            std::swap(m[i], m[j]);
            break;
          }
        }
      }
      if (!m[i][i]) continue;
      int inv = qpow(m[i][i], mod - 2);
      for (int j = i + 1; j <= k; ++j) {
        if (!m[j][i]) continue;
        int mul = (ll)m[j][i] * inv % mod;
        for (int p = i; p <= k; ++p) {
          m[j][p] = (m[j][p] - (ll)m[i][p] * mul % mod + mod) % mod;
        }
      }
    }

    int ans = 1;

    for (int i = 1; i <= k; ++i) ans = (ll)ans * m[i][i] % mod;

    cout << ans << '\n';
  }

  return 0;
}
```

参考资料

[1] 知乎 - LGV 引理证明 

[2] CF348D Turtles 

[3] HDU 5852 Intersection is not allowed! 

11.28 最大团搜索算法

Authors: Persdre

前置知识：团→11.2@p2598

引入

在计算机科学中，团问题指的是在给定的图中找到团（顶点的子集，都彼此相邻，也称为完全子图）的计算问题.

团的问题在现实生活中也有体现．例如我们考虑一个社交网络，其中图的点代表用户，图的边代表其所连接的两个用户互相认识．那么我们找到了一个团，也就找到了一群互相认识的人．

我们如果想要找到这个社交网络中最大的一群互相认识的人，那么就需要用到最大团搜索算法．

我们已经介绍了 极大团→11.2@p2598 的概念，最大团指的是点数量最多的极大团．

解释

想法是利用递归和回溯，用一个列表存储点，每次加入点进来都检查这些点是否仍在一个团中．如果加入进来这个点后就无法还是一个团了，就回溯到满足条件的位置，重新加入别的点．

采用回溯策略的原因是，我们并不知道某个顶点 v **最终** 是否是最大团中的成员．如果递归算法选择 v 作为最大团的成员时，并没有找到最大团，那么应该回溯，并查找最大团中没有 v 的解．

过程

Bron–Kerbosch 算法对于这种想法进行了优化实现．它的基础形式是通过给定三个集合：R、P、X 来递归地进行搜索．步骤如下：

1. 初始化集合 R, X 分别为空，集合 P 是图中所有点的集合．
2. 每次从集合 P 中取顶点 v，当集合中没有顶点时，有两种情况：
    1. 集合 R 是最大团，此时集合 X 为空
    2. 无最大团，此时回溯
3. 对于每一个从集合 P 中取得的顶点 v，有如下处理：
    1. 将顶点 v 加到集合 R 中，之后递归集合 R, P, X
    2. 从集合 P 中删除顶点 v，并将顶点 v 添加到集合 X 中
    3. 若集合 P, X 都为空，则集合 R 即为最大团

此方法也可继续优化．为了节省时间让算法更快的回溯，可以通过设定关键点（pivot vertex）来进行搜索．另一种优化思路是在开始时把所有点排序，枚举时按照下标顺序，防止重复．

实现

伪代码

```
R := {}
P := node set of G
X := {}

BronKerbosch1(R, P, X):
    if P and X are both empty:
        report R as a maximal clique
    for each vertex v in P:
        BronKerbosch1(R ⋃ {v}, P ⋂ N(v), X ⋂ N(v))
        P := P \ {v}
        X := X ⋃ {v}
```

C++ 实现

实现代码

```cpp
#include <cstring>
#include <iostream>
using namespace std;
```

```cpp
constexpr int MAXN = 105;

struct MaxClique {
  bool g[MAXN][MAXN];
  int n, dp[MAXN], st[MAXN][MAXN], ans;

  // dp[i]表示第i个点之后能组成的最大团的大小，
  // st[i][j]表示算法中第i层dfs所需要的点的集合，保存有可能是最大团其中之一的点

  void init(int n) {
    this->n = n;
    memset(g, false, sizeof(g));
  }

  void addedge(int u, int v, int w) { g[u][v] = w; }

  bool dfs(int sz, int num) {
    if (sz == 0) {
      if (num > ans) {
        ans = num;
        return true;
      }
      return false;
    }
    for (int i = 0; i < sz; i++) {  // 在第num层的集合中枚举一个点i
      if (sz - i + num <= ans) return false;  // 剪枝1
      int u = st[num][i];
      if (dp[u] + num <= ans) return false;  // 剪枝2
      int cnt = 0;
      for (
          int j = i + 1; j < sz;
          j++) {  // 在第num层遍历在i之后的且与i所相连的点，并且加入第num+1层集合
        if (g[u][st[num][j]]) st[num + 1][cnt++] = st[num][j];
      }
      if (dfs(cnt, num + 1)) return true;
    }
    return false;
  }

  int solver() {
    ans = 0;
    memset(dp, 0, sizeof(dp));
    for (int i = n; i >= 1; i--) {
      int cnt = 0;
      for (int j = i + 1; j <= n; j++) {  // 初始化第1层集合
        if (g[i][j]) st[1][cnt++] = j;
      }
      dfs(cnt, 1);
      dp[i] = ans;
    }
    return ans;
  }
```

```
56
57 } maxclique;
58
59 int main() {
60   cin.tie(nullptr)->sync_with_stdio(false);
61   int n;
62   while (cin >> n, n) {
63     maxclique.init(n);
64     for (int i = 1; i <= n; i++) {
65       for (int j = 1; j <= n; j++) {
66         int x;
67         cin >> x;
68         maxclique.addedge(i, j, x);
69       }
70     }
71     cout << maxclique.solver() << '\n';
72   }
73   return 0;
74 }
75
```

例题

POJ 2989: All Friends[1]

题目大意：给出 n 个人，其中有 m 对朋友，求最大团数量.

思路：模版题，要用 Bron–Kerbosch 算法

伪代码：

```
BronKerbosch(All, Some, None):
    if Some and None are both empty:
        report All as a maximal clique // 所有点已选完，且没有不能选的点，累加答案
    for each vertex v in Some: // 枚举 Some 中的每一个元素
        BronKerbosch1(All ⋃ {v}, Some ⋂ N(v), None ⋂ N(v))
        // 将 v 加入 All，显然只有与 v 为朋友的人才能作为备选，None 中也只有与 v 为朋友的才会对接下来造成影响
        Some := Some - {v} // 已经搜过，从 Some 中删除，加入 None
        None := None ⋃ {v}
```

为了节省时间和让算法更快的回溯，我们可以通过设定关键点（pivot vertex）v 进行优化.

我们知道在上述的算法中必然有许多重复计算之前计算过的极大团，然后回溯的过程.

以前文提到的 R、P、X 三个集合为例：

我们考虑如下问题，取集合 $P \cup X$ 中的一个点 u，要与 R 集合构成极大团，那么取的点必然是 $P \cap N(u)$ 中一个点（$N(u)$ 代表与 u 相邻的点）.

如果取完 u 之后我们再取与 u 相邻的点 v 也能加入到极大团，那么我们只取 u 就好了. 这样做可以减少之后对 v 的重复计算. 我们之后只需要取与 u 不相邻的点.

加入优化后的 C++ 代码实现：

实现代码

```cpp
#include <cstring>
#include <iostream>
using namespace std;
constexpr int MAXN = 130;
bool mp[MAXN][MAXN];
int some[MAXN][MAXN], none[MAXN][MAXN], all[MAXN][MAXN];
int n, m, ans;

void dfs(int d, int an, int sn, int nn) {
  if (!sn && !nn) ++ans;
  int u = some[d][0];
  for (int i = 0; i < sn; ++i) {
    int v = some[d][i];
    if (mp[u][v]) continue;
    for (int j = 0; j < an; ++j) all[d + 1][j] = all[d][j];
    all[d + 1][an] = v;
    int tsn = 0, tnn = 0;
    for (int j = 0; j < sn; ++j)
      if (mp[v][some[d][j]]) some[d + 1][tsn++] = some[d][j];
    for (int j = 0; j < nn; ++j)
      if (mp[v][none[d][j]]) none[d + 1][tnn++] = none[d][j];
    dfs(d + 1, an + 1, tsn, tnn);
    some[d][i] = 0, none[d][nn++] = v;
    if (ans > 1000) return;
  }
}

int work() {
  ans = 0;
  for (int i = 0; i < n; ++i) some[1][i] = i + 1;
  dfs(1, 0, n, 0);
  return ans;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> n >> m) {
    memset(mp, 0, sizeof mp);
    for (int i = 1; i <= m; ++i) {
      int u, v;
      cin >> u >> v;
      mp[u][v] = mp[v][u] = true;
    }
    int tmp = work();
    if (tmp > 1000)
      cout << "Too many maximal sets of friends.\n";
    else
      cout << tmp << '\n';
  }
  return 0;
}
```

```
52
```

习题

- ZOJ 1492 Maximum Clique[2]
- POJ 1419 无向图最大团[3]
- POJ 1129 广播电台[4]

参考资料

- 团问题 - 维基百科[5]
- 无向图的极大团、最大团（Bron–Kerbosch 算法）[6]
- 最大团问题——Bron–Kerbosch 算法[7]
- 最大团问题[8]

外部参考资料

[1] POJ 2989: All Friends

[2] ZOJ 1492 Maximum Clique

[3] POJ 1419 无向图最大团

[4] POJ 1129 广播电台

[5] 团问题 - 维基百科

[6] 无向图的极大团、最大团（Bron–Kerbosch 算法）

[7] 最大团问题——Bron–Kerbosch 算法

[8] 最大团问题

11.29 支配树

前言

1959 年，「支配」这一概念由 Reese T. Prosser 在 一篇关于网络流的论文[1] 中提出，但并未提出具体的求解算法；直到 1969 年，Edward S. Lowry 和 C. W. Medlock 才首次提出了 有效的求解算法[2]．而目前使用最为广泛的 Lengauer–Tarjan 算法则由 Lengauer 和 Tarjan 于 1979 年在 一篇论文[3] 中提出．

在 OI 界中，支配树的概念最早在 ZJOI2012 灾难[4] 中被引入，当时也被称为「灭绝树」；陈孙立也在 2020 年的国家集训队论文中介绍了这一算法．

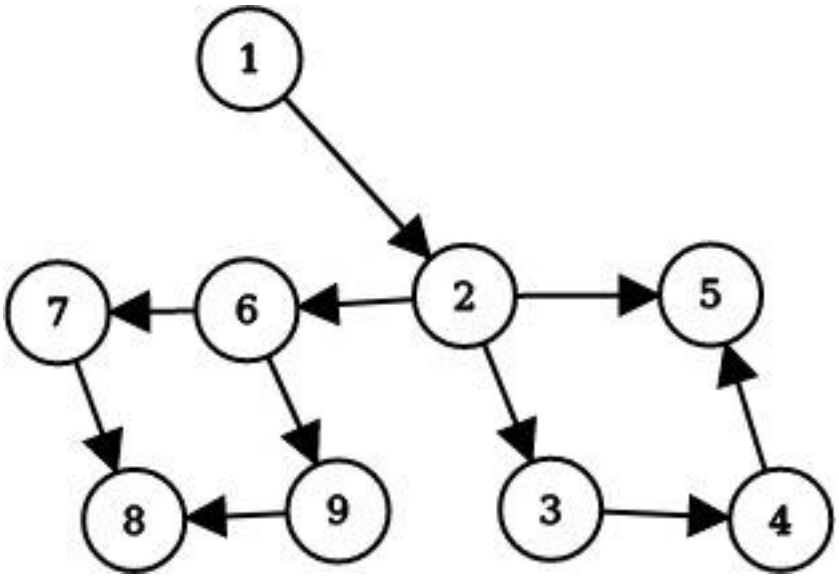
目前支配树在竞赛界并不流行，其相关习题并不多见；但支配树在工业上，尤其是编译器相关领域，已有广泛运用．

本文将介绍支配树的概念及几种求解方法．

支配关系

我们在任意的一个有向图上钦定一个入口结点 s，对于一个结点 u，若从 s 到 u 的每一条路径都经过某一个结点 v，那么我们称 v **支配** u，也称 v 是 u 的一个 **支配点**，记作 $v\ dom\ u$．

对于从 s 出发无法到达的结点，讨论其支配关系是没有意义的，因此在没有特殊说明的情况下，本文默认 s 能到达图上任何一个结点．



例如这张有向图中，2 被 1 支配，3 被 1, 2 支配，4 被 1, 2, 3 支配，5 被 1, 2 支配，etc.

引理

在下文的引理中，默认 $u, v, w \neq s$

引理 1： s 是其所有结点的支配点；任意一个结点都是其自身的支配点.

证明： 显然任何一条从 s 到 u 的路径都必须经过 s 和 u 这两个结点.

引理 2： 仅考虑简单路径得出的支配关系与考虑所有路径得出的关系相同.

证明： 对于非简单路径，我们设两次经过某个结点之间经过的所有结点的点集为 S，若将 S 中的结点删去，便能将每个非简单路径与一个简单路径对应.

在 S 中，在非简单路径而不在简单路径上的点一定不可能成为支配点，因为至少有一条 s 到 u 的简单路径不包括这个点；同时在简单路径和非简单路径上的点只需在简单路径上讨论即可.

综上，删去非简单路径对支配关系没有影响.

引理 3： 如果 $u\ dom\ v$，$v\ dom\ w$，则 $u\ dom\ w$.

证明： 经过 w 的路径必定经过 v，经过 v 的路径必定经过 u，因此经过 w 的路径必定经过 u，即 $u\ dom\ w$.

引理 4： 如果 $u\ dom\ v$，$v\ dom\ u$，则 $u = v$.

证明： 假设 $u \neq v$，则任意一个到达 v 的路径都已经到达过 u，同时任意一个到达 u 的路径都已经到达过 v，矛盾.

引理 5： 若 $u \neq v \neq w$,$u\ dom\ w$ 且 $v\ dom\ w$，则有 $u\ dom\ v$ 或 $v\ dom\ u$.

证明： 考虑一条 $s \to ... \to u \to ... \to v \to ... \to w$ 的路径，若 u,v 不存在支配关系，则一定存在一条不经过 u 的从 s 到 v 的路径，即存在一条 $s \to ... \to v \to ... \to w$ 的路径，与 $u\ dom\ w$ 矛盾.

求解支配关系

结点删除法

一个和定义等价的结论：如果我们删去图中的某一个结点后，有一些结点变得不可到达，那么这个被删去的结点支配这些变得不可到达的结点.

因此我们只要尝试将每一个结点删去后 dfs 即可，代码复杂度为 $O(n^3)$. 下面给出核心代码.

```cpp
// 假设图中有 n 个结点，起始点 s = 1
std::bitset<N> vis;
std::vector<int> edge[N];
std::vector<int> dom[N];

void dfs(int u, int del) {
  vis[u] = true;
  for (int v : edge[u]) {
    if (v == del or vis[v]) {
      continue;
    }
    dfs(v, del);
  }
}

void getdom() {
  for (int i = 2; i <= n; ++i) {
    vis.reset();
    dfs(1, i);
    for (int j = 1; j <= n; ++j) {
      if (!vis[j]) {
        dom[j].push_back(i);
      }
    }
  }
}
```

数据流迭代法

数据流迭代法也是 OI 中不常见的一个知识点，这里先做简要介绍.

数据流分析是编译原理中的概念，用于分析数据如何在程序执行路径上的流动；而数据流迭代法是在程序的流程图的结点上列出方程并不断迭代求解，从而求得程序的某些点的数据流值的一种方法. 这里我们就是把有向图看成了一个程序流程图.

这个问题中，方程为：

$$dom(u) = \{u\} \cup \left(\bigcap_{v \in pre(u)} dom(v) \right)$$

其中 $pre(u)$ 定义为 u 的前驱结点组成的点集. 这个方程可以通过引理 3 得到.

翻译成人话就是，一个点的支配点的点集为它所有前驱结点的支配点集的交集，再并上它本身. 根据这个方程将每个结点上的支配点集不断迭代直至答案不变即可.

为了提高效率，我们希望每轮迭代时，当前迭代的结点的所有前驱结点尽可能都已经执行完了这次迭代，因此我们要利用深度优先排序得出这个图的逆后序，根据这个顺序进行迭代.

下面给出核心代码的参考实现. 这里需要预先处理每个点的前驱结点集和图的逆后序，但这不是本文讨论的主要内容，故这里不提供参考实现.

```cpp
std::vector<int> pre[N];  // 每个结点的前驱结点
std::vector<int> ord;     // 图的逆后序
std::bitset<N> dom[N];
std::vector<int> Dom[N];

void getdom() {
```

```
  dom[1][1] = true;
  flag = true;
  while (flag) {
    flag = false;
    for (int u : ord) {
      std::bitset<N> tmp;
      tmp[u] = true;
      for (int v : pre[u]) {
        tmp &= dom[v];
      }
      if (tmp != dom[u]) {
        dom[u] = tmp;
        flag = true;
      }
    }
  }
  for (int i = 2; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
      if (dom[i][j]) {
        Dom[i].push_back(j);
      }
    }
  }
}
```

不难看出上述算法的复杂度为 $O(n^2)$.

支配树

上一节我们发现，除 s 外，一个点的支配点至少有两个，s 和其自身.

我们将任意一个结点 u 的支配点中，除自身外与自己距离最近的结点 v 称作 u 的直接支配点，记作 $idom(u)=v$. 显然除了 s 没有直接支配点外，每个结点都有唯一一个直接支配点.

我们考虑对于除 s 外每一个结点 u 从 $idom(u)$ 向 u 连边，便构成了一个有 n 个结点，$n-1$ 条边的有向图. 根据引理 3 和引理 4，我们知道支配关系一定不会构成循环，也就是这些边一定不构成环，因此我们得到的图事实上是一棵树. 我们称这颗树为原图的 **支配树**.

求解支配树

根据 dom 求解

不妨考虑某个结点的支配点集 $\{s_1,s_2,...,s_k\}$，则一定存在一条路径 $s\to ...\to s_1\to ...\to s_2\to ...\to ...\to s_k\to ...\to u$. 显然 u 的直接支配点为 s_k. 因此直接支配点的定义等价于：

对于一个结点 u 的支配点集 S，若 $v\in S$ 满足 $\forall w\in S\setminus\{u,v\}, w\ dom\ v$，则 $idom(u)=v$.

因此，利用前文所述的算法得到每个结点的支配点集之后，我们根据上述定义便能很轻松地得到每个点的直接支配点，从而构造出支配树. 下面给出参考代码.

```
std::bitset<N> dom[N];
std::vector<int> Dom[N];
int idom[N];

void getidom() {
  for (int u = 2; u <= n; ++u) {
    for (int v : Dom[u]) {
```

```
      std::bitset<N> tmp = (dom[v] & dom[u]) ^ dom[u];
      if (tmp.count() == 1 and tmp[u]) {
        idom[u] = v;
        break;
      }
    }
  }
  for (int u = 2; u <= n; ++u) {
    e[idom[u]].push_back(u);
  }
}
```

树上的特例

显然树型图的支配树就是它本身.

DAG 上的特例

我们发现 DAG 有一个很好的性质：根据拓扑序求解，先求得的解不会对后续的解产生影响. 我们可以利用这个特点快速求得 DAG 的支配树.

> ⚠ **提醒**
>
> 值得注意的是此处的 DAG 只能有一个起点，如果有多个起点，受起点支配的点在支配树上出现有多个父亲的情况，从而使支配关系不能简单的用支配树来表达.

引理 6： 在有向图上，$v\ dom\ u$ 当且仅当 $\forall w \in pre(u), v\ dom\ w$.

证明： 首先来证明充分性. 考虑任意一条从 s 到 u 的路径都一定经过一个结点 $w \in pre(u)$，而 v 支配这个结点，因此任意一条从 s 到 u 的路径都一定经过 v，因此我们得到 $v\ dom\ u$.

然后是必要性. 如果 $\exists w \in pre(u)$，v 不支配 w，则一定有一条不经过 v 的路径 $s \to \cdots \to w \to \cdots \to u$，因此 v 不支配 u.

我们发现，u 的支配点一定是其所有前驱结点在支配树上的公共祖先，那么显然 u 的直接支配点是所有前驱结点在支配树上的 LCA. 考虑倍增求解 LCA 可以支持每次添加一个结点，上述算法显然是可行的.

下面给出参考实现：

```
std::stack<int> sta;
std::vector<int> e[N], g[N], tree[N];  // g 是原图的反图，tree 是支配树
int n, s, in[N], tpn[N], dep[N], idom[N];  // n 为总点数，s 为起始点，in 为入度
int fth[N][17];

void topo(int s) {
  sta.push(s);
  while (!sta.empty()) {
    int u = sta.top();
    sta.pop();
    tpn[++tot] = u;
    for (int v : e[u]) {
      --in[v];
      if (!in[v]) {
        sta.push(v);
      }
    }
  }
```

```
}

int lca(int u, int v) {
  if (dep[u] < dep[v]) {
    std::swap(u, v);
  }
  for (int i = 15; i >= 0; --i) {
    if (dep[fth[u][i]] >= dep[v]) {
      u = fth[u][i];
    }
  }
  if (u == v) {
    return u;
  }
  for (int i = 15; i >= 0; --i) {
    if (fth[u][i] != fth[v][i]) {
      u = fth[u][i];
      v = fth[v][i];
    }
  }
  return fth[u][0];
}

void build() {
  topo(s);
  for (int i = 1; i <= n; ++i)
    for (int j = 0; j <= 15; ++j) fth[i][j] = s;
  for (int i = 1; i <= n; ++i) {
    int u = tpn[i];
    if (g[u].size()) {
      int v = g[u][0];
      for (int j = 1, q = g[u].size(); j < q; ++j) {
        v = lca(v, g[u][j]);
      }
      tree[v].push_back(u);
      fth[u][0] = v;
      dep[u] = dep[v] + 1;
      for (int i = 1; i <= 15; ++i) {
        fth[u][i] = fth[fth[u][i - 1]][i - 1];
      }
    }
  }
}

```

Lengauer–Tarjan 算法

Lengauer–Tarjan 算法是求解支配树最有名的算法之一，可以在 $O(n\alpha(n,m))$ 的时间复杂度内求出一个有向图的支配树．这一算法引入了 **半支配点** 的概念，并通过半支配点辅助求得直接支配点．

约定

首先，我们从 s 出发对这个有向图进行 dfs，所经过的点和边形成了一颗树 T．我们称走过的边为树边，其余的为非树边；令 $dfn(u)$ 表示结点 u 被第几个遍历到；定义 $u < v$ 当且仅当 $dfn(u) < dfn(v)$.

半支配点

一个结点 u 的半支配点，是满足从这个结点 v 出发有一条路径，路径上除了 u, v 之外每个结点都大于 u 的结点中最小的那一个．形式化的说，u 的半支配点 $sdom(u)$ 定义为：

$$sdom(u) = \min(v \mid \exists v = v_0 \to v_1 \to ... \to v_k = u, \forall 1 \le i \le k-1, v_i > u)$$

我们发现半支配点有一些有用的性质：

引理 7： 对于任意结点 u，$sdom(u) < u$.

证明： 根据定义不难发现，u 在 T 上的父亲 $fa(u)$ 也满足成为半支配点的条件，且 $fa(u) < u$，因此任何大于 u 的结点都不可能成为其半支配点.

引理 8： 对于任意结点 u，$idom(u)$ 是其在 T 上的祖先.

证明： T 上从 s 到 u 的路径对应了原图上的一条路径，则 $idom(u)$ 必定在这个路径上.

引理 9： 对于任意结点 u，$sdom(u)$ 是其在 T 上的祖先.

证明： 假设 $sdom(u)$ 不是 u 的祖先，那么 $sdom(u)$ 不可能连向任何 dfs 序大于等于 u 的结点（否则这个点应在 $sdom(u)$ 的子树内而非其他子树内），矛盾.

引理 10： 对于任意结点 u，$idom(u)$ 是 $sdom(u)$ 的祖先.

证明： 考虑可以从 s 到 $sdom(u)$ 再从定义中的路径走到 u．根据定义，$sdom(u)$ 到 u 的路径上的点均不支配 u，故 $idom(u)$ 一定是 $sdom(u)$ 的祖先.

引理 11： 对于任意结点 $u \neq v$ 满足 v 是 u 的祖先，则要么有 v 是 $idom(u)$ 的祖先，要么 $idom(u)$ 是 $idom(v)$ 的祖先.

证明： 对于任意在 v 和 $idom(v)$ 之间的结点 w，根据直接支配点的定义，一定存在一条不经过 w 的，从 s 到 $idom(v)$ 再到 v 的路径．因此这些结点 w 一定不是 $idom(u)$，因此 $idom(u)$ 要么是 v 的后代，要么是 $idom(v)$ 的祖先.

根据以上引理，我们可以得到以下定理：

定理 1： 一个点 u 的半支配点是其前驱与其支配点在 T 上的，大于 u 的所有祖先的半支配点中最小的节点．形式化地说，$sdom(u) = \min(\{v \mid \exists v \to u, v < u\} \cup \{sdom(w) \mid w > u \ and \ \exists w \to ... \to v \to u\})$.

证明： 令 x 等于上式右侧.

我们首先证明 $sdom(u) \le x$．根据引理 7 我们知道这个命题等价于证明上述的两种都满足成为半支配点的条件．x 是 u 的前驱时的情况是显然的，对于后半部分，我们考虑将半支配点定义中所述路径 $x = v_0 \to ... \to v_j = w$ 和 T 上的一条满足 $\forall i \in [j, k-1], v_i \ge w > u$ 的路径 $w = v_j \to ... \to v_k = v$ 以及路径 $v \to u$ 拼接，从而我们构造出一条满足半支配点定义的路径.

然后我们证明 $sdom(u) \ge x$．考虑 u 到其半支配点的定义中所述路径 $sdom(u) = v_0 \to v_1 \to ... \to v_k = u$．不难看出 $k = 1$ 和 $k > 1$ 分别对应了定义中的两个选取方法．若 $k = 1$，则存在有向边 $sdom(u) \to u$，根据引理 7 即可得证；若 $k > 1$，令 j 是满足 $j \ge 1$ 且 v_j 是 v_{k-1} 在 T 上祖先的最小数．考虑到 k 满足上述条件，这样的 j 一定存在.

考虑证明 $v_0 \to ... \to v_j$ 是满足成为 v_j 半支配点条件的一条路径，即证明 $\forall i \in [1, j), v_i > v_j$．若不是，则令 i 为满足 $v_i < v_j$ 中使 v_i 最小的数．于是路径 $v_i \to ... \to v_j$ 上所有结点都不小于 v_i，根据 dfs 树的性质，v_i 一定是 v_j 的祖先，进而也是 v_{k-1} 的祖先，这和 j 的最小性矛盾．于是 $sdom(v_j) \le sdom(u)$．又因为 $x \le sdom(v_j)$，所以 $sdom(u) \ge x$. 结合此前证明的 $sdom(u) \le x$，故 $x = sdom(u)$.

根据定理 1 我们便可以求出每个点的半支配点了. 不难发现计算半支配点的复杂度瓶颈在第二种情况上，我们考虑利用带权并查集优化，每次路径压缩时更新最小值即可.

```
1 void dfs(int u) {
```

```
  dfn[u] = ++dfc;
  pos[dfc] = u;
  for (int i = h[0][u]; i; i = e[i].x) {
    int v = e[i].v;
    if (!dfn[v]) {
      dfs(v);
      fth[v] = u;
    }
  }
}

int find(int x) {
  if (fa[x] == x) {
    return x;
  }
  int tmp = fa[x];
  fa[x] = find(fa[x]);
  if (dfn[sdm[mn[tmp]]] < dfn[sdm[mn[x]]]) {
    mn[x] = mn[tmp];
  }
  return fa[x];
}

void getsdom() {
  dfs(1);
  for (int i = 1; i <= n; ++i) {
    mn[i] = fa[i] = sdm[i] = i;
  }
  for (int i = dfc; i >= 2; --i) {
    int u = pos[i], res = INF;
    for (int j = h[1][u]; j; j = e[j].x) {
      int v = e[j].v;
      if (!dfn[v]) {
        continue;
      }
      find(v);
      if (dfn[v] < dfn[u]) {
        res = std::min(res, dfn[v]);
      } else {
        res = std::min(res, dfn[sdm[mn[v]]]);
      }
    }
    sdm[u] = pos[res];
    fa[u] = fth[u];
  }
}

```

求解直接支配点

转化为 DAG

可是我还是不知道半支配点有什么用！

我们考虑在 T 上对每一个 u 加入 $sdom(u) \to u$ 的有向边．根据引理 9，新得到的这张图 G 一定是有向无环图；又根据引理 10，我们还发现这样加边不会改变支配关系，因此我们把原图转化为了一张 DAG，利用上文的算法求解即可．

通过半支配点求解

建一堆图也太不优雅了！

定理 2： 对于任意节点 u，若 T 上从 $sdom(u)$ 到 w 的路径上的任意节点 v 都满足 $sdom(v) \geq sdom(w)$，则 $idom(u) = sdom(u)$.

证明： 根据引理 10 我们知道 $idom(u)$ 是 $sdom(u)$ 或其祖先，因此只需证明 $sdom(u)$ dom u.

考虑任意一条 s 到 u 的路径 P，我们需要证明 $sdom(u)$ 一定在 P 中．令 v 为 P 中最后一个满足 $v < sdom(u)$ 的节点．如果 v 不存在则必有 $sdom(u) = idom(u) = s$，否则令 w 是 P 中 v 之后在 DFS 树中从 $sdom(u)$ 到 u 的路径上的第一个点.

我们接下来证明 $sdom(w) \leq v < sdom(v)$. 考虑 T 上 v 到 w 的路径 $v = v_0 \to ...v_k = w$，若不成立，则存在 $i \in [1, k-1], v_i < w$. 此时一定存在某个 $j \in [i, k-1]$ 满足 v_j 是 w 的祖先. 由 v 的取值可知 $sdom(u) \leq v_j$，于是 v_j 也在 DFS 树中从 $sdom(u)$ 到 u 的路径上，与 w 的定义矛盾，因此 $sdom(w) \leq v < sdom(v)$，结合定理的条件有 $y = sdom(u)$，即路径 P 包含 $sdom(u)$.

定理 3： 对于任意节点 u，T 上从 $sdom(u)$ 到 u 的路径上的所有节点中半支配点最小的节点 v 一定满足 $sdom(v) \leq sdom(u)$ 和 $idom(v) = idom(u)$.

证明： 考虑到 u 本身也满足 v 的条件，因此 $sdom(v) \leq sdom(u)$.

由于 $idom(u)$ 是 v 在 T 上的祖先，由引理 11 可知 $idom(u)$ 也是 $idom(v)$ 的祖先，因此只需证明 $idom(v)$ 支配 u.

考虑任意一条 s 到 u 的路径 P，我们需要证明 $sdom(u)$ 一定在 P 中．令 x 为 P 中最后一个满足 $x < sdom(u)$ 的节点．如果 x 不存在则必有 $sdom(u) = idom(u) = s$，否则令 y 是 P 中 x 之后在 DFS 树中从 $sdom(u)$ 到 u 的路径上的第一个点.

与定理 2 的证明过程同理，我们可以得到 $sdom(y) \leq x$．根据引理 10 有 $sdom(y) \leq x < idom(v) \leq sdom(v)$．至此，由 v 的定义可知 y 不能是 $sdom(u)$ 的后代；另一方面，y 不能既是 $idom(v)$ 的后代也是 v 的祖先，否则沿 DFS 树从 s 到 $sdom(y)$ 再沿 P 走到 y，最后沿 DFS 树走到 v 的这条路径不经过 $idom(v)$，与支配点的定义矛盾．因此 $y = idom(v)$，即 P 包含 $idom(v)$.

根据以上两个定理我们能够得到 $sdom(u)$ 与 $idom(u)$ 之间的关系.

令 v 是满足 v 在 $sdom(u)$ 与 u 之间的结点的所有节点中，$sdom(v)$ 最小的一个节点，那么：

$$idom(u) = \begin{cases} sdom(u), \text{if } sdom(u) = sdom(v) \\ idom(v), \qquad\qquad \text{otherwise} \end{cases}$$

只要对上面求解半支配点的代码稍作修改即可.

```
struct E {
  int v, x;
} e[MAX * 4];

int h[3][MAX * 2];

int dfc, tot, n, m, u, v;
int fa[MAX], fth[MAX], pos[MAX], mn[MAX], idm[MAX], sdm[MAX], dfn[MAX],
    ans[MAX];

void add(int x, int u, int v) {
```

```
  e[++tot] = {v, h[x][u]};
  h[x][u] = tot;
}

void dfs(int u) {
  dfn[u] = ++dfc;
  pos[dfc] = u;
  for (int i = h[0][u]; i; i = e[i].x) {
    int v = e[i].v;
    if (!dfn[v]) {
      dfs(v);
      fth[v] = u;
    }
  }
}

int find(int x) {
  if (fa[x] == x) {
    return x;
  }
  int tmp = fa[x];
  fa[x] = find(fa[x]);
  if (dfn[sdm[mn[tmp]]] < dfn[sdm[mn[x]]]) {
    mn[x] = mn[tmp];
  }
  return fa[x];
}

void tar(int st) {
  dfs(st);
  for (int i = 1; i <= n; ++i) {
    fa[i] = sdm[i] = mn[i] = i;
  }
  for (int i = dfc; i >= 2; --i) {
    int u = pos[i], res = INF;
    for (int j = h[1][u]; j; j = e[j].x) {
      int v = e[j].v;
      if (!dfn[v]) {
        continue;
      }
      find(v);
      if (dfn[v] < dfn[u]) {
        res = std::min(res, dfn[v]);
      } else {
        res = std::min(res, dfn[sdm[mn[v]]]);
      }
    }
    sdm[u] = pos[res];
    fa[u] = fth[u];
    add(2, sdm[u], u);
    u = fth[u];
    for (int j = h[2][u]; j; j = e[j].x) {
      int v = e[j].v;
      find(v);
```

```
66        if (sdm[mn[v]] == u) {
67          idm[v] = u;
68        } else {
69          idm[v] = mn[v];
70        }
71      }
72      h[2][u] = 0;
73    }
74    for (int i = 2; i <= dfc; ++i) {
75      int u = pos[i];
76      if (idm[u] != sdm[u]) {
77        idm[u] = idm[idm[u]];
78      }
79    }
80  }
81
```

例题

洛谷 P5180【模板】支配树[5]

可以仅求解支配关系，求解过程中记录各个点支配了多少节点，也可以建出支配树求解每个节点的 size.

这里给出后一种解法的代码.

参考代码

```
1  #include <iostream>
2  using std::cin;
3  using std::cout;
4  constexpr int MAX = 3e5 + 5;
5  constexpr int INF = 0x5fffffff;
6
7  struct E {
8    int v, x;
9  } e[MAX * 4];
10
11 int n, m, u, v, tot, dfc;
12 int ans[MAX], dfn[MAX], pos[MAX], sdm[MAX], idm[MAX], fa[MAX], mn[MAX],
13     fth[MAX];
14 int h[3][MAX * 2];
15
16 void add(int x, int u, int v) {
17   e[++tot] = {v, h[x][u]};
18   h[x][u] = tot;
19 }
20
21 void dfs(int u) {
22   dfn[u] = ++dfc;
23   pos[dfc] = u;
24   for (int i = h[0][u]; i; i = e[i].x) {
25     int v = e[i].v;
26     if (!dfn[v]) {
```

```
      dfs(v);
      fth[v] = u;
    }
  }
}

int find(int x) {
  if (fa[x] == x) {
    return x;
  }
  int tmp = fa[x];
  fa[x] = find(fa[x]);
  if (dfn[sdm[mn[tmp]]] < dfn[sdm[mn[x]]]) {
    mn[x] = mn[tmp];
  }
  return fa[x];
}

void tar(int st) {
  dfs(st);
  for (int i = 1; i <= n; ++i) {
    mn[i] = fa[i] = sdm[i] = i;
  }
  for (int i = dfc; i >= 2; --i) {
    int u = pos[i], res = INF;
    for (int j = h[1][u]; j; j = e[j].x) {
      int v = e[j].v;
      if (!dfn[v]) {
        continue;
      }
      find(v);
      if (dfn[v] < dfn[u]) {
        res = std::min(res, dfn[v]);
      } else {
        res = std::min(res, dfn[sdm[mn[v]]]);
      }
    }
    sdm[u] = pos[res];
    fa[u] = fth[u];
    add(2, sdm[u], u);
    u = fth[u];
    for (int j = h[2][u]; j; j = e[j].x) {
      int v = e[j].v;
      find(v);
      if (u == sdm[mn[v]]) {
        idm[v] = u;
      } else {
        idm[v] = mn[v];
      }
    }
    h[2][u] = 0;
  }
```

```
  for (int i = 2; i <= dfc; ++i) {
    int u = pos[i];
    if (idm[u] != sdm[u]) {
      idm[u] = idm[idm[u]];
    }
  }
  for (int i = dfc; i >= 2; --i) {
    ++ans[pos[i]];
    ans[idm[pos[i]]] += ans[pos[i]];
  }
  ++ans[1];
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= m; ++i) {
    cin >> u >> v;
    add(0, u, v);
    add(1, v, u);
  }
  tar(1);
  for (int i = 1; i <= n; ++i) {
    cout << ans[i] << ' ';
  }
}

```

ZJOI2012 灾难[6]

在 DAG 上求支配树然后求节点 size 即可.

参考代码

```
#include <iostream>
#include <stack>
#include <vector>

using std::cin;
using std::cout;
using std::stack;
using std::vector;

constexpr int MAX = 65536;
int n, x, tot;
int d[MAX], w[MAX], siz[MAX], p[MAX], f[MAX][17];
vector<int> e[MAX], g[MAX], h[MAX];
stack<int> s;

void topo() {
  s.push(0);
  for (int i = 1; i <= n; ++i) {
    if (!w[i]) {
```

```
      e[0].push_back(i);
      g[i].push_back(0);
      ++w[i];
    }
  }
  while (!s.empty()) {
    int x = s.top();
    s.pop();
    p[++tot] = x;
    for (int i : e[x]) {
      --w[i];
      if (!w[i]) {
        s.push(i);
      }
    }
  }
}

int lca(int u, int v) {
  if (d[u] < d[v]) {
    std::swap(u, v);
  }
  for (int i = 15; i >= 0; --i) {
    if (d[f[u][i]] >= d[v]) {
      u = f[u][i];
    }
  }
  if (u == v) {
    return u;
  }
  for (int i = 15; i >= 0; --i) {
    if (f[u][i] != f[v][i]) {
      u = f[u][i];
      v = f[v][i];
    }
  }
  return f[u][0];
}

void dfs(int x) {
  siz[x] = 1;
  for (int i : h[x]) {
    dfs(i);
    siz[x] += siz[i];
  }
}

void build() {
  for (int i = 2; i <= n + 1; ++i) {
    int x = p[i], y = g[x][0];
    for (int j = 1, q = g[x].size(); j < q; ++j) {
      y = lca(y, g[x][j]);
```

```
    }
    h[y].push_back(x);
    d[x] = d[y] + 1;
    f[x][0] = y;
    for (int i = 1; i <= 15; ++i) {
      f[x][i] = f[f[x][i - 1]][i - 1];
    }
  }
}

int main() {
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    while (true) {
      cin >> x;
      if (!x) {
        break;
      }
      e[x].push_back(i);
      g[i].push_back(x);
      ++w[i];
    }
  }
  topo();
  build();
  dfs(0);
  for (int i = 1; i <= n; ++i) {
    cout << siz[i] - 1 << '\n';
  }
  return 0;
}

```

外部参考资料

[1] 一篇关于网络流的论文 [2] 有效的求解算法 [3] 一篇论文

[4] ZJOI2012 灾难 [5] 洛谷 P5180【模板】支配树 [6] ZJOI2012 灾难

11.30 图上随机游走

本页介绍图上随机游走问题. 主要从网格图、稀疏图、一般图三个角度进行探究，介绍了解决这类问题的各种方法，并对比了它们在解决各种问题时的优缺点.

定义

给定一张有向简单图 $G=(V,E)(V=\{v_1,v_2,\cdots,v_{|V|}\})$ 和起点 $s\in V$，终点 $t\in V$，每条边 $e=(x,y)$ 有正权值 w_e，满足 $\forall x\in V\setminus\{t\}$，$\sum_{(x,y)\in E}w_{(x,y)}=1$，且对于任意点 x 都存在一条从 x 出发到达 t 的路径. 有一枚棋子从起点出发，每秒从当前所在点 x 以 $w_{(x,y)}$ 的概率选择出边 (x,y) 并走向 y，到达终点则停止，求期望花费时间.

事实上，这个问题也可以写成矩阵的形式．定义矩阵 P：

$$P_{x,y}=\begin{cases}w_{(x,y)} & \text{if } (x,y)\in E \text{ and } x\neq t\\ 0 & \text{if } (x,y)\notin E \text{ or } x=t\end{cases}$$

要求的答案即为：

$$\sum_{k\geq 0} k\times \left(P^k\right)_{s,t}$$

其中 $\left(P^k\right)_{s,t}$ 表示走了 k 步第一次到达终点的概率．当图有限且所有点都能到达终点时，由 P 的定义可以证明其特征值都小于 1，所以答案一定是收敛的．

为了方便描述，在本页中如无特殊说明，均用 n 代指 $|V|$，m 代指 $|E|$．

另外，在本页中，稀疏图指边数和点数同阶的图．

网格图

例题 1 Circles of Waiting[1]

有一枚棋子起始被放在平面直角坐标系的 $(0,0)$ 点．每秒棋子会随机移动．假设它当前在 (x,y)，它下一秒有 p_1 的概率移动到 $(x-1,y)$，p_2 的概率移动到 $(x,y-1)$，p_3 的概率移动到 $(x+1,y)$，p_4 的概率移动到 $(x,y+1)$．保证 $p_1+p_2+p_3+p_4=1$．求期望经过多少时间它会移动到一个离原点的欧几里得距离大于 R 的位置．$0\leq R\leq 50$，$p_1,p_2,p_3,p_4>0$，答案对 10^9+7 取模．

朴素做法

记 $f(i,j)$ 表示在棋子在 (i,j) 时移动到一个离原点的欧几里得距离大于 R 的位置的期望时间，转移方程为：

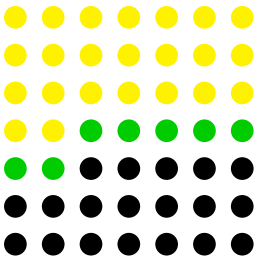
$$f(i,j)=\begin{cases}p_1f(i-1,j)+p_2f(i,j-1)+p_3f(i+1,j)+p_4f(i,j+1)+1 & i^2+j^2\leq R^2\\ 0 & i^2+j^2>R^2\end{cases}$$

由于转移并不存在拓扑序，需要使用高斯消元求解．时间复杂度 $O(R^6)$，无法通过本题．

直接消元法

注意到需要消元的方程的系数大多数都是 0，消元时只对值非 0 的位置进行计算就可以降低复杂度．

考虑消元的过程，将方程按照在坐标系中从上到下，同一层中从左到右的顺序进行消元，将已经消元过的方程染成黄色，与黄色点相邻的点染成绿色，其余点染成黑色，如下图：



接下来要对下一个绿色格子对应的方程进行消元．在这个方程中，只有绿色格子和它下方的第一个黑色格子对应的变量系数可能不为 0；而只有绿色格子和它下方的第一个黑色格子对应的方程中，当前格子对应的变量系数可能不为 0．

注意到绿色格子只有 $O(R)$ 个，所以单个方程消元的时间复杂度为 $O(R^2)$．一共只有 $O(R^2)$ 个方程，所以时间复杂度降低为 $O(R^4)$，可以通过本题．

主元法

方程和变量都有 $O(R^2)$ 个，如果能将规模缩小至 $O(R)$，那么朴素的高斯消元就能通过了.

将每行从左到右第一个格子对应的变量设为主元，共 $2R+1$ 个，设法将其他格子对应的变量用关于这些主元的线性函数表示. 从左到右逐列考虑，对于当前列的每个格子 (i,j)，注意到 $f(i,j), f(i-1,j), f(i,j-1), f(i,j+1)$ 都是已知的关于主元的线性函数，将转移方程移项，有:

$$f(i+1,j)=\frac{f(i,j)-p_1f(i-1,j)-p_2f(i,j-1)-p_4f(i,j+1)-1}{p_3}$$

这样我们就能得到 $f(i+1,j)$ 关于主元的线性函数表示. 如果 $(i+1,j)$ 已经离原点欧几里得距离超过 R 了, 则可以得到一个方程: $f(i+1,j)=0$. 最终, 会得到 $2R+1$ 个方程, 对这些方程进行高斯消元即可.

在递推关于主元的线性函数的阶段，共有 $O(R^2)$ 个变量，递推单个变量需要花费 $O(R)$ 的时间；之后，将问题的规模缩减到了 $O(R)$. 两部分的时间复杂度均为 $O(R^3)$，总的时间复杂度也为 $O(R^3)$，可以通过本题.

两种做法的对比

下面从多种方面对比两种做法:

从时间复杂度方面, 主元法在网格图上的最坏时间复杂度为 $O(n\sqrt{n})$ (当网格图长和宽都为 $O(\sqrt{n})$ 级别时时间复杂度最高)，直接消元法在网格图上最坏时间复杂度为 $O(n^2)$，主元法较优.

从精度方面，对于一些需要进行实数计算而不是取模的题目，直接消元法的精度优于主元法.

从适用性方面，两种做法适用于不同的方面.

当网格图中存在障碍或者走某些边的概率为 0 时, 对于每个障碍或者概率为 0 的边主元法需要增加一个主元，当障碍或者概率为 0 的边的数量多于 $O(R)$ 时，主元法的时间复杂度会增加，而直接消元法的时间复杂度仍然不变.

但主元法还可以做类似于网格图的转移方程的消元，例如 $f(i,j)=p_1f(i+1,j)+p_2f(i,j+1)+p_3f(\mathrm{pre}(i,j))+1$，其中 $\mathrm{pre}(i,j)=(x,y)(x\le i,y\le j)$ 是问题给定的值，而直接消元法的复杂度分析在这种模型中并不适用.

除此以外, 网格图邻接矩阵行列式的计算, 也不能使用主元法, 只能用直接消元法来优化时间复杂度.

综上，两种做法各有所长，需要根据具体题目分析采用不同的做法.

稀疏图

> **例题 2 Expected Value**
>
> 给定一张简单无向连通稀疏图 $G=(V,E)$，有一枚棋子起始被放在 v_1，每秒棋子会从与当前点相连的边中等概率选择一条走到出边指向的点，求到达 v_n 的期望时间. $n\le 2000$，答案对 p 取模，p 是在区间 $[10^9,1.01\times 10^9]$ 内随机生成的一个质数.

基础知识

定义 4.1. 所有满足 $p(A)=0$ 的多项式 $p(\lambda)$ 称为矩阵 A 的零化多项式.

定义 4.2. 记 I_n 表示 n 阶单位矩阵，定义一个 $n\times n$ 矩阵 A 的特征多项式为 $p(\lambda)=\det(\lambda I_n-A)$，其中 det 表示一个矩阵的行列式. 不难发现，一个 n 阶矩阵 A 的特征多项式的次数不超过 n.

定理 4.2.（Cayley–Hamilton 定理）任意矩阵的特征多项式是它的零化多项式.

所以，一个 n 阶矩阵的次数最小的零化多项式的次数也不超过 n.

求解原问题

注意到，期望走的时间 $E(t)=\sum_{i\geq 0}\Pr[t>i]$，如果我们能求出走了 i 步还没有结束的概率，对所有 $i\geq 0$ 求和即为答案.

记 $f(i,j)$ 表示走了 i 步，当前停留在 j，且没有走到过 n 的概率，那么有：

$$f(i,j)=\sum_{(k,j)\in E}\frac{f(i-1,k)}{\deg_k}(j\neq n)$$

其中 $\deg_k$ 表示 k 的度数.

注意到 f 的转移与 i 无关，可以认为一次转移是乘上了一个矩阵，即 $fi+1=f_iM$. 由于 M 的最小零化多项式次数不超过 n，所以 f 的最短递推式长度也不超过 n，故 $\Pr[t>i]=\sum_{j=1}^{n-1}f(i,j)$ 的最短递推式长度也不超过 n. 我们可以在 $O(nm)$ 的时间求出 $\Pr[t>0],\Pr[t>1],\cdots,\Pr[t>3n]$，然后使用 *Berlekamp–Massey* 算法，在 $O(n^2)$ 的时间内求解出 $\Pr[t>i]$ 的最短递推式.

考虑求一个 k 阶线性递推序列 a 的生成函数. 不妨设 $i\geq i_0$ 时 $a_i=\sum_{j=1}^{k}c_ja_{i-j}$，记 a 和 c 的生成函数为 $A(x)$ 和 $C(x)$，那么 $A(x)=A(x)C(x)+A_0(x)$，其中 $A_0(x)$ 是由 $i<i_0$ 的项决定的.

回到原问题，由于我们能求出 $\Pr[t>i]$ 的最短递推式，则我们可以求出 $C(x)$ 和 $A_0(x)$（定义与上一段相同），移项得 $A(x)=\frac{A_0(x)}{1-C(x)}$. 我们要求的是 $\sum_{i\geq 0}[x^i]A(x)$，不难发现这个值就等于 $A(1)$，将 $x=1$ 带入原问题求解即可. 由于模数是随机质数，可以认为分母不会为 0.

这样，我们就在 $O(nm+n^2)$ 的时间复杂度内解决了本题. 如果图 G 的点数与边数同阶，本题中时间复杂度可以认为是 $O(n^2)$.

一般图

例题 3 Frank

给定一张简单强连通有向图 $G=(V,E)$，对于所有 $1\leq s\leq n$，$1\leq t\leq n$，$s\neq t$. 回答下面的问题：有一枚棋子起始被放在 v_s，每秒棋子会从当前点的出边中等概率选择一条走到出边指向的点，求到达 v_t 的期望时间. $3\leq n\leq 400$.

分析和转化

记 $p_{i,j}$ 表示棋子在 i 时，选择出边 (i,j) 走到 j 的概率，特别地，当出边不存在时概率为 0. 记 $f_{i,j}$ 表示 i 随机游走到 j 的期望时间，特别地，$f_{i,i}=0$. 当 $i\neq j$ 时，转移方程为：

$$f_{i,j}=1+\sum_{1\leq k\leq n}p_{i,k}f_{k,j}$$

当 $i=j$ 时，记 g_i 表示从 i 开始随机游走，第一次回到 i 的期望时间，那么：

$$f_{i,i}=1-g_i+\sum_{1\leq k\leq n}p_{i,k}f_{k,i}$$

为了方便观察，我们将转移方程写成矩阵的形式. 记 P 表示这个图的转移矩阵，F 表示答案矩阵，I 表示 n 阶单位矩阵，J 表示 n 阶全 1 矩阵，G 是一个 n 阶矩阵，满足 $G_{i,i}=g_i$，其他位置为 0，则：

$$F=J-G+PF$$

如果我们能求出 G，那么我们只需要解方程：

$$(I-P)F=J-G$$

G 的求法

定义 5.1. 定义一个 n 阶转移矩阵 P 的稳态分布为一个 n 维向量 π，满足 $\sum_{i=1}^{n}\pi_i=1$，$\pi P=\pi$. 其中，π 每一维的值都在区间 $[0,1]$ 内.

我们很容易找出稳态分布的实际意义．如果某个时刻棋子有 π_i 的概率停留在 v_i，则在之后的任意时刻，棋子仍然满足这个概率分布．我们可以在 $O(n^3)$ 的时间通过高斯消元解方程来求出 π，那么 π 与 G 有什么关系呢?

定理 5.1. 对于任意 $1 \leq i \leq n$，有 $\pi_i g_i = 1$.

证明

由 $F = J - G + PF$，移项得:

$$G = PF + J - F$$

两边同时在左边乘上 π 有:

$$\pi G = \pi PF + \pi J - \pi F$$

由 π 的定义有 $\pi P = \pi$，故:

$$\pi G = \pi J$$

所以:

$$\pi_i g_i = \sum_{j=1}^{n} \pi_j = 1$$

原命题得证.

所以，通过引入稳态分布，我们可以在 $O(n^3)$ 的时间内求解 G.

求解原问题

在解方程的过程中，我们发现一个问题：$(I - P)$ 并不满秩，不能通过乘逆矩阵的方法求解.

定义 5.2. 定义一个有向图 $G = (V, E)$ 的以 $r \in V$ 为根的有向生成树是 G 的一个子图 $T = (V, A)$，满足:

1. 对于任意 $i \neq r$，i 的出度为 1.
2. r 的出度为 0.
3. T 中不存在环.

引理 5.1.（有向图上的矩阵树定理）对于一个有向图 G，记 D 表示其出度矩阵，即 $D_{i,i} = d_i$，$D_{i,j} = 0 (i \neq j)$，其中 d_i 表示 i 的出度，记 A 表示其邻接矩阵，则其以 r 为根的有向生成树个数为 $D - A$ 去掉第 r 行第 r 列后的行列式.

定理 5.2. 对于一个强连通图 $G = (V, E)$ 的转移矩阵 P，$(I - P)$ 的秩为 $n - 1$.

证明

因为对矩阵某一行乘上一个非零常数其秩不改变，所以我们将 $(I - P)$ 的第 i 行乘上 v_i 的出度，得到一个新的矩阵 L，只需证明 L 的秩为 $n - 1$ 即可.

由于 L 每行的和均为 0，对 L 的所有列向量求和，会得到零向量，即这些向量线性相 关，所以 L 的秩不为 n.

不难发现 L 等于图 G 的出度矩阵减去其邻接矩阵，由引理 5.1 得 L 去掉第 i 行第 i 列 后行列式表示以 v_i 为根的有向生成树个数.

由于 G 是强连通的，所以以任意点为根的有向生成树个数均不为 0，即 L 去掉第 i 行 第 i 列之后仍然满秩.

因为加上一列秩不会变小，所以 L 去掉第 i 行后所有行向量线性无关．故 L 的秩为 $n - 1$.

回到原问题，考虑求解原问题中的方程．为了方便，我们将方程写成 $AX=B$ 的形式， 其中 A，B 已知，需要求解 X．由于 A 不满秩，解有无数个，我们首先求出一组特解．将 A 和 B 一起做高斯消元．把 A 的前 $n-1$ 行消成只有主对角线和第 n 列有值的形 式，最后一行消成全 0，即下列形式：

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & a_1 \\ 0 & 1 & 0 & \cdots & 0 & a_2 \\ 0 & 0 & 1 & \cdots & 0 & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & a_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} X = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,n-1} & b_{1,n} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,n-1} & b_{2,n} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,n-1} & b_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ b_{n-1,1} & b_{n-1,2} & b_{n-1,3} & \cdots & b_{n-1,n-1} & b_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

令 $X_{n,i}=0$，可以解出一组特解，记为 Y．接下来将特解调整为真正的解．

注意到 $X_{n,i}=0$, 考虑组合意义有 $Y_{i,j}=1+Y_{j,j}+P_{i,k}X_{k,j}$, 不难解出 $X_{i,j}=Y_{i,j}-Y_{j,j}$. 最终在 $O(n^3)$ 的时间复杂度内解决了这个问题.

参考

1. 浅谈图模型上的随机游走问题．IOI2019 中国国家候选队论文集（pp. 17-26）

外部参考资料

[1] Circles of Waiting



第十二章

计算几何

12.1 计算几何部分简介

利用计算机建立数学模型解决几何问题.

12.2 二维计算几何基础

我们将需要解决的几何问题的范围限制在二维平面内，这样就涉及到二维计算几何的基本知识. 在研究时，通常把图形放在平面直角坐标系或极坐标系下，这样解决问题就会方便很多.

前置技能

如并不了解:

- 几何基础
- 平面直角坐标系
- 向量（包括向量积）
- 极坐标与极坐标系

请先阅读 向量→9.14.2@p1707 和 极坐标→9.9@p1149.

图形的记录

点

在平面直角坐标系下，点用横纵坐标表示，比如点 $(5,2)$，点 $(-1,0)$ 等. 由于坐标是有序数对，因此可以利用 `std::pair`[1] 存储，也可以定义结构体存储.

在极坐标系下，点用极径与极角表示，使用类似于直角坐标系的存储方式进行存储即可.

向量

由于向量的坐标表示与点相同，所以只需要像点一样存储向量即可.

在极坐标系下，与点同理.

线

在极坐标系中表示线是较为困难的，因此我们仅讨论直角坐标系中的情况.

直线与射线

一般在求解数学问题时，我们用解析式表示一条直线. 直线的解析式有一般式 $Ax+By+C=0$、斜截式 $y=kx+b$、截距式 $\frac{x}{a}+\frac{y}{b}=1$ 等多种形式，但其形式最终服务于方程求解析解. 在解决一些计算几何问题中，使用解析式表示直线会增加解决问题的难度.

在计算几何中，通常使用直线上一点与直线的方向向量表示一条直线，用射线的端点与射线的方向向量表示一条射线. 方向向量是一个与直线平行的非零向量，通常情况下取单位向量. 与一条直线平行的单位向量有两个，可以根据题目选择任意一个；而对于射线，方向向量通常取方向与射线延伸方向相同的那个.

线段

记录一条线段只需要记录其左右端点即可.

多边形

使用数组按一定顺序记录多边形的每个顶点即可. 对于简单多边形，一般按顺时针或逆时针顺序记录各个顶点.

特殊地，记录矩形时，如果矩形的各边均与坐标轴平行，只需记录其对角线上的两个顶点.

曲线图形

一些特殊曲线，如函数图像等一般记录其解析式．对于圆，直接记录其圆心和半径即可．

基本公式

正弦定理

在 △ ABC 中，若角 A, B, C 所对边分别为 a, b, c，则有：

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R$$

其中，R 为 △ ABC 的外接圆半径．

余弦定理

在 △ ABC 中，若角 A, B, C 所对边分别为 a, b, c，则有：

$$a^2 = b^2 + c^2 - 2bc\cos A$$
$$b^2 = a^2 + c^2 - 2ac\cos B$$
$$c^2 = a^2 + b^2 - 2ab\cos C$$

上述公式的证明略．均为人教版高中数学 A 版必修二内容（旧教材为必修五）．

基本操作

判断一个点在直线的哪边

我们有直线上的一点 P 的直线的方向向量 $\mathbf{v}$，想知道某个点 Q 在直线的哪边．这里所指的「哪边」是相对于方向向量的，也就是说，判断时我们面朝方向向量所指的方向进行判断．

我们利用向量积的性质，算出 $\mathbf{v} \times \overrightarrow{PQ}$．根据右手定则，如果向量积为负，则 Q 在直线右侧；如果向量积为 0，则 Q 在直线上；如果向量积为正，则 Q 在直线左侧．

快速排斥实验与跨立实验

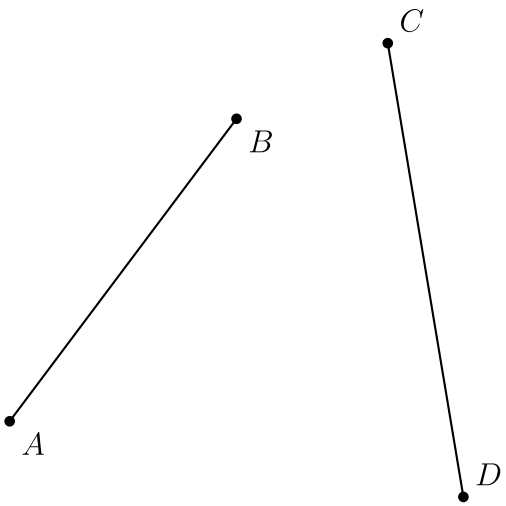
我们现在想判断两条线段是否相交．

首先特判一些特殊情况．如果两线段平行，自然不能相交．这种情况通过判断线段所在直线的斜率是否相等即可．

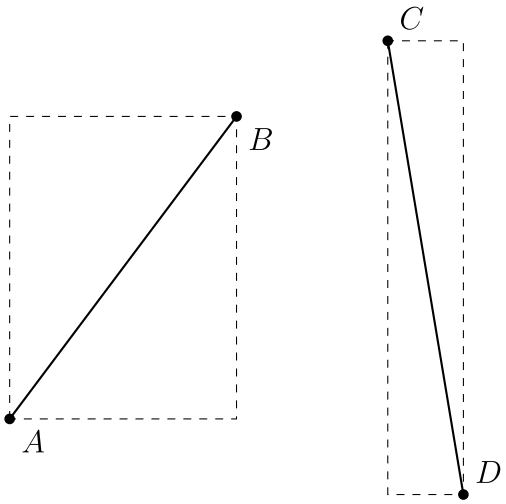
当然，如果两线段重合或部分重合，或者两线段的交点为其中一条线段的端点，只需要判断是否有三点共线的情况即可．

还有些显然不相交的情况，我们口头上称之为「两条线段离着太远了」．规定「一条线段的区域」为以这条线段为对角线的，各边均与某一坐标轴平行的矩形所占的区域，那么可以发现，如果两条线段没有公共区域，则这两条线段一定不相交．

比如有以下两条线段：



它们占用的区域是这样的：



于是可以快速地判断出来这两条线段不相交.

这就是 **快速排斥实验**. 上述情况称作 **未通过快速排斥实验**.

未通过快速排斥实验是两线段无交点的 **充分不必要条件**，我们还需要进一步判断.

因为两线段 a, b 相交，b 线段的两个端点一定分布在 a 线段所在直线两侧；同理，a 线段的两个端点一定分布在 b 线段所在直线两侧. 我们可以直接判断一条线段的两个端点相对于另一线段所在直线的位置关系，如果不同，则两线段相交，反之则不相交. 如上一节所说，直线与点的位置关系我们可以利用向量积判断.

这就是 **跨立实验**，如果对于两线段 a, b，b 线段的两个端点分布在 a 线段所在直线的两侧，**且** a 线段的两个端点分布在 b 线段所在直线的两侧，我们就说 a, b 两线段 **通过了跨立实验**，即两线段相交.

注意到当两条线段共线但不相交时也可以通过跨立实验，因此想要准确判断还需要与快速排斥实验结合.

判断一点是否在多边形内部

在计算几何中，这个问题被称为 PIP 问题[2]，已经有一些成熟的解决方法，下面依次介绍.

光线投射算法 (Ray casting algorithm)

在 这里[3] 可以看到最原始的思路.

我们先特判一些特殊情况，比如「这个点离多边形太远了」. 类似判断两条线段相交，考虑一个能够完全覆盖该多边形的最小矩形，如果这个点不在这个矩形范围内，那么这个点一定不在多边形内. 这样的矩形很好求，只需要知道多边形横坐标与纵坐标的最小值和最大值，坐标两两组合成四个点，就是这个矩形的四个顶点了.

还有点在多边形的某一边或某顶点上，同样十分容易判断. 对于每条边，检查点是否在边上或顶点上即可.

我们考虑以该点为端点引出一条射线，如果这条射线与多边形有奇数个交点，则该点在多边形内部，否则该点在多边形外部，我们简记为 **奇内偶外**. 这个算法同样被称为奇偶规则 (Even-odd rule).

由于 Jordan curve theorem[4]，我们知道，这条射线每次与多边形的一条边相交，就切换一次与多边形的内外关系，所以统计交点数的奇偶即可.

对于选取的射线，可以随机选取这条射线所在直线的斜率，建议为无理数以避免出现射线与多边形某边重合的情况.

在原版代码中，使用的是记录多边形的数组中最后一个点作为射线上一点，这样统计时，如果出现射线过多边形某边或某顶点时，可以规定射线经过的点同在射线一侧，进而做跨立实验即可.

> ⚠ **注意**
>
> 光线投射算法只适用于简单多边形. 对于边自交的多边形并不适用 Jordan curve theorem，因此无法判断.

回转数算法 (Winding number algorithm)

回转数是数学上的概念，是平面内闭合曲线逆时针绕过该点的总次数. 很容易发现，当回转数等于 0 的时候，点在曲线外部. 这个算法同样被称为非零规则 (Nonzero-rule).

如何计算呢? 我们把该点与多边形的所有顶点连接起来，计算相邻两边夹角的和. 注意这里的夹角是 **有方向的**. 如果夹角和为 0，则这个点在多边形外，否则在多边形内.

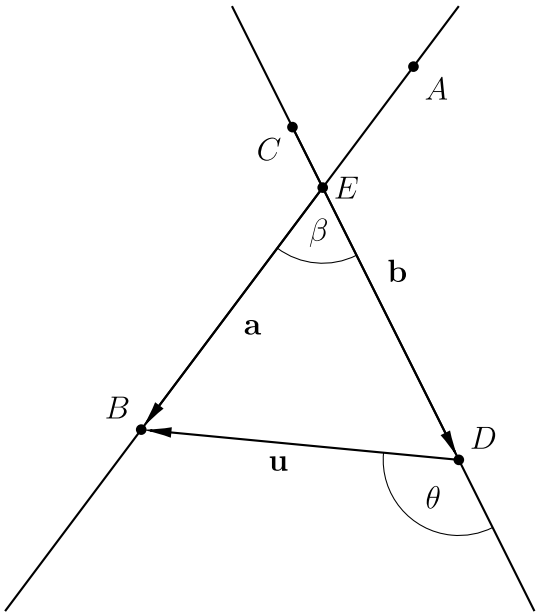
求两条直线的交点

首先，我们需要确定两条直线相交，只需判断一下两条直线的方向向量是否平行即可. 如果方向向量平行，则两条直线平行，交点个数为 0. 进一步地，若两条直线平行且过同一点，则两直线重合.

那么，问题简化为我们有直线 AB, CD 交于一点，想求出交点 E.

如果两直线相交，则交点只有一个，我们记录了直线上的一个点和直线的方向向量，所以我们只需要知道这个点与交点的距离 l，再将这个点沿方向向量平移 l 个单位长度即可.

考虑构造三角形，利用正弦定理求解 l，可以利用向量积构造出正弦定理.



由上图可知，$|\mathbf{a}\times\mathbf{b}|=|\mathbf{a}||\mathbf{b}|\sin\beta$，$|\mathbf{u}\times\mathbf{b}|=|\mathbf{u}||\mathbf{b}|\sin\theta$.

作商得：

$$T=\frac{|\mathbf{u}\times\mathbf{b}|}{|\mathbf{a}\times\mathbf{b}|}=\frac{|\mathbf{u}|\sin\theta}{|\mathbf{a}|\sin\beta}$$

可以看出，$\left|\frac{|\mathbf{u}|\sin\theta}{\sin\beta}\right|=l$. 若绝对值内部式子取值为正，代表沿 $\mathbf{a}$ 方向平移，反之则为反方向.

同时，我们将 T 直接乘上 $\mathbf{a}$，就自动出现了直线的单位向量，不需要进行其他消去操作了.

于是，只需要将点 B 减去 $T\mathbf{a}$ 即可得出交点.

求任意多边形的周长和面积

求任意多边形的周长

直接计算即可，简洁即美德.

求任意多边形的面积

考虑向量积的模的几何意义，我们可以利用向量积完成.

将多边形上的点逆时针标记为 $p_1,p_2,\cdots,p_n$，再任选一个辅助点 O，记向量 $\mathbf{v}_i=p_i-O$，那么这个多边形面积 S 可以表示为：

$$S=\frac{1}{2}\left|\sum_{i=1}^{n}\mathbf{v}_i\times\mathbf{v}_{(i\bmod n)+1}\right|$$

圆与直线相关

求直线与圆的交点

首先判断直线与圆的位置关系. 如果直线与圆相离则无交点，若相切则可以利用切线求出切点与半径所在直线，之后转化为求两直线交点.

若有两交点，则可以利用勾股定理求出两交点的中点，然后沿直线方向加上半弦长即可.

求两圆交点

首先我们判断一下两个圆的位置关系，如果外离或内含则无交点，如果相切，可以算出两圆心连线的方向向量，然后利用两圆半径计算出平移距离，最后将圆心沿这个方向向量进行平移即可.

如果两圆相交，则必有两个交点，并且关于两圆心连线对称．因此下面只说明一个交点的求法，另一个交点可以用类似方法求出．

我们先将一圆圆心与交点相连，求出两圆心连线与该连线所成角．这样，将两圆心连线的方向向量旋转这个角度，就是圆心与交点相连形成的半径的方向向量了．沿方向向量方向将圆心平移半径长度即可得到一个交点．

极角序

一般来说，这类题需要先枚举一个极点，然后计算出其他点的极坐标，在极坐标系下按极角的顺序解决问题．

「JOISC 2014 Day4」两个人的星座[5]

平面内有 n 个点，有三种颜色，每个点的颜色是三种中的一种．求不相交的三色三角形对数．$6 \leq n \leq 3000$．

题解

如果两个三角形不相交，则一定可以做出两条内公切线，如果相交或内含是做不出内公切线的．三角形的公切线可以类比圆的公切线．

先枚举一个原点，记为 O，以这个点为极点，过这个点且与 x 轴平行的直线作为极轴，建立极坐标系，把剩余点按极角由小到大排序．然后统计出在极轴上方和下方的每种点的个数．

然后根据点枚举公切线，记枚举到的点为 P，初始时公切线为极轴．开始统计．那么一定存在一条公切线过点 O 和点 P．因为公切线与三角形不相交，所以一方选择公切线上方的点，另一方一定选择下方的点．然后利用乘法原理统计方案数即可．

统计完后旋转公切线，那么点 P 一定改变了相对于公切线的上下位置，而其他点不动，应该只将它的位置信息改变．这样，可以发现，同一对三角形最终被统计了 4 次，就是同一条公切线会被枚举两次，最后做出的答案应除以 4．

分析一下算法复杂度．我们枚举了一个原点，然后对于每一个原点将剩余点排序后线性统计，因此时间复杂度为 $O(n^2 \log n)$．

代码编写注意事项

由于计算几何经常进行 `double` 类型的浮点数计算，因此带来了精度问题和时间问题．

有些问题，例如求点坐标均为整数的三角形面积，可以利用其特殊性进行纯整数计算，避免用浮点数影响精度．

由于浮点数的读入与计算都比整数慢，所以需要注意程序的常数因子给时间带来的影响．

外部参考资料

[1] `std::pair`



[2] PIP 问题



[3] 这里



[4] Jordan curve theorem



[5]「JOISC 2014 Day4」两个人的星座



12.3 三维计算几何基础

Authors: shuzhouliu

三维几何的很多概念与 二维几何→12.2@p3082 是相通的，我们可以用与解决二维几何问题相同的方法来解决三维几何问题.

基本概念

点，向量，直线这些概念和二维几何是相似的，这里不再展开.

平面

我们可以用平面上的一点 $P_0(x_0,y_0,z_0)$ 和该平面的法向量（即垂直于该平面的向量）$\boldsymbol{n}$ 来表示一个平面.

因为 $\boldsymbol{n}$ 垂直于平面，所以 $\boldsymbol{n}$ 垂直于该平面内的所有直线. 换句话说，设 $\boldsymbol{n}=(A,B,C)$，则该平面上的点 $P(x,y,z)$ 都满足 $\boldsymbol{n}\cdot\overrightarrow{PP_0}=0$.

根据向量点积的定义，上式等价于：

$$A(x-x_0)+B(y-y_0)+C(z-z_0)=0$$

整理后得到：

$$Ax+By+Cz-(Ax_0+By_0+Cz_0)=0$$

令 $D=-(Ax_0+By_0+Cz_0)$，则上式变成 $Ax+By+Cz+D=0$. 我们称这个式子为平面的 **一般式**.

基本操作

直线、平面之间的夹角

运用空间向量的知识，空间中直线、平面之间的夹角可以很快求出.

对于两条异面直线 a，b，过空间中一点 P，作 $a'\parallel a$，$b'\parallel b'$，则 a' 与 b' 所成的锐角或直角被称为 a 和 b 两条 **异面直线所成的角**.

对于直线 a 和平面 α，若 a 与 α 相交于 A，过 a 上一点 P 引平面 α 的垂线交 α 于 O，则 a 与 PO 所成角的余角被称为 **直线与平面所成的角**. 特别地，若 $a\parallel\alpha$ 或 $a\subset\alpha$，则它们之间的夹角为 0°.

对于两个平面 α，β，它们的夹角被定义为与两条平面的交线 l 垂直的两条直线 a,b（其中 $a\subset\alpha$，$b\subset\beta$）所成的角.

两直线夹角定义与关系充要条件

- 两直线的方向向量的夹角，叫做两直线的夹角.

有了这个命题，我们就可以得出以下结论：已知两条直线 l_1,l_2，它们的方向向量分别是 $s_1(m_1,n_1,p_1)$，$s_2(m_2,n_2,p_2)$，设 φ 为两直线夹角，我们可以得到 $\cos\varphi=\dfrac{|m_1m_2+n_1n_2+p_1p_2|}{\sqrt{m_1^2+n_1^2+p_1^2}\sqrt{m_2^2+n_2^2+p_2^2}}$.

- $l_1\perp l_2\iff m_1m_2+n_1n_2+p_1p_2=0$
- $l_1\parallel l_2\iff\dfrac{m_1}{m_2}=\dfrac{n_1}{n_2}=\dfrac{p_1}{p_2}$.

三维向量与平面的夹角

当直线与平面不垂直时，直线和它在平面上的投影直线的夹角 φ（$\varphi\in[0,\frac{\pi}{2}]$）称为直线与平面的夹角.

设直线向量 $s(m,n,p)$，平面法线向量 $f(a,b,c)$，那么以下命题成立：

- 角度的正弦值：$\sin\varphi=\dfrac{|am+bn+cp|}{\sqrt{a^2+b^2+c^2}\sqrt{m^2+n^2+p^2}}$
- 直线与平面平行 $\iff am+bn+cp=0$
- 直线与平面垂直 $\iff\dfrac{a}{m}=\dfrac{b}{n}=\dfrac{c}{p}$

点到平面的距离

直线与平面的交点

直接联立直线方程和平面方程即可.

立体几何定理

三正弦定理

设二面角 $M-AB-N$ 的度数为 α，在平面 M 上有一条射线 AC，它和棱 AB 所成角为 β，和平面 N 所成的角为 γ，则 $\sin\gamma=\sin\alpha\cdot\sin\beta$.

三余弦定理

设 O 为平面上一点，过平面外一点 B 的直线 BO 在面上的射影为 AO，OC 为面上的一条直线，那么 $\angle COB,\angle AOC,\angle AOB$ 三角的余弦关系为：$\cos\angle BOC=\cos\angle AOB\cdot\cos\angle AOC$（$\angle AOC$，$\angle AOB$ 只能是锐角）.

参考资料

- 3D 空间基础概念之一：点、向量（矢量）和齐次坐标[1]

外部参考资料

[1] 3D 空间基础概念之一：点、向量（矢量）和齐次坐标

12.4 距离

Authors: Chrogeek, frank-xjh, ChungZH, hsfzLZH1, Marcythm, Planet6174, partychicken, i-Yirannn

欧氏距离

二维空间

定义

欧氏距离，一般也称作欧几里得距离．在平面直角坐标系中，设点 A,B 的坐标分别为 $A(x_1,y_1),B(x_2,y_2)$，则两点间的欧氏距离为：

$$|AB|=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

解释

举个例子，若在平面直角坐标系中，有两点 $A(6,5),B(2,2)$，通过公式，我们很容易得到 A,B 两点间的欧氏距离：

$$|AB|=\sqrt{(2-6)^2+(2-5)^2}=\sqrt{4^2+3^2}=5$$

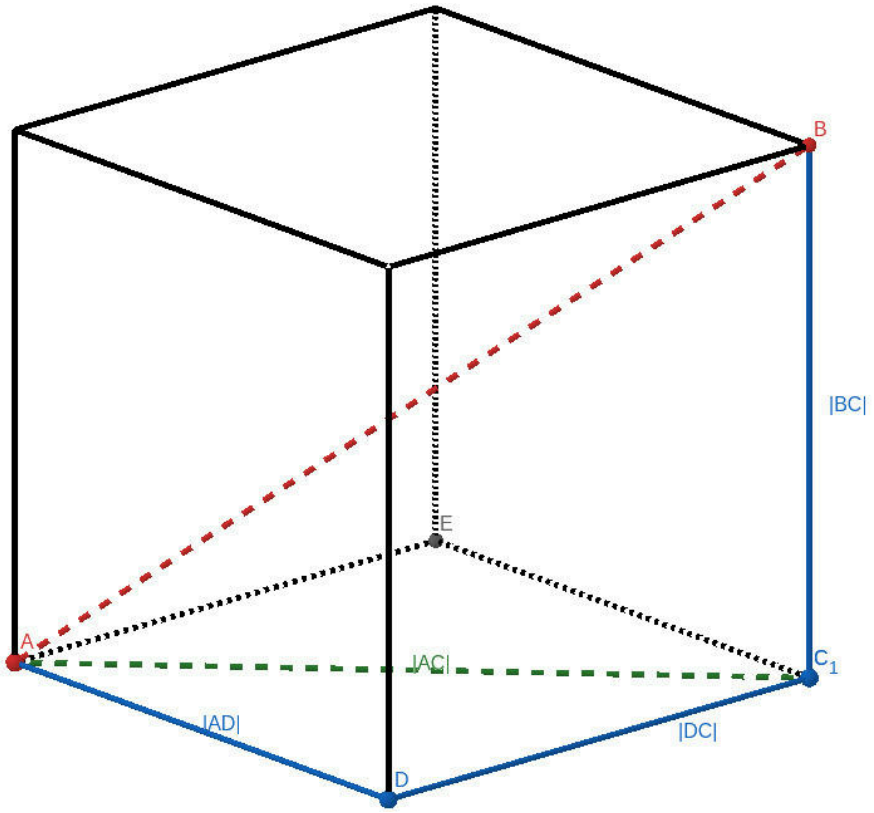
除此之外，$P(x,y)$ 到原点的欧氏距离可以用公式表示为：

$$|\ P\ |=\sqrt{x^2+y^2}$$

n 维空间

引入

那么，三维空间中两点的欧氏距离公式呢？我们来观察下图.



我们很容易发现，在 $\triangle ADC$ 中，$\angle ADC = 90°$；在 $\triangle ACB$ 中，$\angle ACB = 90°$.

$$\begin{aligned}\therefore \quad |AB| &= \sqrt{|AC|^2 + |BC|^2} \\ &= \sqrt{|AD|^2 + |CD|^2 + |BC|^2}\end{aligned}$$

定义

由此可得，三维空间中欧氏距离的距离公式为：

$$|AB| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

$$|P| = \sqrt{x^2 + y^2 + z^2}$$

解释

NOIP2017 提高组 奶酪[1] 就运用了这一知识，可以作为欧氏距离的例题.

以此类推，我们就得到了 n 维空间中欧氏距离的距离公式：对于 $\vec{A}(x_{11}, x_{12}, \cdots, x_{1n})$，$\vec{B}(x_{21}, x_{22}, \cdots, x_{2n})$，有

$$\begin{aligned}||\overrightarrow{AB}|| &= \sqrt{(x_{11} - x_{21})^2 + (x_{12} - x_{22})^2 + \cdots + (x_{1n} - x_{2n})^2} \\ &= \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}\end{aligned}$$

欧氏距离虽然很有用，但也有明显的缺点. 两个整点计算其欧氏距离时，往往答案是浮点型，会存在一定误差.

曼哈顿距离

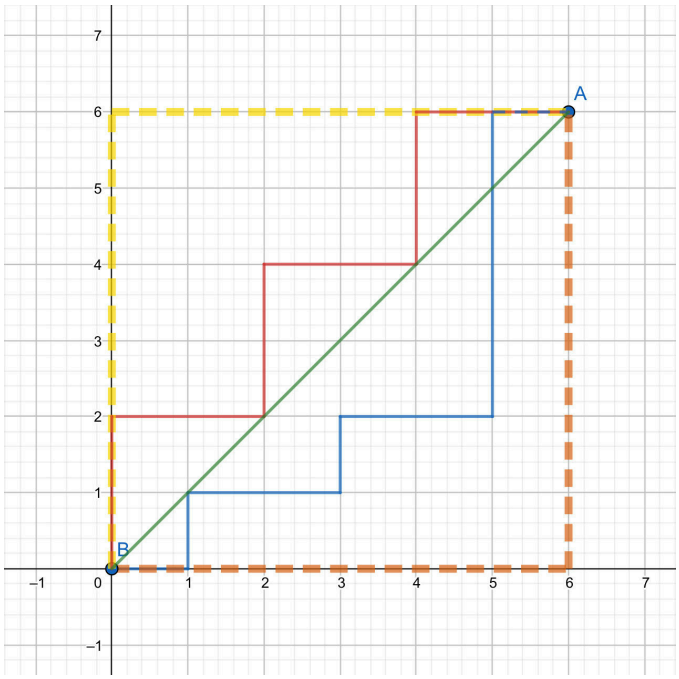
定义

在二维空间内，两个点之间的曼哈顿距离（Manhattan distance）为它们横坐标之差的绝对值与纵坐标之差的绝对值之和. 设点 $A(x_1, y_1), B(x_2, y_2)$，则 A, B 之间的曼哈顿距离用公式可以表示为：

$$d(A,B) = |x_1 - x_2| + |y_1 - y_2|$$

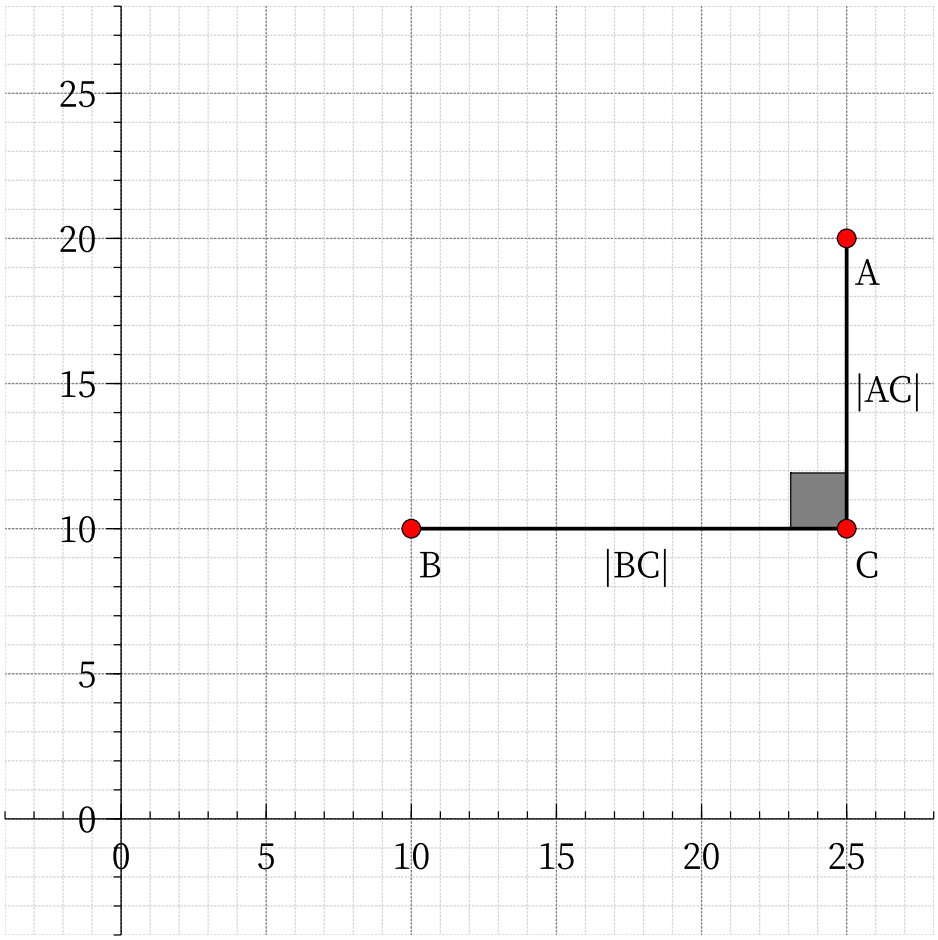
解释

观察下图：



在 A, B 间，黄线、橙线都表示曼哈顿距离，而红线、蓝线表示等价的曼哈顿距离，绿线表示欧氏距离.

同样的例子，在下图中 A, B 的坐标分别为 $A(25, 20), B(10, 10)$.



通过公式，我们很容易得到 A, B 两点间的曼哈顿距离：

$$d(A,B) = |\ 20-10\ | + |\ 25-10\ | = 10+15 = 25$$

经过推导，我们得到 n 维空间的曼哈顿距离公式为：

$$\begin{aligned} d(A,B) &= |\ x_1-y_1| + |\ x_2-y_2| + \cdots + |\ x_n-y_n| \\ &= \sum_{i=1}^{n} |\ x_i-y_i| \end{aligned}$$

性质

除了公式之外，曼哈顿距离还具有以下数学性质：

- 非负性：曼哈顿距离是一个非负数，即 $d(i,j) \geq 0$.
- 统一性：一个点到自身的曼哈顿距离为 0，即 $d(i,i) = 0$.
- 对称性：A 到 B 与 B 到 A 的曼哈顿距离相等，即 $d(i,j) = d(j,i)$.
- 三角不等式：从点 i 到 j 的直接距离不会大于途经的任何其它点 k 的距离，即 $d(i,j) \leq d(i,k) + d(k,j)$.

例题

P5098「USACO04OPEN」Cave Cows 3[2]

根据题意，对于式子 $|\ x_1-x_2| + |\ y_1-y_2|$，我们可以假设 $x_1-x_2 \geq 0$，根据 y_1-y_2 的符号分成两种情况：

- $(y_1 - y_2 \geq 0) \rightarrow |x_1 - x_2| + |y_1 - y_2| = x_1 + y_1 - (x_2 + y_2)$
- $(y_1 - y_2 < 0) \rightarrow |x_1 - x_2| + |y_1 - y_2| = x_1 - y_1 - (x_2 - y_2)$

只要分别求出 $x + y, x - y$ 的最大值和最小值即能得出答案.

参考代码

- C++

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;

int main() {
  int n, x, y, minx = 0x7fffffff, maxx = 0, miny = 0x7fffffff, maxy = 0;
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &x, &y);
    minx = min(minx, x + y), maxx = max(maxx, x + y);
    miny = min(miny, x - y), maxy = max(maxy, x - y);
  }
  printf("%d\n", max(maxx - minx, maxy - miny));
  return 0;
}
```

- Python

```python
minx = 0x7FFFFFFF
maxx = 0
miny = 0x7FFFFFFF
maxy = 0
n = int(input())
for i in range(1, n + 1):
    x, y = map(lambda x: int(x), input().split())
    minx = min(minx, x + y)
    maxx = max(maxx, x + y)
    miny = min(miny, x - y)
    maxy = max(maxy, x - y)
print(max(maxx - minx, maxy - miny))
```

其实还有第二种做法，那就是把曼哈顿距离转化为切比雪夫距离求解，最后部分会讲到.

切比雪夫距离

定义

切比雪夫距离（Chebyshev distance）是向量空间中的一种度量，二个点之间的距离定义为其各坐标数值差的最大值. [1]

在二维空间内，两个点之间的切比雪夫距离为它们横坐标之差的绝对值与纵坐标之差的绝对值的最大值. 设点 $A(x_1, y_1), B(x_2, y_2)$，则 A, B 之间的切比雪夫距离用公式可以表示为:

$$d(A, B) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

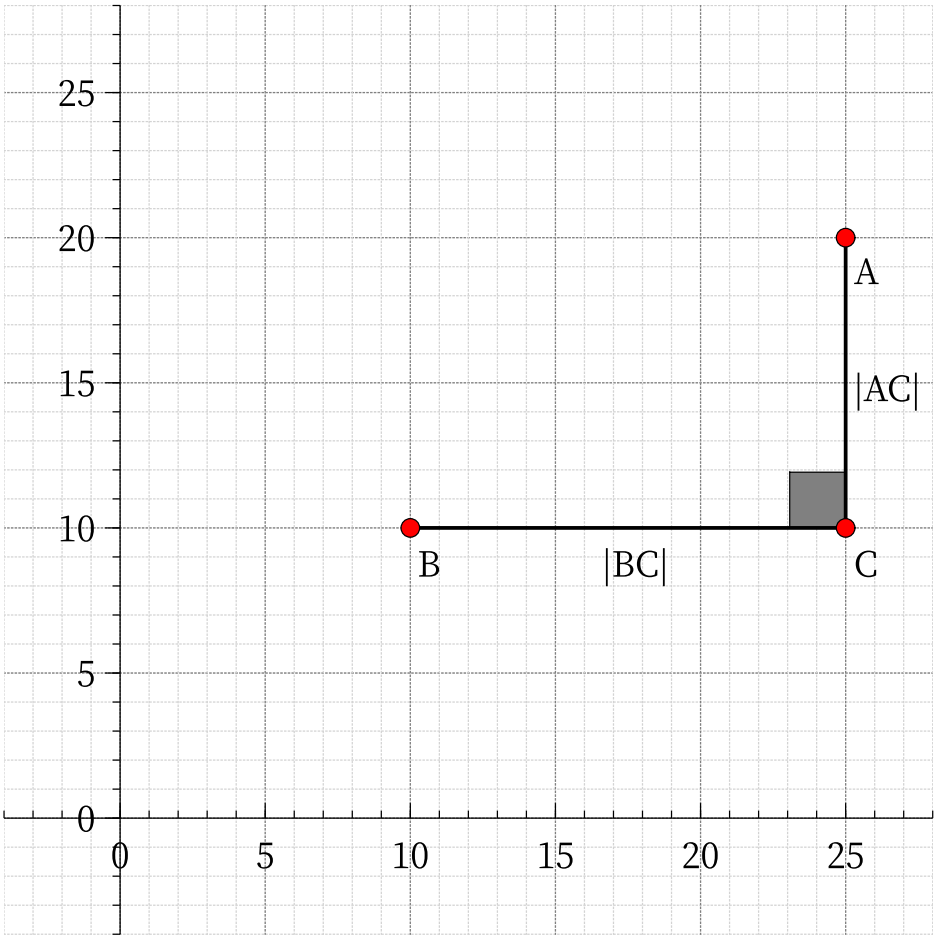
n 维空间中切比雪夫距离的距离公式可以表示为:

[1] 切比雪夫距离 - 维基百科，自由的百科全书[3]

$$\begin{aligned} d(x,y) &= \max\{|\ x_1-y_1|,|\ x_2-y_2|,\cdots,|\ x_n-y_n|\} \\ &= \max\{|\ x_i-y_i|\}(i\in[1,n]) \end{aligned}$$

解释

仍然是这个例子，下图中 A,B 的坐标分别为 $A(25,20),B(10,10)$.



$$d(A,B)=\max(|\ 20-10\ |,|\ 25-10\ |)=\max(10,15)=15$$

曼哈顿距离与切比雪夫距离的相互转化

过程

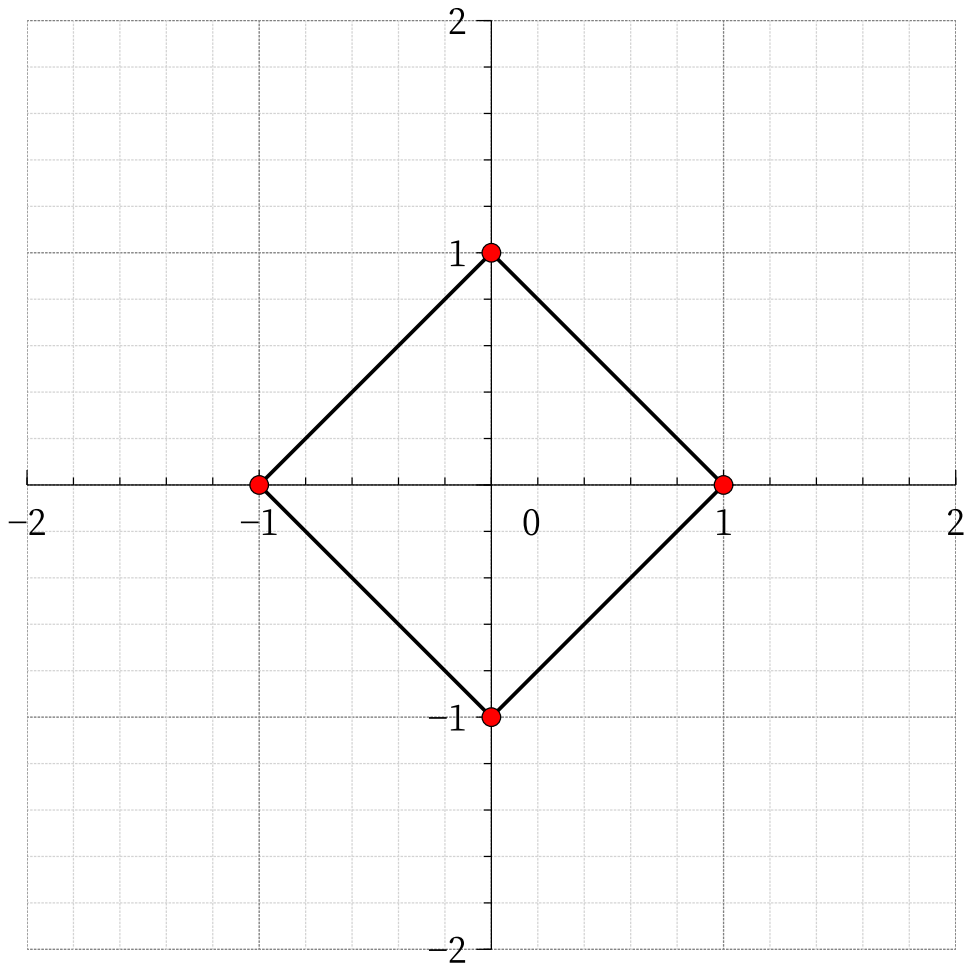
首先，我们考虑画出平面直角坐标系上所有到原点的曼哈顿距离为 1 的点.

通过公式，我们很容易得到方程 $|\ x\ |+|\ y\ |=1$.

将绝对值展开，得到 4 个 一次函数，分别是：

$$\begin{aligned} y&=-x+1(x\geq 0,y\geq 0) \\ y&=x+1\quad(x\leq 0,y\geq 0) \\ y&=x-1\quad(x\geq 0,y\leq 0) \\ y&=-x-1(x\leq 0,y\leq 0) \end{aligned}$$

将这 4 个函数画到平面直角坐标系上，得到一个边长为 $\sqrt{2}$ 的正方形，如下图所示：



正方形边界上所有的点到原点的 曼哈顿距离 都是 1.

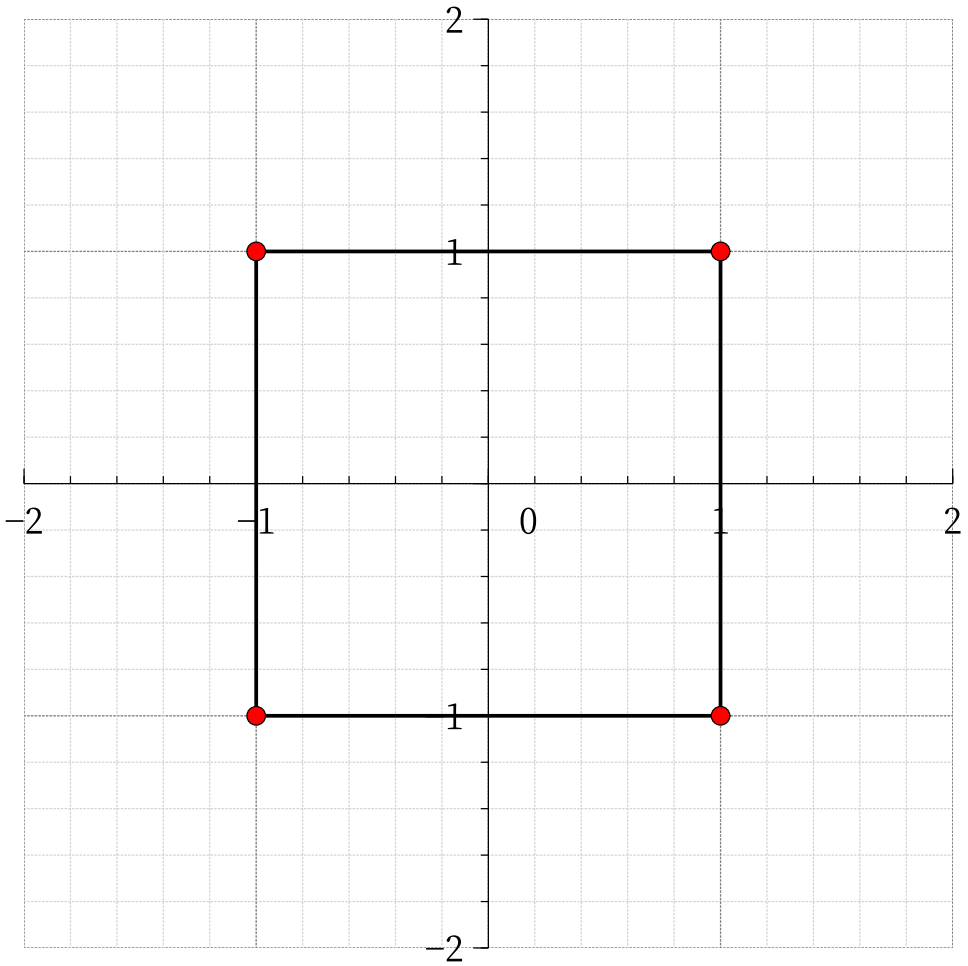
同理，我们再考虑画出平面直角坐标系上所有到原点的 切比雪夫距离 为 1 的点.

通过公式，我们知道 $\max(|\ x\ |, |\ y\ |) = 1$.

我们将式子展开，也同样可以得到 4 条线段，分别是:

$$
\begin{aligned}
y &= 1 \quad (-1 \leq x \leq 1) \\
y &= -1 \ (-1 \leq x \leq 1) \\
x &= 1, \quad (-1 \leq y \leq 1) \\
x &= -1, (-1 \leq y \leq 1)
\end{aligned}
$$

画到平面直角坐标系上，可以得到一个边长为 2 的正方形，如下图所示:



正方形边界上所有的点到原点的切比雪夫距离都是 1.

将这两幅图对比，我们会神奇地发现：

这 2 个正方形是相似图形.

证明

所以，曼哈顿距离与切比雪夫距离之间会不会有联系呢?

接下来我们简略证明一下：

假设 $A(x_1,y_1),B(x_2,y_2)$,

我们把曼哈顿距离中的绝对值拆开，能够得到四个值，这四个值中的最大值是两个非负数之和，即曼哈顿距离. 则 A,B 两点的曼哈顿距离为：

$$\begin{aligned} d(A,B) &= |x_1-x_2|+|y_1-y_2| \\ &= \max\{x_1-x_2+y_1-y_2, x_1-x_2+y_2-y_1, x_2-x_1+y_1-y_2, x_2-x_1+y_2-y_1\} \\ &= \max(|(x_1+y_1)-(x_2+y_2)|, |(x_1-y_1)-(x_2-y_2)|) \end{aligned}$$

我们很容易发现，这就是 $(x_1+y_1,x_1-y_1),(x_2+y_2,x_2-y_2)$ 两点之间的切比雪夫距离.

所以将每一个点 (x,y) 转化为 $(x+y,x-y)$，新坐标系下的切比雪夫距离即为原坐标系下的曼哈顿距离.

同理，A,B 两点的切比雪夫距离为：

$$d(A,B)=\max\{|x_1-x_2|,|y_1-y_2|\}$$
$$=\max\left\{\left|\frac{x_1+y_1}{2}-\frac{x_2+y_2}{2}\right|+\left|\frac{x_1-y_1}{2}-\frac{x_2-y_2}{2}\right|\right\}$$

而这就是 $(\frac{x_1+y_1}{2},\frac{x_1-y_1}{2}),(\frac{x_2+y_2}{2},\frac{x_2-y_2}{2})$ 两点之间的曼哈顿距离.

所以将每一个点 (x,y) 转化为 $(\frac{x+y}{2},\frac{x-y}{2})$，新坐标系下的曼哈顿距离即为原坐标系下的切比雪夫距离.

结论

- 曼哈顿坐标系是通过切比雪夫坐标系旋转 45° 后，再缩小到原来的一半得到的.
- 将一个点 (x,y) 的坐标变为 $(x+y,x-y)$ 后，原坐标系中的曼哈顿距离等于新坐标系中的切比雪夫距离.
- 将一个点 (x,y) 的坐标变为 $(\frac{x+y}{2},\frac{x-y}{2})$ 后，原坐标系中的切比雪夫距离等于新坐标系中的曼哈顿距离.

碰到求切比雪夫距离或曼哈顿距离的题目时，我们往往可以相互转化来求解. 两种距离在不同的题目中有不同的优缺点，应该灵活运用.

例题

P4648「IOI2007」pairs 动物对数[4]（曼哈顿距离转切比雪夫距离）

P3964「TJOI2013」松鼠聚会[5]（切比雪夫距离转曼哈顿距离）

最后给出 P5098「USACO04OPEN」Cave Cows 3[6] 的第二种解法：

我们考虑将题目所求的曼哈顿距离转化为切比雪夫距离，即把每个点的坐标 (x,y) 变为 $(x+y,x-y)$.

所求的答案就变为 $\max\limits_{i,j\in n}\{\max\{|x_i-x_j|,|y_i-y_j|\}\}$.

现要使得横坐标之差和纵坐标之差最大，只需要预处理出 x,y 的最大值和最小值即可.

参考代码

- C++

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;

int main() {
  int n, x, y, a, b, minx = 0x7fffffff, maxx = 0, miny = 0x7fffffff, maxy = 0;
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &a, &b);
    x = a + b, y = a - b;
    minx = min(minx, x), maxx = max(maxx, x);
    miny = min(miny, y), maxy = max(maxy, y);
  }
  printf("%d\n", max(maxx - minx, maxy - miny));
  return 0;
}
```

- Python

```python
minx = 0x7FFFFFFF
maxx = 0
miny = 0x7FFFFFFF
maxy = 0
n = int(input())
for i in range(1, n + 1):
    a, b = map(lambda x: int(x), input().split())
    x = a + b
    y = a - b
    minx = min(minx, x)
    maxx = max(maxx, x)
    miny = min(miny, y)
    maxy = max(maxy, y)
print(max(maxx - minx, maxy - miny))
```

对比两份代码，我们又能够发现，两种不同的思路，写出来的代码却是完全等价的，是不是很神奇呢？当然，更高深的东西需要大家另行研究.

闵可夫斯基距离

我们定义 n 维空间中两点 $X(x_1,x_2,...,x_n)$，$Y(y_1,y_2,...,y_n)$ 之间的闵可夫斯基距离为：

$$D(X,Y)=\left(\sum_{i=1}^{n}|x_i-y_i|^p\right)^{\frac{1}{p}}.$$

特别的：

1. 当 $p=1$ 时，$D(X,Y)=\sum_{i=1}^{n}|x_i-y_i|$ 即为曼哈顿距离；
2. 当 $p=2$ 时，$D(X,Y)=\left(\sum_{i=1}^{n}(x_i-y_i)^2\right)^{1/2}$ 即为欧几里得距离；
3. 当 $p\to\infty$ 时，$D(X,Y)=\lim_{p\to\infty}\left(\sum_{i=1}^{n}|x_i-y_i|^p\right)^{1/p}=\max_{i=1}^{n}|x_i-y_i|$ 即为切比雪夫距离.

注意：当 $p\geq 1$ 时，闵可夫斯基距离才是度量，具体证明参见 Minkowski distance - Wikipedia[7].

参考资料与链接

1. 浅谈三种常见的距离算法[8]，感谢作者 xuxing 的授权.

[1] NOIP2017 提高组 奶酪

[2] P5098「USACO04OPEN」Cave Cows 3

[3] 切比雪夫距离 - 维基百科，自由的百科全书

[4] P4648「IOI2007」pairs 动物对数

[5] P3964「TJOI2013」松鼠聚会

[6] P5098「USACO04OPEN」Cave Cows 3

[7] Minkowski distance - Wikipedia

[8] 浅谈三种常见的距离算法

12.5 Pick 定理

Pick 定理

Pick 定理：给定顶点均为整点的简单多边形，皮克定理说明了其面积 A 和内部格点数目 i、边上格点数目 b 的关系：$A=i+\frac{b}{2}-1$.

具体证明：Pick's theorem[1]

它有以下推广：

- 取格点的组成图形的面积为一单位．在平行四边形格点，皮克定理依然成立．套用于任意三角形格点，皮克定理则是 $A = 2 \times i + b - 2$.
- 对于非简单的多边形 P，皮克定理 $A = i + \frac{b}{2} - \chi(P)$，其中 $\chi(P)$ 表示 P 的 **欧拉特征数**.
- 高维推广：Ehrhart 多项式
- 皮克定理和 **欧拉公式**（$V - E + F = 2$）等价.

一道例题 (POJ 1265[2])

题目大意

在直角坐标系中，一个机器人从任意点出发进行 n 次移动，每次向右移动 dx，向上移动 dy，最后会形成一个平面上的封闭简单多边形，求边上的点的数量，多边形内的点的数量，多边形面积.

题解

这道题目其实用了以下三个知识：

- 以整点为顶点的线段，如果边 dx 和 dy 都不为 0，经过的格点数是 $\gcd(dx, dy) + 1$，当然，如果要算一整个图形的，多加的点会被上一条边计算，也就不需要加了．那么一条边覆盖的点的个数为 $\gcd(dx, dy)$，其中，dx, dy 分别为线段横向占的点数和纵向占的点数．如果 dx 或 dy 为 0，则覆盖的点数为 dy **或** dx.
- Pick 定理：平面上以整点为顶点的简单多边形的面积 = 边上的点数/2 + 内部的点数 - 1.
- 任意一个多边形的面积等于按顺序求相邻两个点与原点组成的向量的叉积之和的一半（这个也可以通过顺时针定积分求得）.

参考代码

```cpp
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;
constexpr int MAXN = 110;

struct node {
  int x, y;
} p[MAXN];

// 求最大公约数
int gcd(int x, int y) { return y == 0 ? x : gcd(y, x % y); }

// 求区域
int area(int a, int b) { return p[a].x * p[b].y - p[a].y * p[b].x; }

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int t, ncase = 1;
  cin >> t;
  while (t--) {
    int n, dx, dy, x, y, num = 0, sum = 0;
    cin >> n;
    p[0].x = 0, p[0].y = 0;
    for (int i = 1; i <= n; i++) {
      cin >> x >> y;
      p[i].x = x + p[i - 1].x, p[i].y = y + p[i - 1].y;
```

```
28        dx = x, dy = y;
29        if (x < 0) dx = -x;
30        if (y < 0) dy = -y;
31        num += gcd(dx, dy);
32        sum += area(i - 1, i);
33      }
34      if (sum < 0) sum = -sum;
35      cout << fixed << setprecision(1);
36      cout << "Scenario #" << ncase++ << ":\n";
37      cout << ((sum - num + 2) >> 1) << ' ' << num << ' ' << sum * 0.5
<< "\n\n";
38    }
39    return 0;
40  }
```

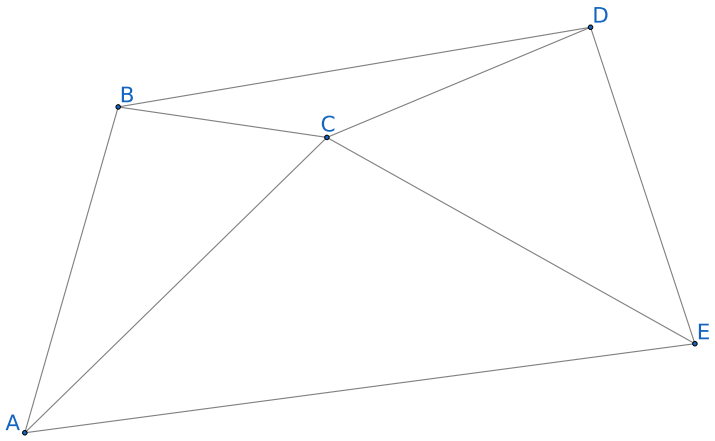
外部参考资料

[1] Pick's theorem  [2] POJ 1265 

12.6 三角剖分

Authors: xehoth

在几何中，三角剖分是指将平面对象细分为三角形，并且通过扩展将高维几何对象细分为单纯形．对于一个给定的点集，有很多种三角剖分，如：



本文介绍二维 Delaunay 三角剖分（简称 DT）及其分治构造算法．

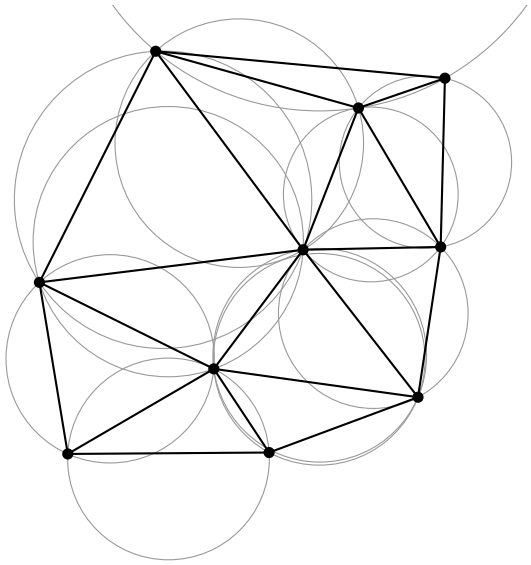
Delaunay 三角剖分

定义

在数学和计算几何中，对于给定的平面中的离散点集 P，其 Delaunay 三角剖分 DT(P) 满足：

1. 空圆性：DT(P) 是 **唯一** 的（任意四点不能共圆），在 DT(P) 中，**任意** 三角形的外接圆范围内不会有其它点存在．

2. 最大化最小角：在点集 P 可能形成的三角剖分中，DT(P) 所形成的三角形的最小角最大．从这个意义上讲，DT(P) 是 **最接近于规则化** 的三角剖分．具体的说是在两个相邻的三角形构成凸四边形的对角线，在相互交换后，两个内角的最小角不再增大．



性质

1. 最接近：以最接近的三点形成三角形，且各线段（三角形的边）皆不相交．
2. 唯一性：不论从区域何处开始构建，最终都将得到一致的结果（点集中任意四点不能共圆）．
3. 最优性：任意两个相邻三角形构成的凸四边形的对角线如果可以互换的话，那么两个三角形六个内角中最小角度不会变化．
4. 最规则：如果将三角剖分中的每个三角形的最小角进行升序排列，则 Delaunay 三角剖分的排列得到的数值最大．
5. 区域性：新增、删除、移动某一个顶点只会影响邻近的三角形．
6. 具有凸边形的外壳：三角剖分最外层的边界形成一个凸多边形的外壳．

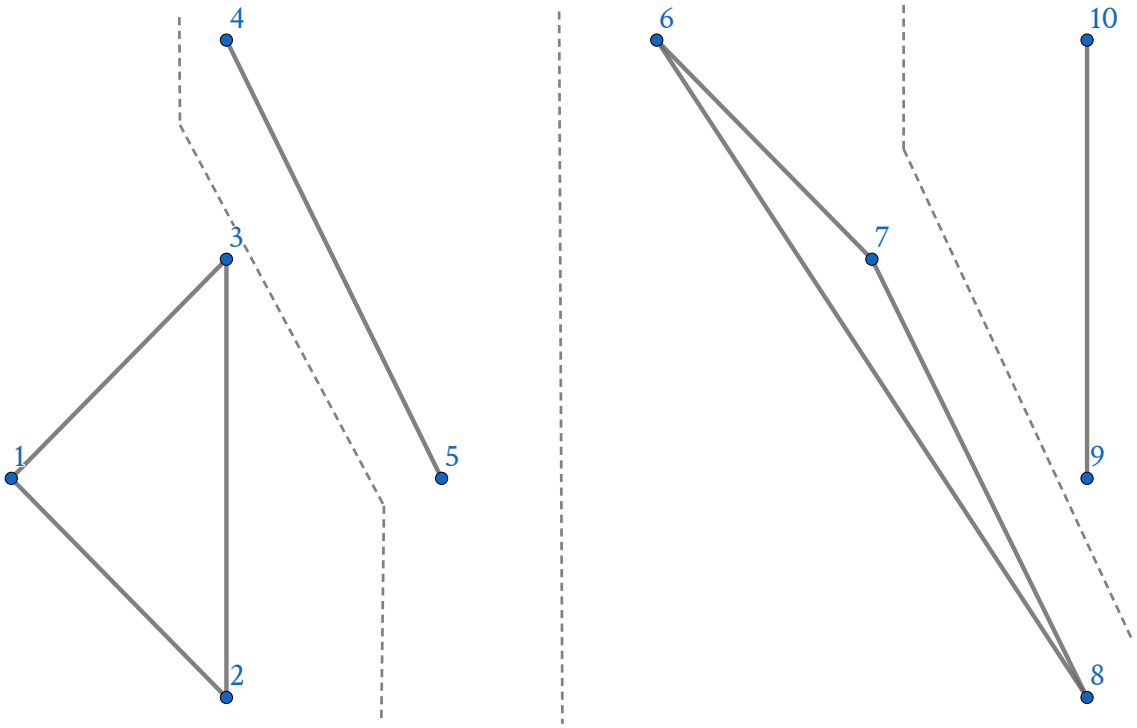
构造 DT 的分治算法

DT 有多种构造算法，下面介绍时间复杂度为 $O(n \log n)$ 的分治算法．

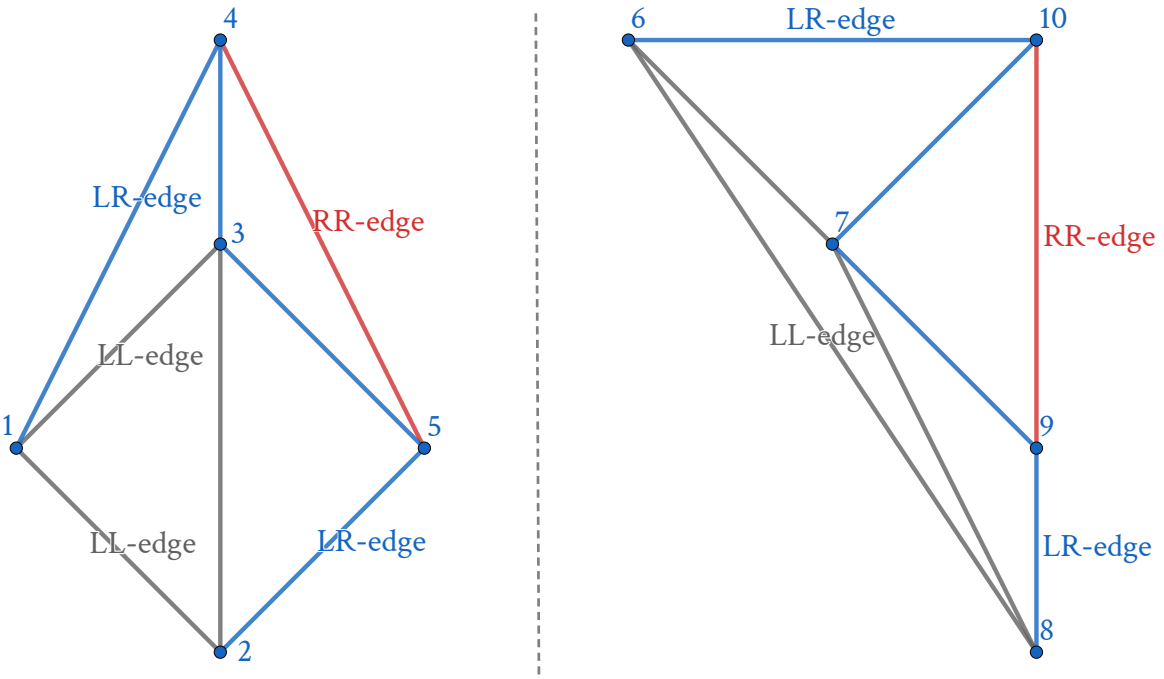
分治构造 DT 的第一步是将给定点集按照 x 坐标 **升序** 排列，x 相同时按照 y 坐标升序排列，并去除重合点．如下图是排好序的大小为 10 的点集．



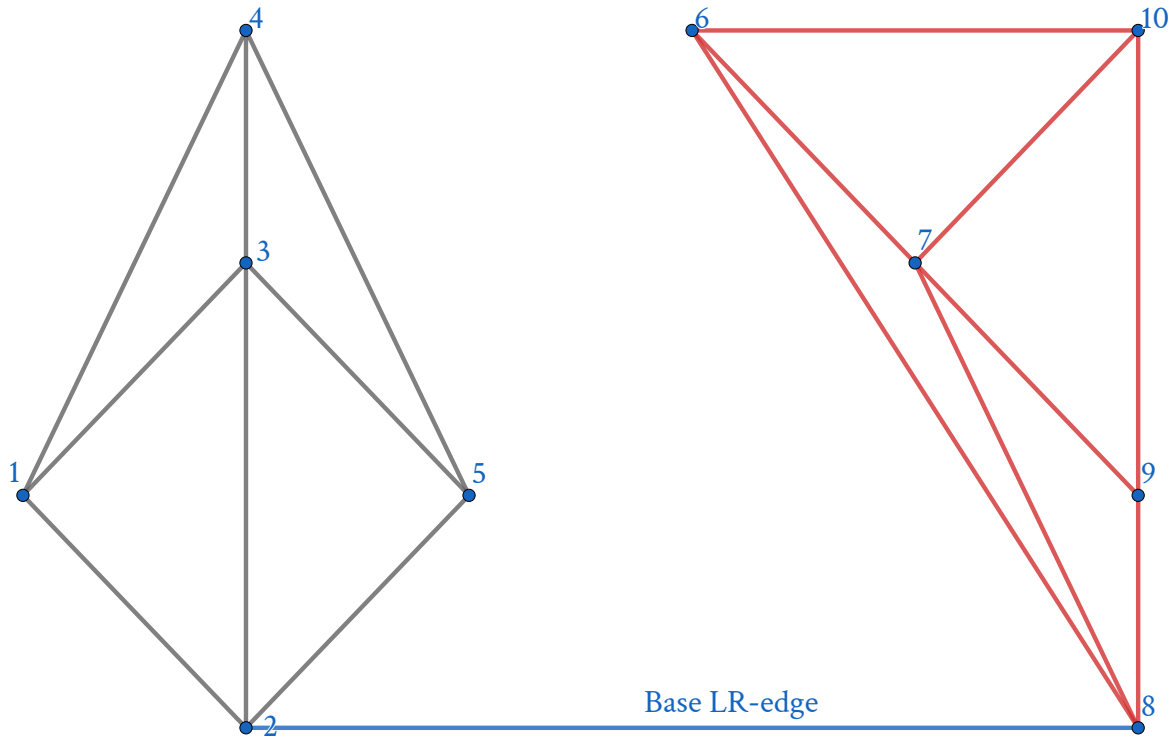
若点数不足 2，无需连边．否则，将有序点集不断从中间分成两部分，直到子点集大小为 2 或 3．其中两个点连成一条边，三个不共线的点连成一个三角形，三个共线的点只连接排序后相邻的两对点．



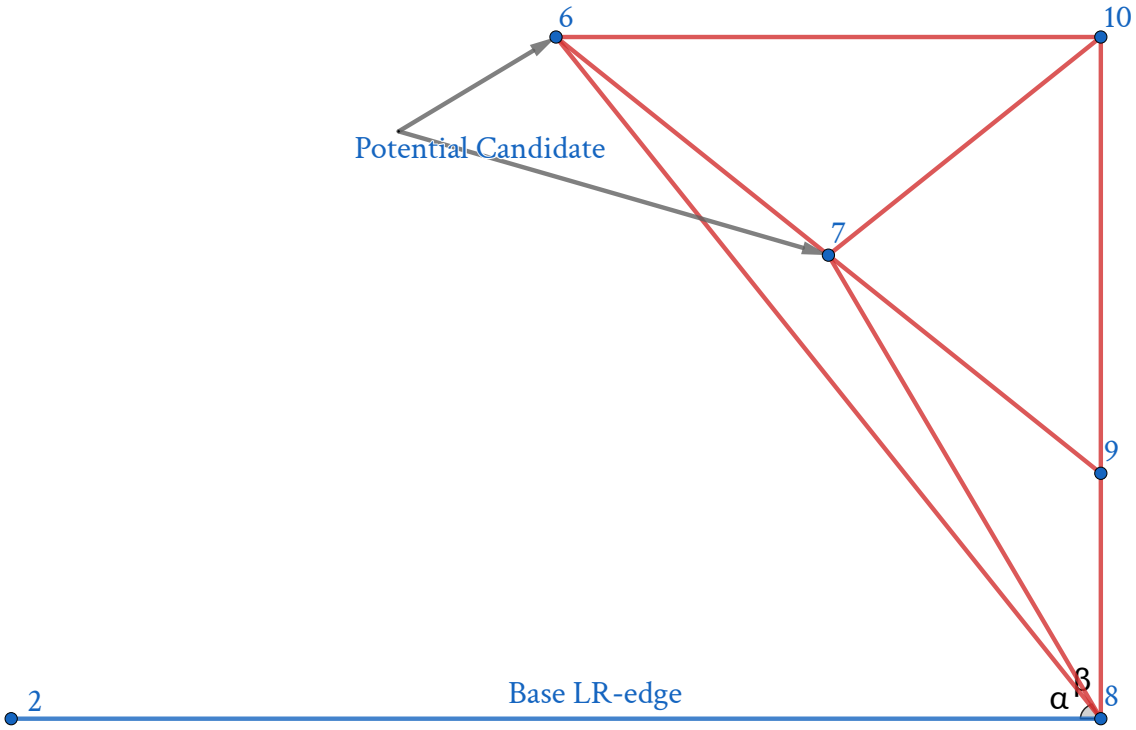
然后在分治回溯的过程中，依次合并左右子点集的剖分．合并后的边分为 LL-edge（左侧子点集内部的边）、RR-edge（右侧子点集内部的边）和 LR-edge（连接左右子点集的边），在下图中分别用灰色、红色和蓝色表示．为了维持 DT 性质，合并时 **可能** 需要删除部分 LL-edge 和 RR-edge，但 **不会** 增加这两类边．



合并左右两个剖分的第一步是找到两个凸包的下公切线，并插入对应的 base LR-edge．分治时返回左右凸包的边界边，从左侧凸包的最右端、右侧凸包的最左端开始，沿凸包边界移动，直到所有点都不在从左端点指向右端点的有向直线右侧．

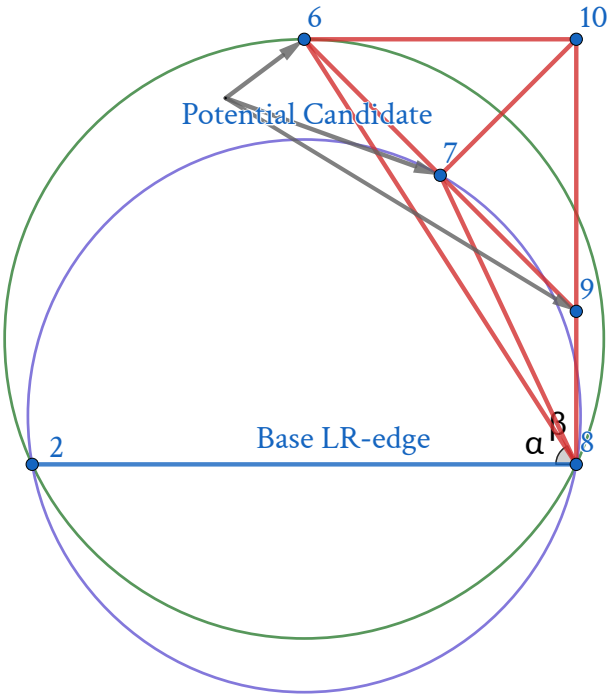


然后，我们需要确定下一条 **紧接在** base LR-edge 之上的 LR-edge．比如对于右侧点集，下一条 LR-edge 的可能端点（右端点）为与 base LR-edge 右端点相连的 RR-edge 的另一端点（6, 7, 9 号点），左端点即为 2 号点．



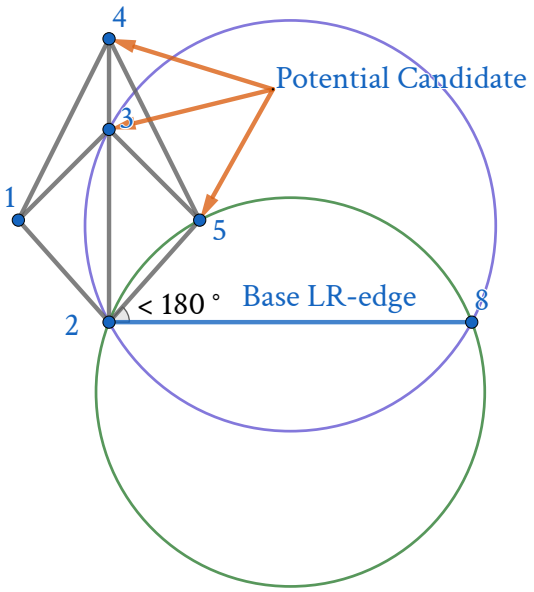
以右端点为例，从指向 base LR-edge 左端点的射线开始，按顺时针环绕顺序检查与右端点相连的 RR-edge：

1. 只有严格位于 base LR-edge 上方的端点才是有效候选点，即该点位于从 base 左端点指向右端点的有向直线的左侧．对应的顺时针转角须在 $(0^\circ, 180^\circ)$ 内．
2. 设当前候选点为 c，沿同一方向紧邻的下一个邻点为 d．若 d 严格位于 base LR-edge 两端点与 c 的外接圆内，则删除通向 c 的 RR-edge，并继续检查通向 d 的边．
3. 否则保留当前候选点，停止这一侧的检查．由于这一侧已经是 Delaunay 三角剖分，只需按环绕顺序比较相邻的候选边即可．



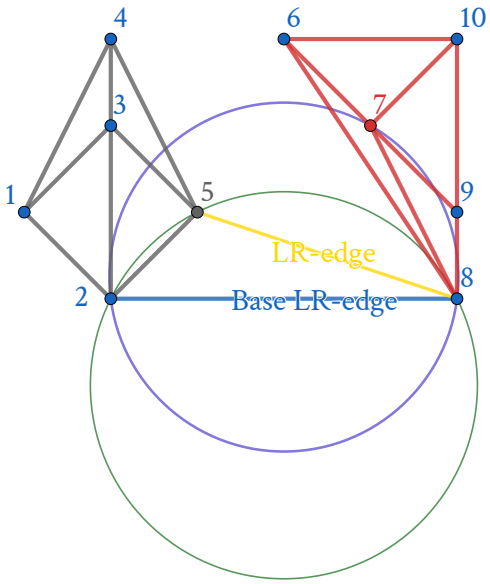
如上图，依次检查 6, 7, 9 号点．6 号点对应的绿色圆包含下一个邻点 7，因此删除通向 6 的 RR-edge；7 号点对应的紫色圆不包含下一个邻点 9，于是保留 7 作为右侧候选点．之后还要将它与左侧候选点比较，才能确定下一条 LR-edge．

对于左侧点集，从指向 base LR-edge 右端点的射线开始，按逆时针环绕顺序作镜像处理即可．

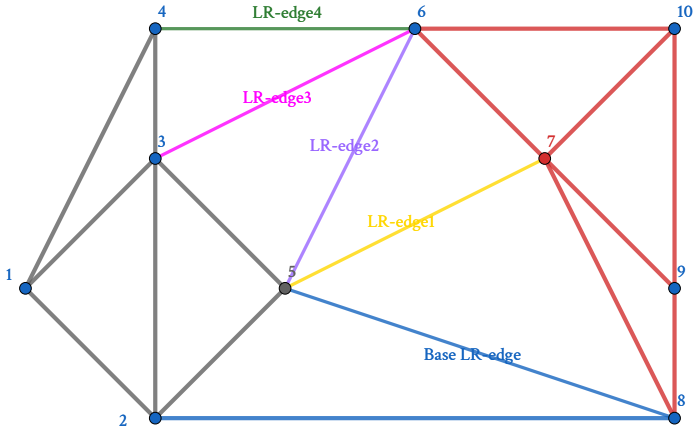


当左右两侧都没有有效候选点时，当前 base LR-edge 就是上公切线，合并完成．若只有一侧有有效候选点，就将它与 base LR-edge 的另一端点连接，得到新的 LR-edge．

当左右两侧都有有效候选点时，若右侧候选点严格位于左侧候选点与 base 两端点确定的外接圆内，则选择右侧候选点；否则选择左侧候选点．将选中的候选点与 base 的另一侧端点连接，得到新的 LR-edge．四点共圆时两者均可．



当这条 LR-edge 添加好后，将其作为 base LR-edge 重复以上步骤，继续添加下一条，直到合并完成．



实现

若只用无序邻接表存边，并在每次添加 LR-edge 时扫描两端点的所有邻边，则时间复杂度为 $O(n^2)$，因为一个端点可能连续形成多条 LR-edge，导致邻接表被反复扫描．

[1]Leonidas Guibas, Jorge Stolfi.Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams[1]. ACM Transactions on Graphics, 4(2), 1985, 74–123.

参考实现使用 Quad-edge[1]结构维护边的环绕顺序．每条无向边用四条有向边记录，其中两条表示原图的两个方向，另外两条表示对偶图的两个方向．同一组记录连续存放，所以只需维护每条有向边的起点和同起点的下一条逆时针边，就能在 $O(1)$ 时间内实现以下操作：

操作	含义
`rev(e)`	反向边
`onext(e)`	起点相同的下一条逆时针边
`oprev(e)`	起点相同的上一条逆时针边
`lnext(e)`	沿左侧面的边界前进一条边
`onext(rev(e))`	沿右侧面的边界后退一条边

`splice(a, b)` 同时修改原图和对偶图的环绕关系，用来拼接或拆开两条边所在的环．`connect(a, b)` 在同一个面内连接 `a` 的终点和 `b` 的起点．删除边时，将它的两个方向分别从对应的环中移除．这些拓扑操作只需修改常数个记录；参考实现使用动态数组分配和回收边，均摊耗时为 $O(1)$．

代码中 `base` 的方向是从右侧点集指向左侧点集，因此图示中 base LR-edge「上方」的点位于有向边 `base` 的右侧．左侧候选边为 `onext(rev(base))`，右侧候选边为 `oprev(base)`，删除候选边后，只需沿这一侧的环绕顺序继续前进．

实现

```cpp
#include <algorithm>
#include <cmath>
#include <limits>
#include <utility>
#include <vector>

using data_t = double;

struct Point {
  data_t x, y;
  int id;
};

class Delaunay {
  // 每条无向边由四个槽位组成
  // 偶数槽位表示原图中的两个方向，奇数槽位表示对偶边
  // 所有连接均存下标，避免迭代器失效
  struct Edge {
    int origin, next;
  };

  std::vector<Point> p;
  std::vector<Edge> edges;
  std::vector<int> free_edges;

  // 使用相对容差近似判断符号
  static int sign(data_t value, data_t scale) {
    const data_t tolerance =
        16 * std::numeric_limits<data_t>::epsilon() * scale;
    return (value > tolerance) - (value < -tolerance);
  }
```

```
32
33  static int cross(const Point& a, const Point& b, const Point& c) {
34    data_t u = (b.x - a.x) * (c.y - a.y);
35    data_t v = (b.y - a.y) * (c.x - a.x);
36    return sign(u - v, std::abs(u) + std::abs(v));
37  }
38
39  // a、b、c 逆时针排列时，判断 d 是否在外接圆内
40  static bool in_circle(const Point& a, const Point& b, const Point& c,
41                        const Point& d) {
42    data_t ax = a.x - d.x, ay = a.y - d.y;
43    data_t bx = b.x - d.x, by = b.y - d.y;
44    data_t cx = c.x - d.x, cy = c.y - d.y;
45    data_t a2 = ax * ax + ay * ay, b2 = bx * bx + by * by,
46           c2 = cx * cx + cy * cy;
47    data_t bc1 = bx * cy, bc2 = by * cx;
48    data_t ca1 = cx * ay, ca2 = cy * ax;
49    data_t ab1 = ax * by, ab2 = ay * bx;
50    data_t det = a2 * (bc1 - bc2) + b2 * (ca1 - ca2) + c2 * (ab1 - ab2);
51    // 在相减前累加绝对值，避免抵消后低估舍入误差
52    data_t scale = a2 * (std::abs(bc1) + std::abs(bc2)) +
53                   b2 * (std::abs(ca1) + std::abs(ca2)) +
54                   c2 * (std::abs(ab1) + std::abs(ab2));
55    return sign(det, scale) > 0;
56  }
57
58  static int rot(int e) { return (e & ~3) | ((e + 1) & 3); }
59
60  static int rev(int e) { return e ^ 2; }
61
62  int org(int e) const { return edges[e].origin; }
63
64  int dest(int e) const { return org(rev(e)); }
65
66  // 同一起点的边按逆时针方向组成循环链表
67  int onext(int e) const { return edges[e].next; }
68
69  int oprev(int e) const { return rot(onext(rot(e))); }
70
71  // 沿 e 左侧面的边界前进一条边
72  int lnext(int e) const { return rot(onext(rev(rot(e)))); }
73
74  bool left_of(int v, int e) const {
75    return cross(p[org(e)], p[dest(e)], p[v]) > 0;
76  }
77
78  bool right_of(int v, int e) const {
79    return cross(p[org(e)], p[dest(e)], p[v]) < 0;
80  }
81
82  int make_edge(int u, int v) {
83    int e;
```

```
    if (free_edges.empty()) {
      e = (int)edges.size();
      edges.resize(edges.size() + 4);
    } else {
      e = free_edges.back();
      free_edges.pop_back();
    }
    edges[e] = {u, e};
    edges[e + 1] = {-1, e + 3};
    edges[e + 2] = {v, e + 2};
    edges[e + 3] = {-1, e + 1};
    return e;
  }

  // 交换两条边的后继，同时更新对偶图的连接
  void splice(int a, int b) {
    int alpha = rot(onext(a)), beta = rot(onext(b));
    std::swap(edges[a].next, edges[b].next);
    std::swap(edges[alpha].next, edges[beta].next);
  }

  void delete_edge(int e) {
    splice(e, oprev(e));
    splice(rev(e), oprev(rev(e)));
    e &= ~3;
    edges[e].origin = edges[e + 2].origin = -1;
    free_edges.push_back(e);
  }

  // 加入从 a 的终点到 b 的起点的边
  int connect(int a, int b) {
    int e = make_edge(dest(a), org(b));
    splice(e, lnext(a));
    splice(rev(e), b);
    return e;
  }

  // 返回起于最左、最右顶点的凸包边，分别使外部面位于右侧、左侧
  // 区间使用 [l, r)，递归只处理至少两个点的情况
  std::pair<int, int> divide(int l, int r) {
    if (r - l == 2) {
      int a = make_edge(l, l + 1);
      return {a, rev(a)};
    }
    if (r - l == 3) {
      int a = make_edge(l, l + 1), b = make_edge(l + 1, l + 2);
      splice(rev(a), b);
      int turn = cross(p[l], p[l + 1], p[l + 2]);
      if (turn == 0) return {a, rev(b)};  // 共线时只保留相邻点连边
      int c = connect(b, a);
      if (turn > 0) return {a, rev(b)};
      return {rev(c), c};
```

```
    }

    int m = l + (r - l) / 2;
    auto left = divide(l, m), right = divide(m, r);
    int ldo = left.first, ldi = left.second;
    int rdi = right.first, rdo = right.second;
    // 从递归返回的凸包边出发，沿凸包寻找下公切线
    while (true) {
      if (left_of(org(rdi), ldi)) {
        ldi = lnext(ldi);
      } else if (right_of(org(ldi), rdi)) {
        rdi = onext(rev(rdi));
      } else {
        break;
      }
    }
    int base = connect(rev(rdi), ldi);  // base 从右侧指向左侧
    if (org(ldi) == org(ldo)) ldo = rev(base);
    if (org(rdi) == org(rdo)) rdo = base;

    while (true) {
      // 候选边来自有序的环形邻接表，只需访问当前边的前驱或后继
      int lcand = onext(rev(base));
      if (right_of(dest(lcand), base)) {
        while (in_circle(p[dest(base)], p[org(base)], p[dest(lcand)],
                         p[dest(onext(lcand))])) {
          int next = onext(lcand);
          delete_edge(lcand);
          lcand = next;
        }
      }
      int rcand = oprev(base);
      if (right_of(dest(rcand), base)) {
        while (in_circle(p[dest(base)], p[org(base)], p[dest(rcand)],
                         p[dest(oprev(rcand))])) {
          int prev = oprev(rcand);
          delete_edge(rcand);
          rcand = prev;
        }
      }
      bool lvalid = right_of(dest(lcand), base);
      bool rvalid = right_of(dest(rcand), base);
      if (!lvalid && !rvalid) break;  // 已到达上公切线
      if (!lvalid || (rvalid && in_circle(p[dest(lcand)], p[org(lcand)],
                                          p[org(rcand)], p[dest(rcand)]))) {
        base = connect(rcand, rev(base));
      } else {
        base = connect(rev(base), rev(lcand));
      }
    }
    return {ldo, rdo};
  }
```

```cpp

 public:
  // 要求点互不重合，id 互不相同
  void init(std::vector<Point> points) {
    p = std::move(points);
    edges.clear();
    free_edges.clear();
    std::sort(p.begin(), p.end(), [](const Point& a, const Point& b) {
      return a.x < b.x || (a.x == b.x && a.y < b.y);
    });
    if (p.size() >= 2) divide(0, (int)p.size());
  }

  // 每条无向边只返回一次，端点编号为输入的原始 id
  std::vector<std::pair<int, int>> getEdge() const {
    std::vector<std::pair<int, int>> result;
    for (int e = 0; e < (int)edges.size(); e += 4) {
      if (org(e) != -1) result.emplace_back(p[org(e)].id, p[dest(e)].id);
    }
    return result;
  }
};

```

复杂度

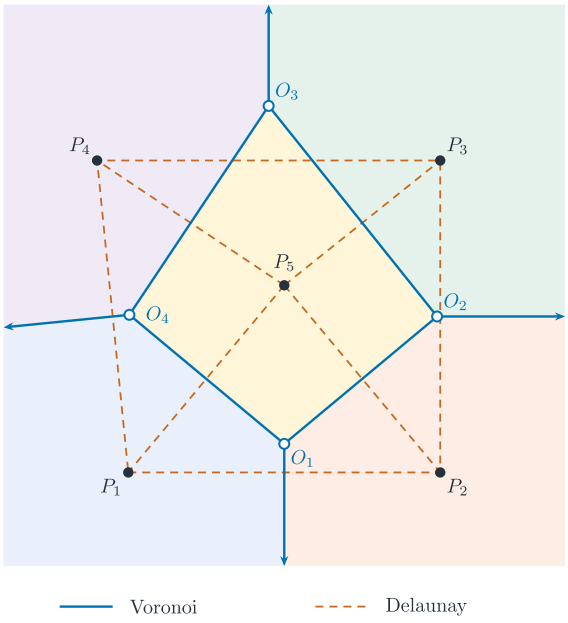
设一次合并涉及 k 个点．寻找下公切线时，每次移动都沿某一侧的凸包边界前进，总计 $O(k)$ 次．选候选点时，每次继续向后检查都伴随一条 LL-edge 或 RR-edge 的删除，而两侧子剖分总共只有 $O(k)$ 条边．每次合并主循环除这些删除操作外只做常数次判断，并添加一条 LR-edge；新添加的 LR-edge 在本次合并中不再删除，数目也是 $O(k)$．因此一次合并的总时间为 $O(k)$．

初始排序耗时 $O(n \log n)$，递归满足 $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$，总时间复杂度为 $O(n \log n)$．任一时刻保留的边数为 $O(n)$，代码还会回收被删除边的存储位置，避免保存所有历史边，因此空间复杂度为 $O(n)$．

Voronoi 图

给定平面上 $n \geq 1$ 个互不重合的种子点，每个种子点对应的 Voronoi 区域由到该点的距离不大于到其他任一种子点距离的所有点组成. 这些区域是可能无界的凸区域, 其内部互不相交, 并共同覆盖整个平面; 相邻区域的公共边界位于相应两种子点连线的垂直平分线上.

对于不全共线且任意四点不共圆的点集，Voronoi 图与 Delaunay 三角剖分互为对偶：每个三角面对应其外心，每条内部边对应连接两侧三角形外心的线段，每条凸包边对应从所在三角形外心出发、垂直于该边并朝凸包外侧延伸的射线．若存在四点共圆，构造后需合并重合的外心并去除零长对偶边．全部点共线时，Voronoi 边为排序后相邻点连线的垂直平分直线；只有一个点时，其区域为整个平面．



上图中，实心点 P_i 为种子点，空心点 O_i 为三角形外心；蓝色实线构成 Voronoi 图，橙色虚线构成 Delaunay 三角剖分．背景色区分各个 Voronoi 区域，箭头表示无界边；图中仅展示有限视窗内的部分．

构造 DT 后，利用已有的边环绕顺序枚举面和边，可在 $O(n)$ 时间内完成上述转换，因此构造 Voronoi 图的总时间复杂度为 $O(n \log n)$．

题目

Luogu P6362 平面欧几里得最小生成树[2] 三角剖分经典应用

SGU 383 Caravans[3] 三角剖分 + 倍增

ContestHunter. 无尽的毁灭[4] 三角剖分求对偶图建 Voronoi 图

Codeforces Gym 103485M. Constellation collection[5] 三角剖分之后建图进行 Floodfill

参考资料与拓展阅读

1. Wikipedia - Triangulation (geometry)[6]
2. Wikipedia - Delaunay triangulation[7]
3. Samuel Peterson - Computing Constrained Delaunay Triangulations in 2-D (1997-98)[8]

[1] Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams

[2] Luogu P6362 平面欧几里得最小生成树

[3] SGU 383 Caravans

[4] ContestHunter. 无尽的毁灭

[5] Codeforces Gym 103485M. Constellation collection

[6] Wikipedia - Triangulation (geometry)

[7] Wikipedia - Delaunay triangulation

[8] Samuel Peterson - Computing Constrained Delaunay Triangulations in 2-D (1997-98)

12.7 凸包

二维凸包

定义

凸多边形

凸多边形是指所有内角大小都在 $[0,\pi]$ 范围内的 **简单多边形**.

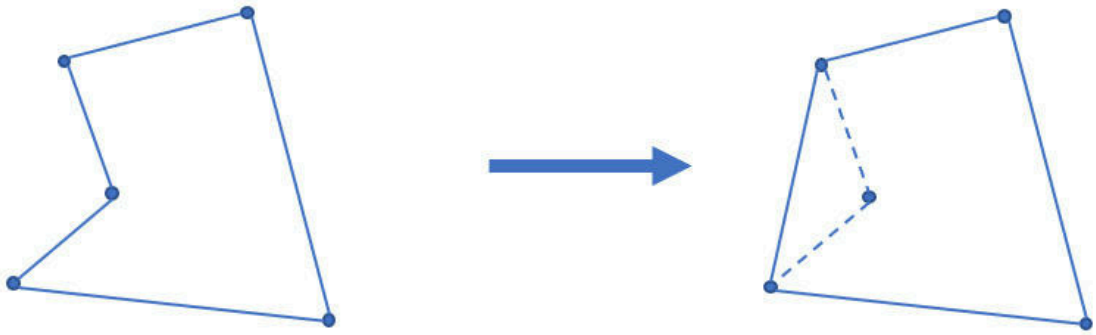
凸包

在平面上能包含所有给定点的最小凸多边形叫做凸包.

其定义为：对于给定集合 X，所有包含 X 的凸集的交集 S 被称为 X 的 **凸包**.

实际上可以理解为用一个橡皮筋包含住所有给定点的形态.

凸包用最小的周长围住了给定的所有点．如果一个凹多边形围住了所有的点，它的周长一定不是最小，如下图．根据三角不等式，凸多边形在周长上一定是最优的.



Andrew 算法求凸包

常用的求法有 Graham 扫描法和 Andrew 算法，这里主要介绍 Andrew 算法.

性质

该算法的时间复杂度为 $O(n\log n)$，其中 n 为待求凸包点集的大小，复杂度的瓶颈在于对所有点坐标的双关键字排序.

过程

首先把所有点以横坐标为第一关键字，纵坐标为第二关键字排序.

显然排序后最小的元素和最大的元素一定在凸包上. 而且因为是凸多边形, 我们如果从一个点出发逆时针走，轨迹总是「左拐」的，一旦出现右拐，就说明这一段不在凸包上．因此我们可以用一个单调栈来维护上下凸壳.

因为从左向右看，上下凸壳所旋转的方向不同，为了让单调栈起作用，我们首先 **升序枚举** 求出下凸壳，然后 **降序** 求出上凸壳.

求凸壳时，一旦发现即将进栈的点（P）和栈顶的两个点（S_1, S_2，其中 S_1 为栈顶）行进的方向向右旋转，即叉积小于 0：$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$，则弹出栈顶，回到上一步，继续检测，直到 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \geq 0$ 或者栈内仅剩一个元素为止.

通常情况下不需要保留位于凸包边上的点，因此上面一段中 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$ 这个条件中的「$<$」可以视情况改为 $\leq$，同时后面一个条件应改为 $>$.

实现

代码实现

- C++

```cpp
// stk[] 是整型，存的是下标
// p[] 存储向量或点
tp = 0;                           // 初始化栈
std::sort(p + 1, p + 1 + n);  // 对点进行排序
stk[++tp] = 1;
// 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
for (int i = 2; i <= n; ++i) {
  while (tp >= 2  // 下一行 * 操作符被重载为叉积
         && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
    used[stk[tp--]] = 0;
  used[i] = 1;  // used 表示在凸壳上
  stk[++tp] = i;
}
int tmp = tp;  // tmp 表示下凸壳大小
for (int i = n - 1; i > 0; --i)
  if (!used[i]) {
    // ↓求上凸壳时不影响下凸壳
    while (tp > tmp && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] -
p[stk[tp]]) <= 0)
      used[stk[tp--]] = 0;
    used[i] = 1;
    stk[++tp] = i;
  }
for (int i = 1; i <= tp; ++i)  // 复制到新数组中去
  h[i] = p[stk[i]];
int ans = tp - 1;
```

- Python

```python
stk = []  # 是整型，存的是下标
p = []  # 存储向量或点
tp = 0  # 初始化栈
p.sort()  # 对点进行排序
tp = tp + 1
stk[tp] = 1
# 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
```

```
for i in range(2, n + 1):
    while tp >= 2 and (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0:
        # 下一行 * 操作符被重载为叉积
        used[stk[tp]] = 0
        tp = tp - 1
    used[i] = 1  # used 表示在凸壳上
    tp = tp + 1
    stk[tp] = i
tmp = tp  # tmp 表示下凸壳大小
for i in range(n - 1, 0, -1):
    if used[i] == False:
        #       ↓求上凸壳时不影响下凸壳
        while tp > tmp and (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0:
            used[stk[tp]] = 0
            tp = tp - 1
        used[i] = 1
        tp = tp + 1
        stk[tp] = i
for i in range(1, tp + 1):
    h[i] = p[stk[i]]
ans = tp - 1
```

根据上面的代码，最后凸包上有 ans 个元素（额外存储了 1 号点，因此 h 数组中有 $ans+1$ 个元素），并且按逆时针方向排序．周长就是

$$\sum_{i=1}^{ans} \left|\overrightarrow{h_i h_{i+1}}\right|$$

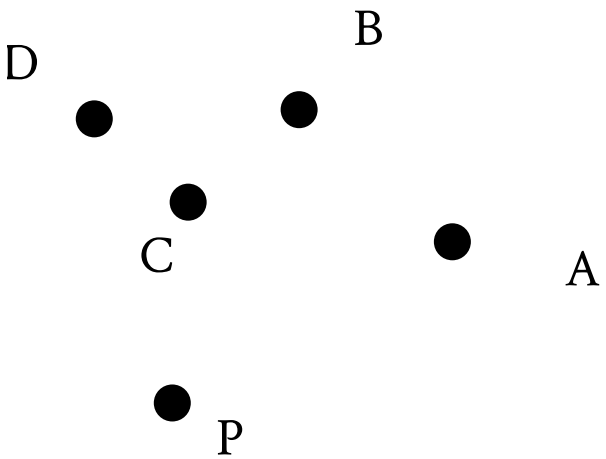
Graham 扫描法

性质

与 Andrew 算法相同，Graham 扫描法的时间复杂度为 $O(n \log n)$，复杂度瓶颈也在于对所有点排序．

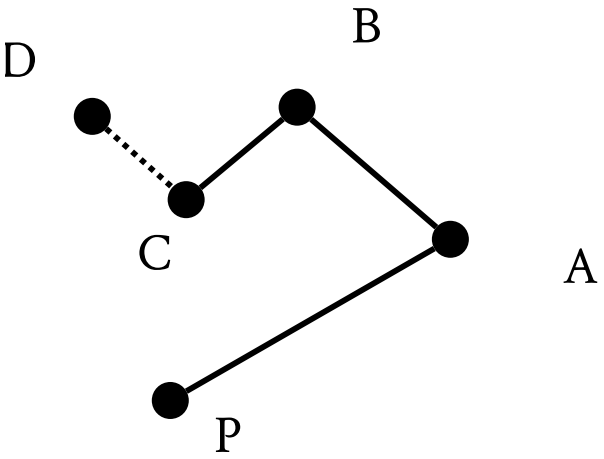
过程

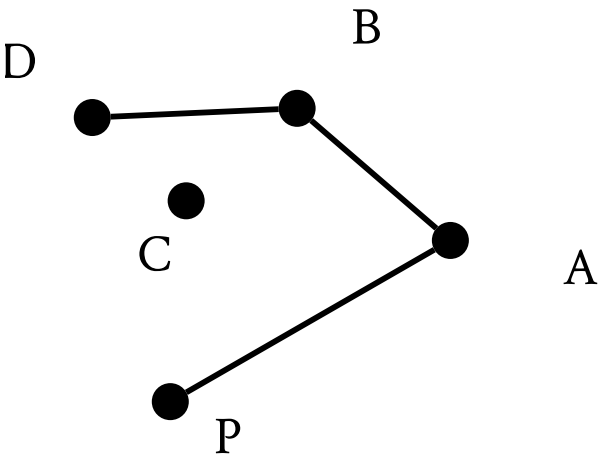
首先找到所有点中，纵坐标最小的一个点 P．根据凸包的定义我们知道，这个点一定在凸包上．然后将所有的点以相对于点 P 的极角大小为关键字进行排序．



和 Andrew 算法类似地，我们考虑从点 P 出发，在凸包上逆时针走，那么我们经过的所有节点一定都是「左拐」的．形式化地说，对于凸包逆时针方向上任意连续经过的三个点 P_1, P_2, P_3，一定满足 $\overrightarrow{P_1P_2} \times \overrightarrow{P_2P_3} \geq 0$.

新建一个栈用于存储凸包的信息，先将 P 压入栈中，然后按照极角序依次尝试加入每一个点．如果进栈的点 P_0 和栈顶的两个点 P_1, P_2（其中 P_1 为栈顶）行进的方向「右拐」了，那么就弹出栈顶的 P_1，不断重复上述过程直至进栈的点与栈顶的两个点满足条件，或者栈中仅剩下一个元素，再将 P_0 压入栈中.





代码实现

```
struct Point {
  double x, y, ang;

  Point operator-(const Point& p) const { return {x - p.x, y - p.y, 0}; }
} p[MAXN];

double dis(Point p1, Point p2) {
  return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

bool cmp(Point p1, Point p2) {
  if (p1.ang == p2.ang) {
    return dis(p1, p[1]) < dis(p2, p[1]);
```

```
  }
  return p1.ang < p2.ang;
}

double cross(Point p1, Point p2) { return p1.x * p2.y - p1.y * p2.x; }

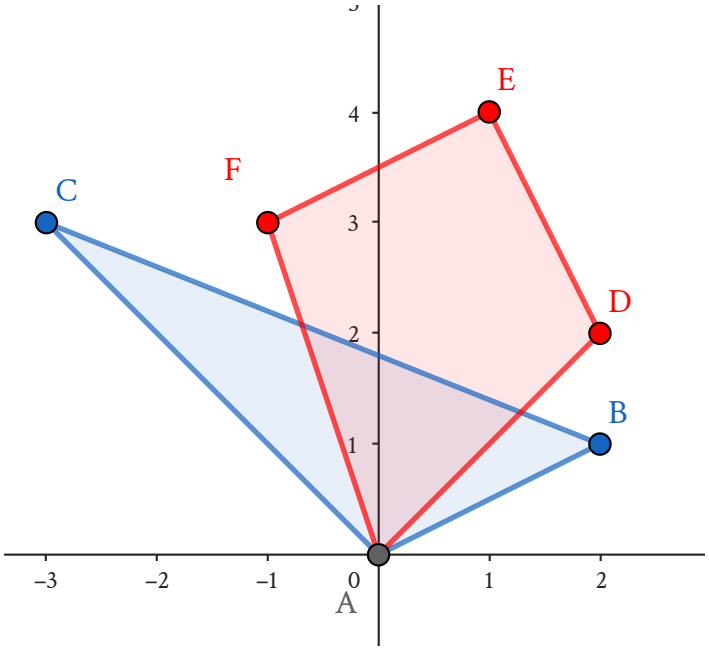
int main() {
  for (int i = 2; i <= n; ++i) {
    if (p[i].y < p[1].y || (p[i].y == p[1].y && p[i].x < p[1].x)) {
      std::swap(p[1], p[i]);
    }
  }
  for (int i = 2; i <= n; ++i) {
    p[i].ang = atan2(p[i].y - p[1].y, p[i].x - p[1].x);
  }
  std::sort(p + 2, p + n + 1, cmp);
  sta[++top] = 1;
  for (int i = 2; i <= n; ++i) {
    while (top >= 2 &&
           cross(p[sta[top]] - p[sta[top - 1]], p[i] - p[sta[top]]) < 0) {
      top--;
    }
    sta[++top] = i;
  }
  return 0;
}
```

闵可夫斯基和

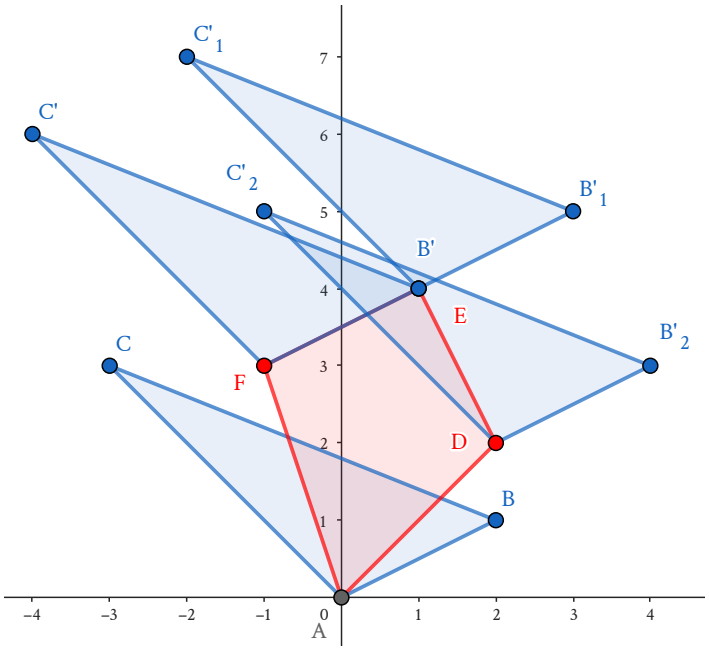
定义

点集 P 和点集 Q 的闵可夫斯基和 $P+Q$ 定义为 $P+Q=\{a+b \mid a\in P, b\in Q\}$，即把点集 Q 中的每个点看做一个向量，将点集 P 中每个点沿这些向量平移，最终得到的结果的集合就是点集 $P+Q$. 此处仅讨论 **凸包** 的闵可夫斯基和.

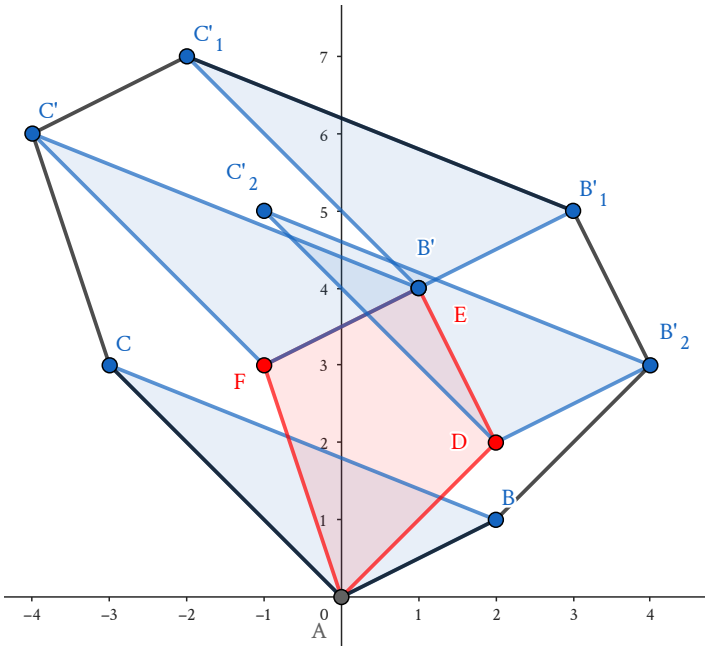
例如：对于点集 $P=\{(0,0),(-3,3),(2,1)\}$ 和 点集 $Q=\{(0,0),(-1,3),(1,4),(2,2)\}$,



将 P 沿 Q 的每个向量平移:



不难发现新图形也是一个 **凸包**:



性质

1. 若点集 P，Q 为凸集，则其闵可夫斯基和 $P+Q$ 也是凸集.

证明

设 $e,f\in P+Q$，有 $a,b\in P$，$c,d\in Q$ 且 $e=a+c,f=b+d$，则对任意 $t\in[0,1]$ 均有：

$$\begin{aligned} te+(1-t)f&=t(a+c)+(1-t)(b+d)\\ &=(ta+(1-t)b)+(tc+(1-t)d)\\ &\in P+Q. \end{aligned}$$

证毕.

2. 若点集 P，Q 为凸集，则其闵可夫斯基和 $P+Q$ 的边集是由凸集 P，Q 的边按极角排序后连接的结果.

证明

不妨假设凸集 P 中任意一条边的斜率与 Q 中任意一条边的斜率均不相同. 将坐标系进行旋转，使得 P 上的一条边 XY 与 x 轴平行且在最下方.

设此时 Q 中最低的点 U，$P+Q$ 的 **最低** 且 **靠左** 的点 A.

可知 $\vec{A}=\vec{X}+\vec{U}$，所以 A 必然在 $P+Q$ 的边界上.

同理，$P+Q$ 中 **最低** 且 **靠右** 的点 B 有 $\vec{B}=\vec{Y}+\vec{U}$，也必然在 $P+Q$ 的边界上.

因此，有 $\overrightarrow{AB}=\overrightarrow{XY}+\vec{U}$.

若按顺序进行旋转，则结果连续的构成了 $P+Q$ 中的每条边.

证毕.

实现

我们可以根据性质 2，将凸集 P,Q 极角排序，得到它们在 $P+Q$ 上的出现顺序，把 P_1+Q_1 看做 $P+Q$ 的起点，然后用类似 **归并** 的做法依次放边即可.

时间复杂度：$O(n+m)$

实现

```cpp
template <class T>
struct Point {
  T x, y;

  Point(T x = 0, T y = 0) : x(x), y(y) {}

  friend Point operator+(const Point &a, const Point &b) {
    return {a.x + b.x, a.y + b.y};
  }

  friend Point operator-(const Point &a, const Point &b) {
    return {a.x - b.x, a.y - b.y};
  }

  // 点乘
  friend T operator*(const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y;
  }

  // 叉乘
  friend T operator^(const Point &a, const Point &b) {
    return a.x * b.y - a.y * b.x;
  }
};

template <class T>
vector<Point<T>> minkowski_sum(vector<Point<T>> a, vector<Point<T>> b) {
  vector<Point<T>> c{a[0] + b[0]};
  for (usz i = 0; i + 1 < a.size(); ++i) a[i] = a[i + 1] - a[i];
  for (usz i = 0; i + 1 < b.size(); ++i) b[i] = b[i + 1] - b[i];
  a.pop_back(), b.pop_back();
  c.resize(a.size() + b.size() + 1);
  merge(a.begin(), a.end(), b.begin(), b.end(), c.begin() + 1,
        [](const Point<T> &a, const Point<T> &b) { return (a ^ b) < 0; });
  for (usz i = 1; i < c.size(); ++i) c[i] = c[i] + c[i - 1];
  return c;
}
```

例题

例题 [JSOI2018] 战争[1]

有两个凸包 P, Q，平移 q 次 Q，问每次移动后是否有交点．$1 \le n, m \le 10^5, 1 \le q \le 10^5$.

实现

```cpp
#include <algorithm>
#include <cassert>
#include <cstdint>
```

```cpp
#include <iostream>
#include <vector>
using namespace std;
using i64 = int64_t;
using isz = ptrdiff_t;
using usz = size_t;

template <class T>
struct Point {
  T x, y;

  Point(T x = 0, T y = 0) : x(x), y(y) {}

  friend Point operator+(const Point &a, const Point &b) {
    return {a.x + b.x, a.y + b.y};
  }

  friend Point operator-(const Point &a, const Point &b) {
    return {a.x - b.x, a.y - b.y};
  }

  // 点乘
  friend T operator*(const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y;
  }

  // 叉乘
  friend T operator^(const Point &a, const Point &b) {
    return a.x * b.y - a.y * b.x;
  }

  friend istream &operator>>(istream &is, Point &p) { return is >> p.x >> p.y; }
};

template <class T>
vector<Point<T>> convex_hull(vector<Point<T>> p) {
  assert(!p.empty());
  sort(p.begin(), p.end(),
       [](const Point<i64> &a, const Point<i64> &b) { return a.x < b.x; });
  vector<Point<T>> u{p[0]}, d{p.back()};
  for (usz i = 1; i < p.size(); ++i) {
    while (u.size() >= 2 &&
           ((u.back() - u[u.size() - 2]) ^ (p[i] - u.back())) > 0)
      u.pop_back();
    u.push_back(p[i]);
  }
  for (usz i = p.size() - 2; (isz)i >= 0; --i) {
    while (d.size() >= 2 &&
           ((d.back() - d[d.size() - 2]) ^ (p[i] - d.back())) > 0)
      d.pop_back();
    d.push_back(p[i]);
```

```cpp
  }
  u.insert(u.end(), d.begin() + 1, d.end());
  return u;
}

template <class T>
vector<Point<T>> minkowski_sum(vector<Point<T>> a, vector<Point<T>> b) {
  vector<Point<T>> c{a[0] + b[0]};
  for (usz i = 0; i + 1 < a.size(); ++i) a[i] = a[i + 1] - a[i];
  for (usz i = 0; i + 1 < b.size(); ++i) b[i] = b[i + 1] - b[i];
  a.pop_back(), b.pop_back();
  c.resize(a.size() + b.size() + 1);
  merge(a.begin(), a.end(), b.begin(), b.end(), c.begin() + 1,
        [](const Point<T> &a, const Point<T> &b) { return (a ^ b) < 0; });
  for (usz i = 1; i < c.size(); ++i) c[i] = c[i] + c[i - 1];
  return c;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  uint32_t n, m, q;
  vector<Point<i64>> a, b;
  cin >> n >> m >> q;
  a.resize(n), b.resize(m);
  for (auto &p : a) cin >> p;
  for (auto &p : b) cin >> p, p = 0 - p;
  a = convex_hull(a), b = convex_hull(b);
  a = minkowski_sum(a, b);
  a.pop_back();
  for (usz i = 1; i < a.size(); ++i) a[i] = a[i] - a[0];
  while (q--) {
    Point<i64> v;
    cin >> v;
    v = v - a[0];
    if (v.x < 0) {
      cout << "0\n";
      continue;
    }
    auto it = upper_bound(
        a.begin() + 1, a.end(), v,
        [](const Point<i64> &a, const Point<i64> &b) { return (a ^ b)
< 0; });
    if (it == a.begin() + 1 || it == a.end()) {
      cout << "0\n";
      continue;
    }
    i64 s0 = *it ^ *prev(it), s1 = v ^ *prev(it), s2 = *it ^ v;
    cout << (s1 >= 0 && s2 >= 0 && s1 + s2 <= s0) << '\n';
  }
  return 0;
}

```

三维凸包

基础知识

> 圆的反演：反演中心为 O，反演半径为 R，若经过 O 的直线经过 P,P'，且 $OP \times OP' = R^2$，则称 P、P' 关于 O 互为反演.

过程

求凸包的过程如下：

- 首先对其微小扰动，避免出现四点共面的情况.
- 对于一个已知凸包，新增一个点 P，将 P 视作一个点光源，向凸包做射线，可以知道，光线的可见面和不可见面一定是由若干条棱隔开的.
- 将光的可见面删去，并新增由其分割棱与 P 构成的平面. 重复此过程即可，由 Pick 定理→12.5@p3098、欧拉公式（在凸多面体中，其顶点 V、边数 E 及面数 F 满足 $V-E+F=2$）和圆的反演，复杂度 $O(n^2)$. [1]

模板题

P4724【模板】三维凸包[3]

重复上述过程即可得到答案.

代码实现

```cpp
#include <cmath>
#include <cstdlib>
#include <iomanip>
#include <iostream>
using namespace std;
constexpr int N = 2010;
constexpr double eps = 1e-9;
int n, cnt, vis[N][N];
double ans;

double Rand() { return rand() / (double)RAND_MAX; }

double reps() { return (Rand() - 0.5) * eps; }

struct Node {
  double x, y, z;

  void shake() {
    x += reps();
    y += reps();
    z += reps();
  }

  double len() { return sqrt(x * x + y * y + z * z); }

  Node operator-(Node A) const { return {x - A.x, y - A.y, z - A.z}; }

  Node operator*(Node A) const {
```

[1] 三维凸包学习小记[2]

```
    return {y * A.z - z * A.y, z * A.x - x * A.z, x * A.y - y * A.x};
  }

  double operator&(Node A) const { return x * A.x + y * A.y + z * A.z; }
} A[N];

struct Face {
  int v[3];

  Node Normal() { return (A[v[1]] - A[v[0]]) * (A[v[2]] - A[v[0]]); }

  double area() { return Normal().len() / 2.0; }
} f[N], C[N];

int see(Face a, Node b) { return ((b - A[a.v[0]]) & a.Normal()) > 0; }

void Convex_3D() {
  f[++cnt] = {1, 2, 3};
  f[++cnt] = {3, 2, 1};

  for (int i = 4, cc = 0; i <= n; i++) {
    for (int j = 1, v; j <= cnt; j++) {
      if (!(v = see(f[j], A[i]))) C[++cc] = f[j];

      for (int k = 0; k < 3; k++) vis[f[j].v[k]][f[j].v[(k + 1) % 3]] = v;
    }

    for (int j = 1; j <= cnt; j++)
      for (int k = 0; k < 3; k++) {
        int x = f[j].v[k], y = f[j].v[(k + 1) % 3];

        if (vis[x][y] && !vis[y][x]) C[++cc] = {x, y, i};
      }

    for (int j = 1; j <= cc; j++) f[j] = C[j];

    cnt = cc;
    cc = 0;
  }
}

int main() {
  cin >> n;

  for (int i = 1; i <= n; i++) cin >> A[i].x >> A[i].y >> A[i].z,
A[i].shake();

  Convex_3D();

  for (int i = 1; i <= cnt; i++) ans += f[i].area();

  cout << fixed << setprecision(3) << ans << '\n';
```

```
80    return 0;
81 }
```

练习

- UVa11626 Convex Hull[4]
- 「USACO5.1」圈奶牛 Fencing the Cows[5]
- POJ1873 The Fortified Forest[6]
- POJ1113 Wall[7]
- USACO22JAN Multiple Choice Test P[8]
- 「SHOI2012」信用卡凸包[9]

参考资料与注释

[1] 例题 [JSOI2018] 战争 

[2] 三维凸包学习小记 

[3] P4724【模板】三维凸包 

[4] UVa11626 Convex Hull 

[5] 「USACO5.1」圈奶牛 Fencing the Cows 

[6] POJ1873 The Fortified Forest 

[7] POJ1113 Wall 

[8] USACO22JAN Multiple Choice Test P 

[9] 「SHOI2012」信用卡凸包 

12.8 扫描线

引入

扫描线一般运用在图形上面，它和它的字面意思十分相似，就是一条线在整个图上扫来扫去，它一般被用来解决图形面积，周长，以及二维数点等问题.

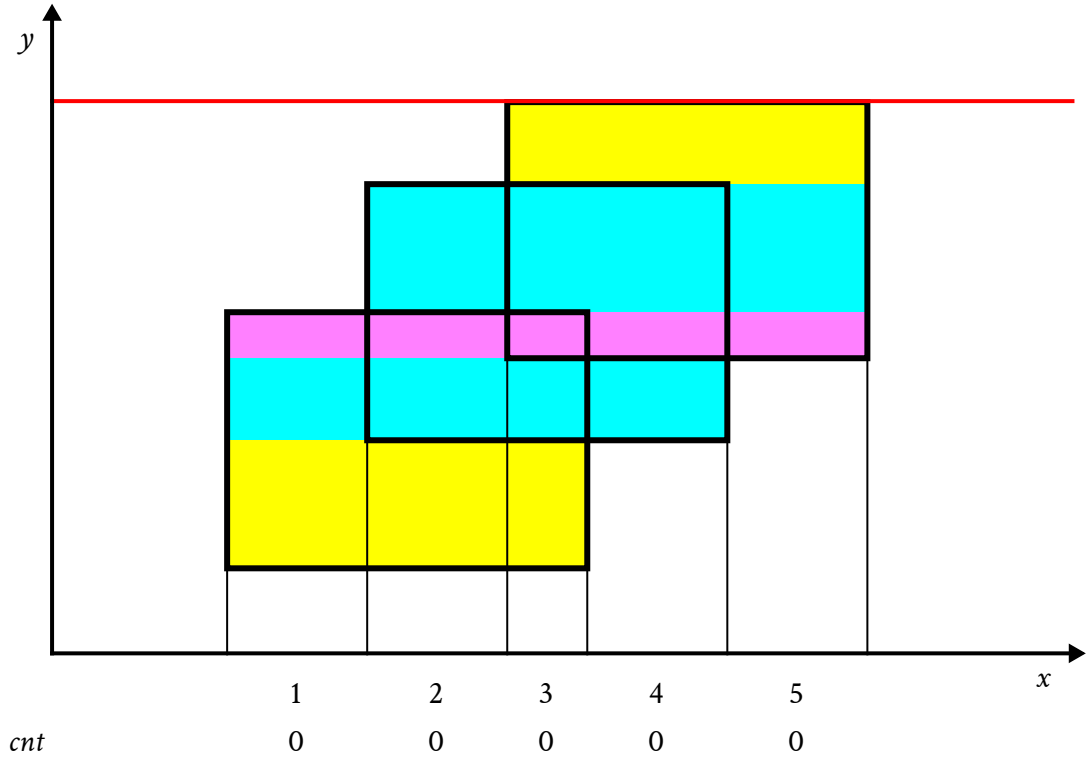
二维矩形面积并问题

在二维坐标系上，给出多个矩形的左下以及右上坐标，求出所有矩形构成的图形的面积.

过程

根据图片可知总面积可以直接暴力即可求出面积，如果数据大了怎么办？这时就需要讲到 **扫描线** 算法.

现在假设我们有一根线，从下往上开始扫描：



如图所示，把整个矩形分成如图各个颜色不同的小矩形，小矩形的高是扫过的距离，然而矩形的水平宽一直在变化.

给每一个矩形的上下边进行标记，下面的边标记为 1，上面的边标记为 -1. 每遇到一个水平边时，让这条边（在横轴投影区间）的权值加上这条边的标记.

Note

这个操作类似遍历括号序列：开括号加 1，闭括号减 1，「权值」对应当前位置的深度，「权值」是否大于 0，对应当前在不在括号里，也就是这段区间是否记入小矩形的宽度.

小矩形（不一定只有一个）的宽度就是整个数轴上权值大于 0 的区间总长度.

实现

用线段树维护矩形的长，也就是整个数轴上覆盖次数大于 0 的点. 需求列举如下：

- 一段区间权值加 1、减 1.
- 统计整个数轴上，区间权值大于 0 的「区间长度和」.

如果你尝试直接用普通线段树模板来实现的话，也许会遇到些挫折. 具体地，由于在区间加时，即使修改区间和节点管理区间重合，我们还是不能常数时间知道覆盖次数如何变化. 这是因为我们不能直接知道：管理范围里有多长的区间会从 1 变成 0（从 0 变成 1）.

这道题只需要朴素的分治就能实现：维护每个节点管理区间中「完全覆盖区间的次数 `v[]`」（类似不用下放的懒惰标记）和「已覆盖的长度 `w[]`」两个信息.

需要 离散化→13.2@p3170.

洛谷 P5490【模板】扫描线 & 矩形面积并[1] 参考代码

```cpp
#include <algorithm>
#include <iostream>

using ll = long long;
constexpr int N = 1e5 + 1;

int n, a[N * 2], tot;   // a[] 和 tot 用于把 x 离散化
ll v[N * 8], w[N * 8];  // 完全覆盖区间的次数、已覆盖的长度

struct St {
  ll x1, x2, y, o;
} b[N * 2];  // 矩形上下边缘

int f(int y) {  // 离散化，把坐标映射到 a 中的下标
  return std::lower_bound(a, a + tot, y) - a;
}

void up(int u, int ul, int ur) {  // pushup
  if (v[u]) w[u] = a[ur] - a[ul];
  // 如果对叶子节点调用 w[u*2+1]，那么线段树需要开 8 倍空间
  // 乘上矩形上下两边就是 16 倍
  else if (ul + 1 == ur)
    w[u] = 0;
  else
    w[u] = w[u * 2 + 1] + w[u * 2 + 2];
}

void add(int lf, int rg, ll o, int u = 0, int ul = 0, int ur = tot - 1) {
  // 区间加
  if (lf == ul && rg == ur) return v[u] += o, up(u, ul, ur), void();
  int um = (ul + ur) / 2;
  if (lf < um) add(lf, std::min(rg, um), o, u * 2 + 1, ul, um);
  if (um < rg) add(std::max(lf, um), rg, o, u * 2 + 2, um, ur);
  up(u, ul, ur);
}

int main() {
  std::cin >> n;
  for (int i = 0, x1, x2, y1, y2; i < n; i++) {
    // y1 是局部变量不会重名
    std::cin >> x1 >> y1 >> x2 >> y2;
    b[i] = {x1, x2, y1, 1};
    b[i + n] = {x1, x2, y2, -1};
    a[i] = x1, a[i + n] = x2;
  }

  std::sort(a, a + n * 2), tot = 1;
  for (int i = 1; i < n * 2; i++)
    if (a[i] != a[tot - 1]) a[tot++] = a[i];  // 离散化

  std::sort(b, b + n * 2,
            [](St &i, St &j) -> bool { return i.y < j.y; });  // 操作排序
```

```

  ll sum = 0;
  add(f(b[0].x1), f(b[0].x2), 1);
  for (int i = 1; i < n * 2; i++) {
    int x1 = f(b[i].x1), x2 = f(b[i].x2);
    sum += (b[i].y - b[i - 1].y) * w[0];  // 对每个小矩形面积求和
    add(x1, x2, b[i].o);
  }
  std::cout << sum << '\n';
}

```

「POJ 1151」Atlantis[2] 参考代码

```
#include <algorithm>
#include <cstdio>
#include <cstring>
constexpr int MAXN = 300;
using namespace std;

int lazy[MAXN << 3];  // 标记了这条线段出现的次数
double s[MAXN << 3];

struct node1 {
  double l, r;
  double sum;
} cl[MAXN << 3];  // 线段树

struct node2 {
  double x, y1, y2;
  int flag;
} p[MAXN << 3];  // 坐标

// 定义sort比较
bool cmp(node2 a, node2 b) { return a.x < b.x; }

// 上传
void pushup(int rt) {
  if (lazy[rt] > 0)
    cl[rt].sum = cl[rt].r - cl[rt].l;
  else
    cl[rt].sum = cl[rt * 2].sum + cl[rt * 2 + 1].sum;
}

// 建树
void build(int rt, int l, int r) {
  if (r - l > 1) {
    cl[rt].l = s[l];
    cl[rt].r = s[r];
    build(rt * 2, l, (l + r) / 2);
    build(rt * 2 + 1, (l + r) / 2, r);
```

```
    pushup(rt);
  } else {
    cl[rt].l = s[l];
    cl[rt].r = s[r];
    cl[rt].sum = 0;
  }
  return;
}

// 更新
void update(int rt, double y1, double y2, int flag) {
  if (cl[rt].l == y1 && cl[rt].r == y2) {
    lazy[rt] += flag;
    pushup(rt);
    return;
  } else {
    if (cl[rt * 2].r > y1) update(rt * 2, y1, min(cl[rt * 2].r, y2), flag);
    if (cl[rt * 2 + 1].l < y2)
      update(rt * 2 + 1, max(cl[rt * 2 + 1].l, y1), y2, flag);
    pushup(rt);
  }
}

int main() {
  int temp = 1, n;
  double x1, y1, x2, y2, ans;
  while (scanf("%d", &n) && n) {
    ans = 0;
    for (int i = 0; i < n; i++) {
      scanf("%lf %lf %lf %lf", &x1, &y1, &x2, &y2);
      p[i].x = x1;
      p[i].y1 = y1;
      p[i].y2 = y2;
      p[i].flag = 1;
      p[i + n].x = x2;
      p[i + n].y1 = y1;
      p[i + n].y2 = y2;
      p[i + n].flag = -1;
      s[i + 1] = y1;
      s[i + n + 1] = y2;
    }
    sort(s + 1, s + (2 * n + 1));  // 离散化
    sort(p, p + 2 * n, cmp);  // 把矩形的边的横坐标从小到大排序
    build(1, 1, 2 * n);       // 建树
    memset(lazy, 0, sizeof(lazy));
    update(1, p[0].y1, p[0].y2, p[0].flag);
    for (int i = 1; i < 2 * n; i++) {
      ans += (p[i].x - p[i - 1].x) * cl[1].sum;
      update(1, p[i].y1, p[i].y2, p[i].flag);
    }
    printf("Test case #%d\nTotal explored area: %.2lf\n\n", temp++, ans);
  }
```

```
90   return 0;
91 }
```

练习

- 「POJ1177」 Picture[3]
- 「POJ3832」 Posters[4]
- 洛谷 P1856 [IOI1998] [USACO5.5] 矩形周长 Picture[5]
  - 横边贡献就是覆盖长度变化量.
  - 两个方向分别算一次可以避免竖直边的讨论.
  - 操作排序时注意考虑两个矩形边重合的情况.
  - 数据范围允许不用线段树，直接平方时间模拟.

B 维正交范围

B 维正交范围指在一个 B 维直角坐标系下，第 i 维坐标在一个整数范围 $[l_i, r_i]$ 间，内部的点集.

一般来说，一维正交范围简称区间，二维正交范围简称矩形，三维正交范围简称立方体（我们常说的二维数点就是二维正交范围）.

对于一个静态的二维问题，我们可以使用扫描线扫一维，数据结构维护另一维. 在扫描线从左到右扫的过程中，会在数据结构维护的那一维上产生一些修改与查询. 如果查询的信息可差分的话直接使用差分，否则需要使用分治. 差分一般用树状数组和线段树维护，但因为树状数组好写而且常数小，所以大部分人会选择用树状数组来维护. 分治一般是 CDQ 分治（但是这里不涉及分治）.

另一种比较容易理解的看待问题的角度是站在序列角度，而不站在二维平面角度. 如果我们这样看待问题，则扫描线实际上是枚举了右端点 $r = 1 \cdots n$，维护一个数据结构，支持查询对于当前的 r，给定一个值 l，l 到 r 的答案是什么. 即扫描线扫询问右端点，数据结构维护所有左端点的答案，或者说遍历一维，数据结果维护另一维.

复杂度一般为 $O((n+m)\log n)$.

二维数点

给一个长为 n 的序列，有 m 次查询，每次查区间 $[l, r]$ 中值在 $[x, y]$ 内的元素个数.

这个问题就叫做二维数点. 我们可以发现等价于我们要查询一个二维平面上矩形内的点的数量和. 这里讲一下这个问题最简单的处理方法，扫描线 + 树状数组.

很显然，这个问题是一个静态的二维问题，我们通过扫描线可以将静态的二维问题转换为动态的一维问题. 维护动态的一维问题就使用数据结构维护序列，这里可以使用树状数组.

先将所有的询问离散化，用树状数组维护权值，对于每次询问的 l 和 r，我们在枚举到 $l-1$ 时统计当前位于区间 $[x, y]$ 内的数的数量 a，继续向后枚举，枚举到 r 时统计当前位于区间 $[x, y]$ 内的数的数量 b，$b - a$ 即为该次询问的答案.

例题

洛谷 P2163 [SHOI2007] 园丁的烦恼[6]

首先离散化. 设一个左下角为 $(0, 0)$，右上角为 (x, y) 的矩形内包含 $ans_{x,y}$ 个点. 则询问的答案可以被差分为 $ans_{c,d} - ans_{a-1,d} - ans_{c,b-1} + ans_{a-1,b-1}$.

代码

```cpp
#include <algorithm>
#include <iostream>

int n, m;
int x[500010], y[500010], ans[500010];
int ax[1500010], ay[1500010], tx, ty;  // 离散化

struct query {
  int a, b, c, d;
} q[500010];  // 保存查询操作方便离散化

struct ope {
  int type, x, y, id;

  ope(int type = 0, int x = 0, int y = 0, int id = 0) {
    this->type = type, this->x = x, this->y = y, this->id = id;
  }

  bool operator<(const ope& rhs) const {
    if (x == rhs.x) return type < rhs.type;
    return x < rhs.x;
  }
};

ope op[2500010];
int tot;  // 操作总数

int sum[1500010];  // 树状数组

int lowbit(int x) { return x & (-x); }

void add(int x, int k) {
  while (x <= 1500000) {
    sum[x] = sum[x] + k;
    x = x + lowbit(x);
  }
}

int getsum(int x) {
  int ret = 0;
  while (x > 0) {
    ret = ret + sum[x];
    x = x - lowbit(x);
  }
  return ret;
}

using std::cin;
using std::cout;
```

```cpp

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m, tx = n, ty = n;
  for (int i = 1; i <= n; i++) cin >> x[i] >> y[i], ax[i] = x[i], ay[i] = y[i];
  for (int i = 1, l, r; i <= m; i++) {
    cin >> q[i].a >> q[i].b >> q[i].c >> q[i].d;
    ax[++tx] = q[i].a, ay[++ty] = q[i].b, ax[++tx] = q[i].c, ay[++ty] = q[i].d;
  }
  std::sort(ax + 1, ax + tx + 1), std::sort(ay + 1, ay + ty + 1);
  tx = std::unique(ax + 1, ax + tx + 1) - ax - 1;
  ty = std::unique(ay + 1, ay + ty + 1) - ay - 1;
  for (int i = 1; i <= n; i++) {
    x[i] = std::lower_bound(ax + 1, ax + tx + 1, x[i]) - ax;
    y[i] = std::lower_bound(ay + 1, ay + ty + 1, y[i]) - ay;
    op[++tot] = ope(0, x[i], y[i], i);  // 加点操作
  }
  for (int i = 1; i <= m; i++) {
    q[i].a = std::lower_bound(ax + 1, ax + tx + 1, q[i].a) - ax;
    q[i].b = std::lower_bound(ay + 1, ay + ty + 1, q[i].b) - ay;
    q[i].c = std::lower_bound(ax + 1, ax + tx + 1, q[i].c) - ax;
    q[i].d = std::lower_bound(ay + 1, ay + ty + 1, q[i].d) - ay;
    op[++tot] = ope(1, q[i].c, q[i].d, i);  // 将查询差分
    op[++tot] = ope(1, q[i].a - 1, q[i].b - 1, i);
    op[++tot] = ope(2, q[i].a - 1, q[i].d, i);
    op[++tot] = ope(2, q[i].c, q[i].b - 1, i);
  }
  std::sort(op + 1, op + tot + 1);  // 将操作按横坐标排序，且优先执行加点操作
  for (int i = 1; i <= tot; i++) {
    if (op[i].type == 0)
      add(op[i].y, 1);
    else if (op[i].type == 1)
      ans[op[i].id] += getsum(op[i].y);
    else
      ans[op[i].id] -= getsum(op[i].y);
  }
  for (int i = 1; i <= m; i++) cout << ans[i] << '\n';
  return 0;
}
```

洛谷 P1908 逆序对[7]

没错，逆序对也可以用扫描线的思维来做. 考虑将求逆序对的个数转化为从后向前枚举每个位置 i，求在区间 $[i+1,n]$ 中，大小在区间 $[0,a_i]$ 中的点的个数. 题目中数据范围为 10^9，很显然要先进行离散化，我们可以考虑从后向前遍历数组，每次遍历到一个数时更新树状数组（线段树），之后统计当前一共有多少个数小于当前枚举的数, 因为我们是从后向前遍历的, 所以比当前值小的数的个数就是他的逆序对的个数，可以用树状数组或线段树进行单点修改和区间查询.

代码

```cpp
#include <algorithm>
#include <iostream>
using ll = long long;
using namespace std;

struct node {
  ll data;
  ll num;
} f[500010];

ll n, ans, a[500010];

bool cmp(node a, node b) {
  if (a.data == b.data) {
    return a.num < b.num;
  }
  return a.data < b.data;
}

ll sum[500010];

int lowbit(int x) { return x & (-x); }

void add(int x, int k) {
  while (x <= n) {
    sum[x] = sum[x] + k;
    x = x + lowbit(x);
  }
}

int getsum(int x) {
  int ret = 0;
  while (x > 0) {
    ret = ret + sum[x];
    x = x - lowbit(x);
  }
  return ret;
}

int main() {
  cin >> n;
  for (ll i = 1; i <= n; i++) {
    cin >> f[i].data;
    f[i].num = i;
  }
  sort(f + 1, f + 1 + n, cmp);
  for (int i = 1; i <= n; i++) {
    a[f[i].num] = i;
  }
```

```
  for (ll i = n; i > 0; i--) {
    ans += getsum(a[i]);
    add(a[i], 1);
  }
  cout << ans;
  return 0;
}

```

洛谷 P1972 [SDOI2009] HH 的项链[8]

简要题意：给定一个序列，多次询问区间 $[l,r]$ 中有多少种不同的数.

这类问题我们可以考虑推导性质，之后使用扫描线枚举所有右端点，数据结构维护每个左端点的答案的方法来实现，我们也可以将问题转换到二维平面上，变为一个矩形查询信息的问题.

在本题中，我们设序列中 a_i 上一次出现的位置为 pre_i，如果 a_i 没有出现过，则 $pre_i=0$. 根据题意，如果一种数在区间中出现多次，只会产生一次贡献. 不妨认为每种数产生贡献的位置是区间中第一次出现的位置，这时可以发现，产生的总贡献即为 $pre_x \le l-1$ 的个数，反证法易证.

现在问题即为：给定一个序列 pre，多次查询区间 $[l,r]$ 中有多少个 $pre_i \le l-1$.

我们可以把 pre_i 看成二维平面的点：i 是横坐标，pre_i 是纵坐标，问题就转化为了二维数点问题：每次询问左下角为 $(l,0)$，右上角为 $(r,l-1)$ 的矩形中有几个点.

注意到这个询问是可差分的，我们可以将询问差分为左下角为 $(0,0)$，右上角为 $(r,l-1)$ 的矩形减去左下角为 $(0,0)$，右上角为 $(l-1,l-1)$ 的矩形有几个点，这样方便我们使用扫描线思想.

单次操作复杂度 $O(\log n)$，共有 n 次加点操作和 $2m$ 次查询操作，总时间复杂度 $O((n+m)\log n)$.

代码

```
#include <algorithm>
#include <iostream>

int n, m, a[1000010], ans[1000010];
int pre[1000010], lst[1000010];  // 处理 pre

struct ope {
  int type, x, y, id;

  ope(int type = 0, int x = 0, int y = 0, int id = 0) {
    this->type = type, this->x = x, this->y = y, this->id = id;
  }

  bool operator<(const ope& rhs) const {
    if (x == rhs.x) return type < rhs.type;
    return x < rhs.x;
  }
};

```

```cpp
ope op[2500010];
int tot;  // 操作总数

int sum[1000010];  // 树状数组

int lowbit(int x) { return x & (-x); }

void add(int x, int k) {
  x++;  // 位置 0 也要进行修改，所以树状数组下标均加 1
  while (x <= n) {
    sum[x] = sum[x] + k;
    x = x + lowbit(x);
  }
}

int getsum(int x) {
  x++;
  int ret = 0;
  while (x > 0) {
    ret = ret + sum[x];
    x = x - lowbit(x);
  }
  return ret;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
    pre[i] = lst[a[i]], lst[a[i]] = i;  // 处理 pre
    op[++tot] = ope{0, i, pre[i], i};   // 加点操作
  }
  cin >> m;
  for (int i = 1, l, r; i <= m; i++) {
    cin >> l >> r;
    op[++tot] = ope{1, r, l - 1, i};  // 将查询差分
    op[++tot] = ope{2, l - 1, l - 1, i};
  }
  std::sort(op + 1, op + tot + 1);  // 将操作按横坐标排序，且优先执行加点操作
  for (int i = 1; i <= tot; i++) {
    if (op[i].type == 0)
      add(op[i].y, 1);
    else if (op[i].type == 1)
      ans[op[i].id] += getsum(op[i].y);
    else
      ans[op[i].id] -= getsum(op[i].y);
  }
```

```
71   for (int i = 1; i <= m; i++) cout << ans[i] << '\n';
72   return 0;
73 }
```

练习

- 洛谷 P8593「KDOI-02」一个弹的投[9] 逆序对的应用.
- AcWing 4709. 三元组[10] 上题的弱化版，同样为逆序对的应用.
- 洛谷 P8773 [蓝桥杯 2022 省 A] 选数异或[11] HH 的项链魔改版.
- 洛谷 P8844 [传智杯 #4 初赛] 小卡与落叶[12] 树上问题转序列问题然后进行二维数点.

总而言之，二维数点的主要思路就是数据结构维护一维，然后枚举另一维.

参考资料

- cnblogs/Yang1208：扫描线讲解，动态开点版线段树[13]
- csdn/riba2534：POJ1151 Atlantis 题解[14]
- csdn/刀刀狗 0102：POJ1151 Atlantis 题解[15]
- 浅谈扫描线[16]

外部参考资料

[1] 洛谷 P5490【模板】扫描线 & 矩形面积并

[2] 「POJ 1151」Atlantis

[3] 「POJ1177」Picture

[4] 「POJ3832」Posters

[5] 洛谷 P1856 [IOI1998] [USACO5.5] 矩形周长 Picture

[6] 洛谷 P2163 [SHOI2007] 园丁的烦恼

[7] 洛谷 P1908 逆序对

[8] 洛谷 P1972 [SDOI2009] HH 的项链

[9] 洛谷 P8593「KDOI-02」一个弹的投

[10] AcWing 4709. 三元组

[11] 洛谷 P8773 [蓝桥杯 2022 省 A] 选数异或

[12] 洛谷 P8844 [传智杯 #4 初赛] 小卡与落叶

[13] cnblogs/Yang1208：扫描线讲解，动态开点版线段树

[14] csdn/riba2534：POJ1151 Atlantis 题解

[15] csdn/刀刀狗 0102：POJ1151 Atlantis 题解

[16] 浅谈扫描线

12.9 旋转卡壳

本页面将主要介绍旋转卡壳.

引入

旋转卡壳（Rotating Calipers，也称「旋转卡尺」）算法，在凸包算法的基础上，通过枚举凸包上某一条边的同时维护其他需要的点，能够在线性时间内求解如凸包直径、最小矩形覆盖等和凸包性质相关的问题.

算法中文名称

该算法比较常见的中文名是「旋转卡壳」. 可以理解为：根据我们枚举的边，可以从每个维护的点画出一条或平行或垂直的直线，为了确保对于当前枚举的边的最优性，我们的任务就是使这些直线

能将凸包正好卡住. 而边通常是按照向某一方向旋转的顺序来枚举, 所以整个过程就是在边「旋转」, 边「卡壳」.

其英文名「rotating calipers」的直译应为「旋转卡尺」, 其中「calipers」的意思是「卡尺」. 第一次提出该术语的论文[1]原意为: 使用一个可动态调整的「卡尺」夹住凸包后, 绕凸包「旋转」该「卡尺」.

求凸包直径

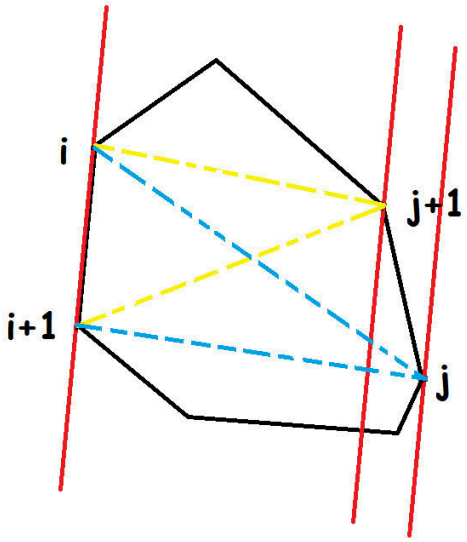
例题 1 :Luogu P1452 Beauty Contest G[1]

给定平面上 n 个点, 求所有点对之间的最长距离. ($2 \le n \le 50000, |x|, |y| \le 10^4$)

过程

首先使用任何一种凸包算法求出给定所有点的凸包, 有着最长距离的点对一定在凸包上. 而由于凸包的形状, 我们发现, 逆时针地遍历凸包上的边, 对于每条边都找到离这条边最远的点, 那么这时随着边的转动, 对应的最远点也在逆时针旋转, 不会有反向的情况, 这意味着我们可以在逆时针枚举凸包上的边时, 记录并维护一个当前最远点, 并不断计算、更新答案.

求出凸包后的数组自然地是按照逆时针旋转的顺序排列, 不过要记得提前将最左下角的 1 节点补到数组最后, 这样在挨个枚举边 $(i, i+1)$ 时, 才能把所有边都枚举到.



枚举过程中, 对于每条边, 都检查 $j+1$ 和边 $(i, i+1)$ 的距离是不是比 j 更大, 如果是就将 j 加一, 否则说明 j 是此边的最优点. 判断点到边的距离大小时可以用叉积分别算出两个三角形的面积 (如图, 黄、蓝两个同底三角形的面积) 并直接比较.

[1] Toussaint, Godfried T. (1983). "Solving geometric problems with the rotating calipers". Proc. MELECON '83, Athens. CiteSeerX 10.1.1.155.5671

实现

核心代码

- C++

```cpp
int sta[N], top;  // 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同

ll pf(ll x) { return x * x; }

ll dis(int p, int q) { return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y); }

ll sqr(int p, int q, int y) { return abs((a[q] - a[p]) * (a[y] - a[q])); }

ll mx;

void get_longest() {  // 求凸包直径
  int j = 3;
  if (top < 4) {
    mx = dis(sta[1], sta[2]);
    return;
  }
  for (int i = 1; i < top; ++i) {
    while (sqr(sta[i], sta[i + 1], sta[j]) <=
           sqr(sta[i], sta[i + 1], sta[j % top + 1]))
      j = j % top + 1;
    mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])));
  }
}
```

- Python

```python
sta = [0] * N
top = 0  # 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同


def pf(x):
    return x * x


def dis(p, q):
    return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y)


def sqr(p, q, y):
    return abs((a[q] - a[p]) * (a[y] - a[q]))


def get_longest():  # 求凸包直径
    j = 3
    if top < 4:
        mx = dis(sta[1], sta[2])
        return
```

```
    for i in range(1, top):
        while sqr(sta[i], sta[i + 1], sta[j]) <= sqr(
            sta[i], sta[i + 1], sta[j % top + 1]
        ):
            j = j % top + 1
        mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])))
```

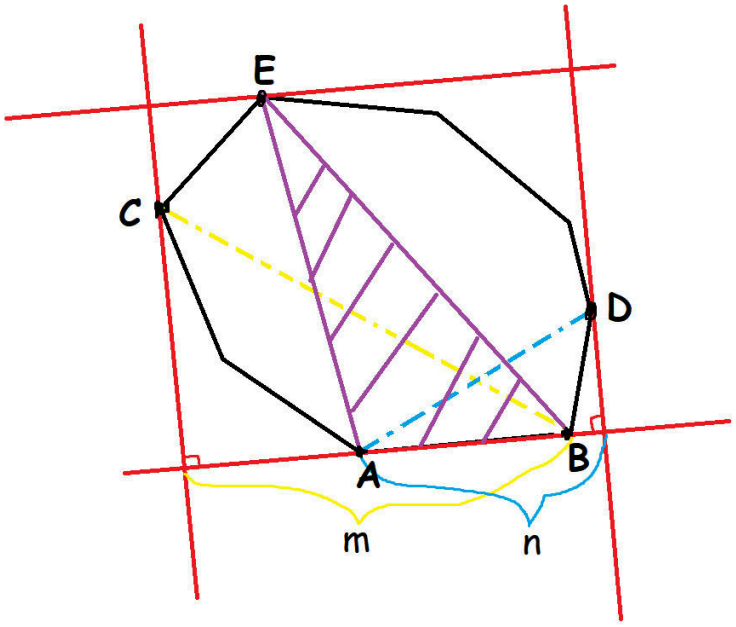
求最小矩形覆盖

Luogu P3187 最小矩形覆盖[2]

给定一些点的坐标，求能够覆盖所有点的最小面积的矩形.（$3 \leq n \leq 50000$）

过程

有了上一道题做铺垫，这道题比较直观的想法仍然是使用旋转卡壳法，不过这次要求的是面积，像上一题一样只维护一个最优点就只能找到一对距离最小的平行线，我们还需要确定矩形的左右边界. 所以这次我们需要维护三个点：一个在所枚举的直线对面的点、两个在不同侧面的点. 对面的最优点仍然是用叉积算面积来比较，此时比较面积就是在比较这个矩形的一个边长. 侧面的最优点则是用点积来比较，因为比较点积就是比较投影的长度，左右两个投影长度相加可以代表这个矩形的另一个边长. 这两个边长的最优性相互独立，因此找到三个最优点的位置就能够确定以当前边所在直线为矩阵的一条边时，能覆盖所有点的矩形最小面积.



最后统计答案时，如果题目没有要求将四个顶点都求出来，其实有一种较为巧妙的利用叉积和点积的方式直接算出矩阵的面积. 设紫色部分面积的两倍为 S，最后的面积就是

$$S \times (|\overrightarrow{AD} \cdot \overrightarrow{AB}| + |\overrightarrow{BC} \cdot \overrightarrow{BA}| - |\overrightarrow{AB} \cdot \overrightarrow{BA}|)/|\overrightarrow{AB} \cdot \overrightarrow{BA}|$$

实现

必要的求凸包过程略去，这里贴出本题核心代码：

核心代码

- C++

```cpp
void get_biggest() {
  int j = 3, l = 2, r = 2;
  double t1, t2, t3, ans = 2e10;
  for (int i = 1; i < top; ++i) {
    while (sqr(sta[i], sta[i + 1], sta[j]) <=
           sqr(sta[i], sta[i + 1], sta[j % top + 1]))
      j = j % top + 1;
    while (dot(sta[i + 1], sta[r % top + 1], sta[i]) >=
           dot(sta[i + 1], sta[r], sta[i]))
      r = r % top + 1;
    if (i == 1) l = r;
    while (dot(sta[i + 1], sta[l % top + 1], sta[i]) <=
           dot(sta[i + 1], sta[l], sta[i]))
      l = l % top + 1;
    t1 = sqr(sta[i], sta[i + 1], sta[j]);
    t2 = dot(sta[i + 1], sta[r], sta[i]) + dot(sta[i + 1], sta[l],
sta[i]);
    t3 = dot(sta[i + 1], sta[i + 1], sta[i]);
    ans = min(ans, t1 * t2 / t3);
  }
}
```

- Python

```python
def get_biggest():
    j = 3
    l = 2
    r = 2
    ans = 2e10
    for i in range(1, top):
        while sqr(sta[i], sta[i + 1], sta[j]) <= sqr(
            sta[i], sta[i + 1], sta[j % top + 1]
        ):
            j = j % top + 1
        while dot(sta[i + 1], sta[r % top + 1], sta[i]) >= dot(
            sta[i + 1], sta[r], sta[i]
        ):
            r = r % top + 1
        if i == 1:
            l = r
        while dot(sta[i + 1], sta[l % top + 1], sta[i]) <= dot(
            sta[i + 1], sta[l], sta[i]
        ):
            l = l % top + 1
        t1 = sqr(sta[i], sta[i + 1], sta[j])
        t2 = dot(sta[i + 1], sta[r], sta[i]) + dot(sta[i + 1], sta[l],
sta[i])
        t3 = dot(sta[i + 1], sta[i + 1], sta[i])
        ans = min(ans, t1 * t2 / t3)
```

练习

- POJ 3608. Bridge Across Islands[3]
- 2011 ACM-ICPC World Finals, Problem K. Trash Removal[4]
- ICPC WF Moscow Invitational Contest - Online Mirror, Problem F. Framing Pictures[5]

参考资料与注释

- https://en.wikipedia.org/wiki/Rotating_calipers[6]
- http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html[7]
- Shamos, Michael (1978). "Computational Geometry" (PDF). Yale University. pp. 76–81.

[1] Luogu P1452 Beauty Contest G

[2] Luogu P3187 最小矩形覆盖

[3] POJ 3608. Bridge Across Islands

[4] 2011 ACM-ICPC World Finals, Problem K. Trash Removal

[5] ICPC WF Moscow Invitational Contest - Online Mirror, Problem F. Framing Pictures

[6] https://en.wikipedia.org/wiki/Rotating_calipers

[7] http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html

12.10 半平面交

Authors: wjy-yy, Ir1d, Xeonacid

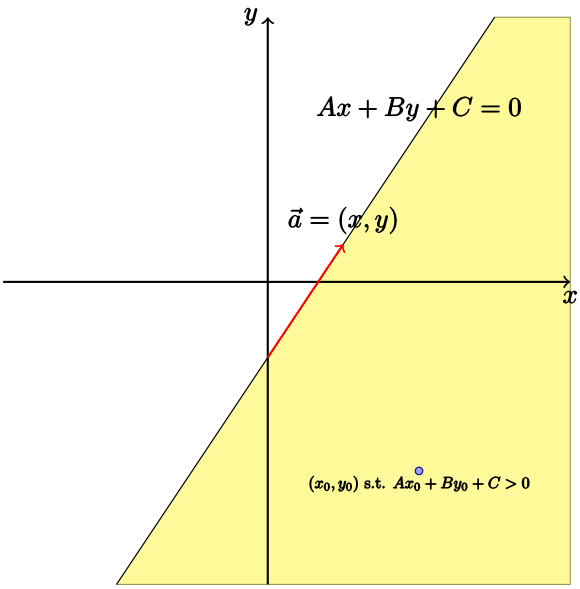
定义

半平面

一条直线和直线的一侧. 半平面是一个点集，因此是一条直线和直线的一侧构成的点集. 当包含直线时，称为闭半平面；当不包含直线时，称为开半平面.

解析式一般为 $Ax + By + C \geq 0$.

在计算几何中用向量表示，整个题统一以向量的左侧或右侧为半平面.



半平面交

半平面交是指多个半平面的交集. 因为半平面是点集, 所以点集的交集仍然是点集. 在平面直角坐标系围成一个区域.

这就很像普通的线性规划问题了, 得到的半平面交就是线性规划中的可行域. 一般情况下半平面交是有限的，经常考察面积等问题的解决.

它可以理解为向量集中每一个向量的右侧的交，或者是下面方程组的解.

$$\begin{cases} A_1x + B_1y + C \geq 0 \\ A_2x + B_2y + C \geq 0 \\ \cdots \end{cases}$$

多边形的核

如果一个点集中的点与多边形上任意一点的连线与多边形没有其他交点，那么这个点集被称为多边形的核.

把多边形的每条边看成是首尾相连的向量，那么这些向量在多边形内部方向的半平面交就是多边形的核.

解法 - S&I 算法

极角排序

C 语言有一个库函数叫做 `atan2(double y,double x)`，可以返回 $\theta \in (-\pi, \pi]$，$\theta = \arctan \frac{y}{x}$.

直接以向量为自变量，调用这个函数，以返回值为关键字排序，得到新的边（向量）集.

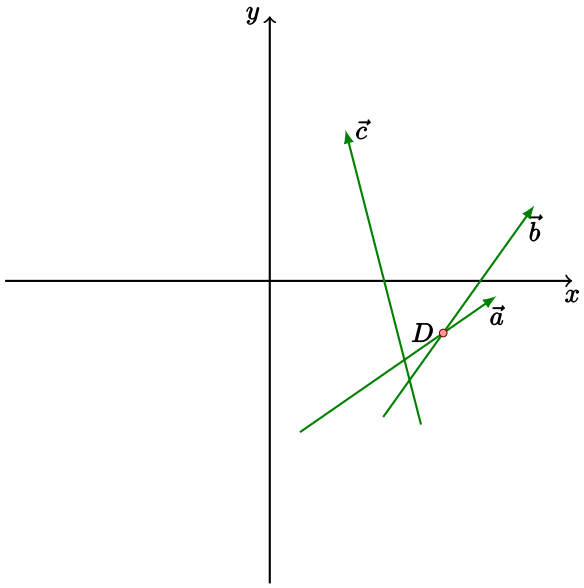
排序时，如果遇到共线向量（且方向相同），则取靠近可行域的一个. 比如两个向量的极角相同，而我们要的是向量的左侧半平面, 那么我们只需要保留左侧的向量. 判断方法是取其中一个向量的起点或终点与另一个比较，检查是在左边还是在右边.

维护单调队列

因为半平面交是一个凸多边形, 所以需要维护一个凸壳. 因为后来加入的只可能会影响最开始加入的或最后加入的边（此时凸壳连通），只需要删除队首和队尾的元素，所以需要用单调队列.

我们遍历排好序了的向量，并维护另一个交点数组. 当单队中元素超过 2 个时，他们之间就会产生交点.

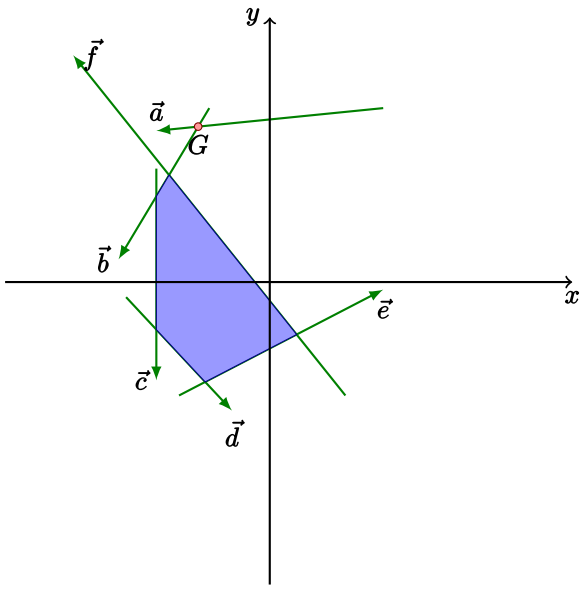
对于当前向量，如果上一个交点在这条向量表示的半平面交的 **异侧**，那么上一条边就没有意义了.



如上图，假设取向量左侧半平面．极角排序后，遍历顺序应该是 $\vec{a} \to \vec{b} \to \vec{c}$．当 $\vec{a}$ 和 $\vec{b}$ 入队时，在交点数组里会产生一个点 D（交点数组保存队列中相同下标的向量与前一向量的交点）．

接下来枚举到 $\vec{c}$ 时，发现 D 在 $\vec{c}$ 的右侧．而因为 **产生 D 的向量的极角一定比 $\vec{c}$ 要小**，所以产生 D 的向量（指 $\vec{b}$）就对半平面交没有影响了．

还有一种可能的情况是快结束的时候，新加入的向量会从队首开始造成影响．



仍然假设取向量左侧半平面．加入向量 $\vec{f}$ 之后，第一个交点 G 就在 $\vec{f}$ 的右侧，我们把上面的判断标准逆过来看，就知道此时应该删除向量 $\vec{a}$，也即 **队首** 的向量．

最后用队首的向量排除一下队尾多余的向量．因为队首的向量会被后面的约束，而队尾的向量不会．此时它们围成了一个环，因此队首的向量就可以约束队尾的向量．

得到半平面交

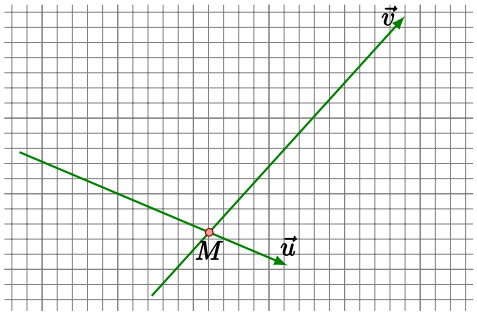
如果半平面交是一个凸 n 边形，最后在交点数组里会得到 n 个点．我们再把它们首尾相连，就是一个统一方向（顺或逆时针）的 n 多边形．

此时就可以用三角剖分求面积了．（求面积是最基础的考法）

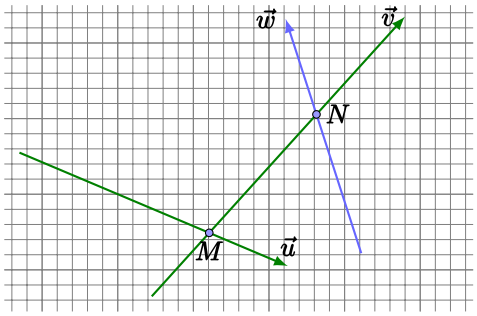
偶尔会出现半平面交不存在或面积为 0 的情况，注意考虑边界.

注意事项

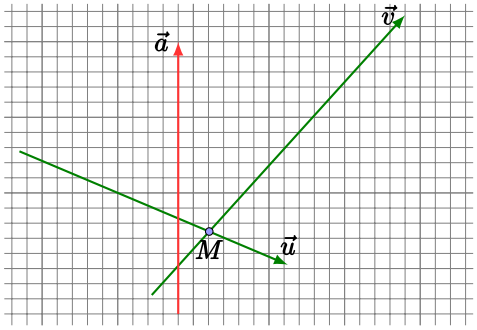
当出现一个可以把队列里的点全部弹出去的向量（即所有队列里的点都在该向量的右侧），则我们 **必须** 先处理队尾，再处理队首．因此在循环中，我们先枚举 `--r;` 的部分，再枚举 `++l;` 的部分，才不会错．原因如下.



一般情况下，我们在队列（队列顺序为 $\{\vec{u},\vec{v}\}$）后面加一条边（向量 $\vec{w}$），会产生一个交点 N，缩小 $\vec{v}$ 后面的范围.



但是毕竟每次操作都是一般的，因此可能会有把 M 点「挤出去」的情况.



如果此时出现了向量 $\vec{a}$，使得 M 在 $\vec{a}$ 的右侧，那么 M 就要出队了．此时如果从队首枚举 `++l`，显然是扩大了范围．实际上 M 点是由 $\vec{u}$ 和 $\vec{v}$ 共同构成的，因此需要考虑影响到现有进程的是 $\vec{u}$ 还是 $\vec{v}$．而因为我们在极角排序后，向量是逆时针顺序，所以 $\vec{v}$ 的影响要更大一些.

就如上图，如果 M 确认在 $\vec{a}$ 的右侧，那么此时 $\vec{v}$ 的影响一定不会对半平面交的答案作出任何贡献.

而我们排除队首的原因是 **当前向量的限制比队首向量要大**，这个条件的前提是队列里有不止两个线段（向量），不然就会出现上面的情况.

所以一定要先排除队尾再排除队首.

代码 - 比较部分

```
friend bool operator<(seg x, seg y) {
  db t1 = atan2((x.b - x.a).y, (x.b - x.a).x);
  db t2 = atan2((y.b - y.a).y, (y.b - y.a).x);  // 求极角
  if (fabs(t1 - t2) > eps)                      // 如果极角不等
    return t1 < t2;
  return (y.a - x.a) * (y.b - x.a) >
         eps;  // 判断向量x在y的哪边，令最靠左的排在最左边
}
```

代码 - 增量部分

```
// pnt its(seg a,seg b)表示求线段a,b的交点
// s[]是极角排序后的向量
// q[]是向量队列
// t[i]是s[i-1]与s[i]的交点
// 【码风】队列的范围是(l,r]
// 求的是向量左侧的半平面
int l = 0, r = 0;
for (int i = 1; i <= n; ++i)
  if (s[i] != s[i - 1]) {
    // 注意要先检查队尾
    while (r - l > 1 && (s[i].b - t[r]) * (s[i].a - t[r]) >
                            eps)  // 如果上一个交点在向量右侧则弹出队尾
      --r;
    while (r - l > 1 && (s[i].b - t[l + 2]) * (s[i].a - t[l + 2]) >
                            eps)  // 如果第一个交点在向量右侧则弹出队首
      ++l;
    q[++r] = s[i];
    if (r - l > 1) t[r] = its(q[r], q[r - 1]);  // 求新交点
  }
while (r - l > 1 &&
       (q[l + 1].b - t[r]) * (q[l + 1].a - t[r]) > eps)  // 注意删除多余元素
  --r;
t[r + 1] = its(q[l + 1], q[r]);  // 再求出新的交点
++r;
// 这里不能在t里面++r需要注意一下……
```

练习

POJ 2451 Uyuw's Concert[1] 注意边界

POJ 1279 Art Gallery[2] 求多边形的核

「CQOI2006」凸多边形[3]

外部参考资料

[1] POJ 2451 Uyuw's Concert

[2] POJ 1279 Art Gallery

[3] 「CQOI2006」凸多边形

12.11 平面最近点对

引入

给定 n 个二维平面上的点，求一组欧几里得距离最近的点对.

下面我们介绍一种时间复杂度为 $O(n \log n)$ 的分治算法来解决这个问题. 该算法在 1975 年由 Franco P. Preparata[1] 提出，Preparata 和 Michael Ian Shamos[2] 证明了该算法在决策树模型下是最优的.

过程

与常规的分治算法一样，我们将这个有 n 个点的集合拆分成两个大小相同的集合 S_1, S_2，并不断递归下去. 但是我们遇到了一个难题: 如何合并? 即如何求出一个点在 S_1 中，另一个点在 S_2 中的最近点对? 这里我们先假设合并操作的时间复杂度为 $O(n)$，可知算法总复杂度为 $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

我们先将所有点按照 x_i 为第一关键字、y_i 为第二关键字排序，并以点 $p_m (m = \lfloor \frac{n}{2} \rfloor)$ 为分界点，拆分点集为 A_1, A_2:

$$A_1 = \{p_i \mid i = 0...m\}$$

$$A_2 = \{p_i \mid i = m+1...n-1\}$$

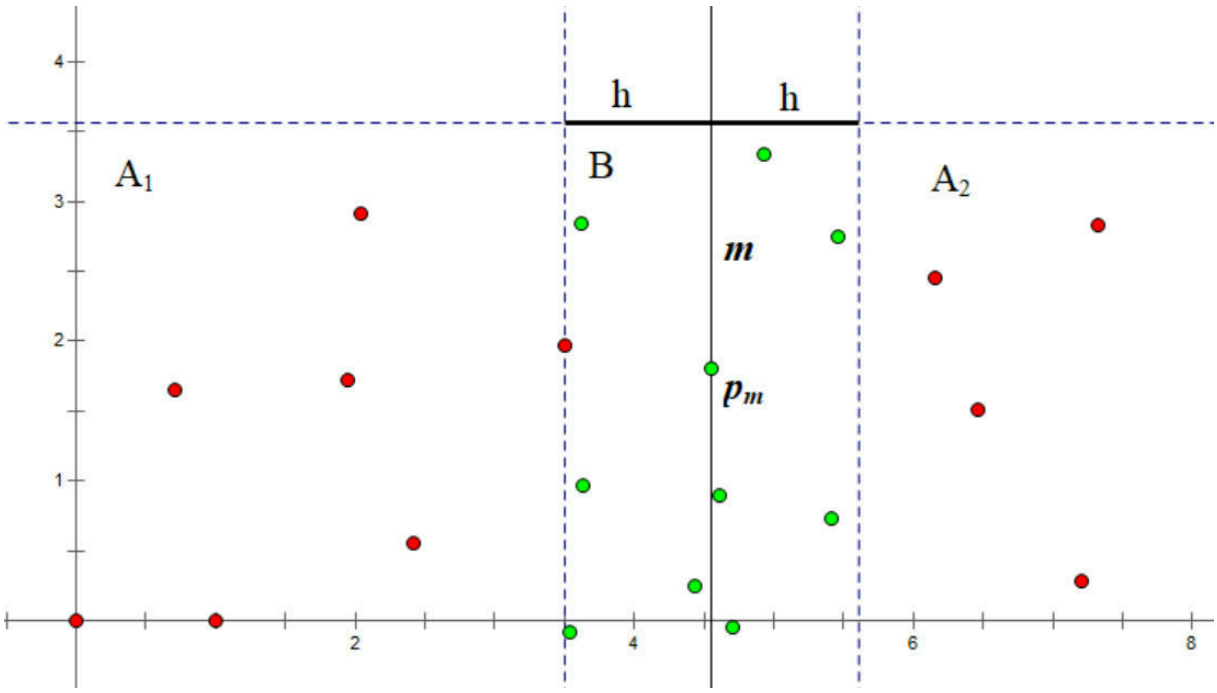
并递归下去，求出两点集各自内部的最近点对，设距离为 h_1, h_2，取较小值设为 h.

现在该合并了! 我们试图找到这样的一组点对，其中一个属于 A_1，另一个属于 A_2，且二者距离小于 h. 因此我们将所有横坐标与 x_m 的差小于 h 的点放入集合 B:

$$B = \{p_i \mid |x_i - x_m| < h\}$$

结合图像，直线 m 将点分成了两部分. m 左侧为 A_1 点集，右侧为 A_2 点集.

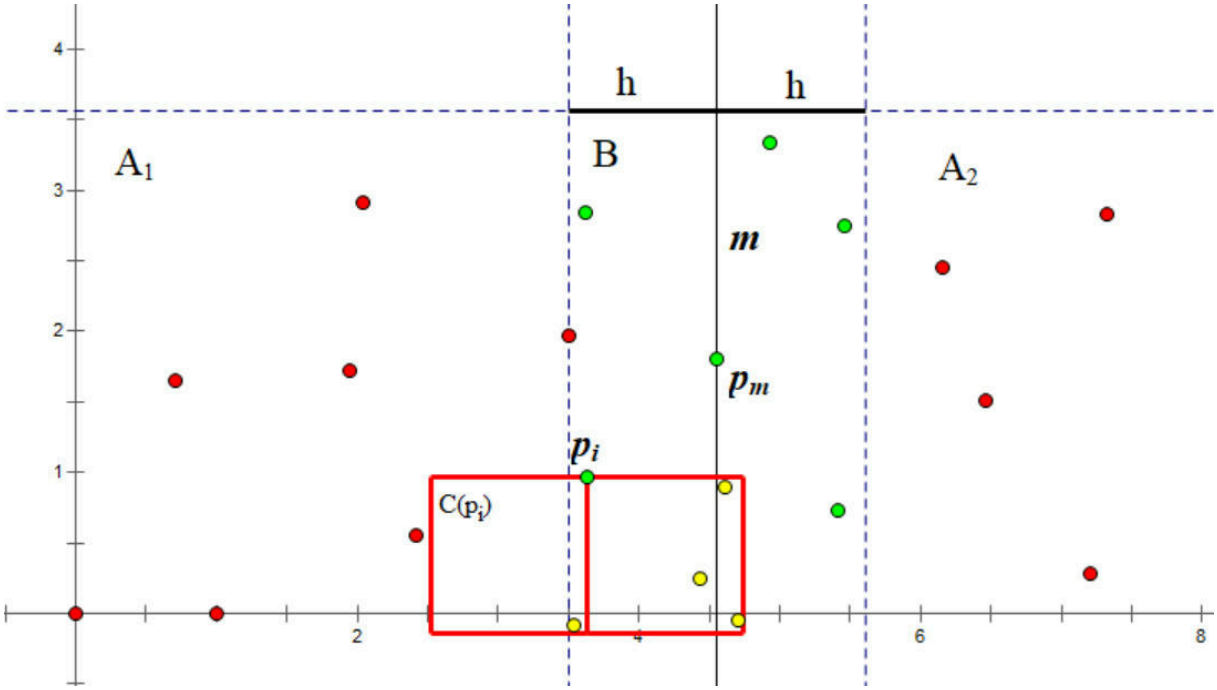
再根据 $B = \{p_i \mid |x_i - x_m| < h\}$ 规则，得到绿色点组成的 B 点集.



对于 B 中的每个点 p_i，我们当前目标是找到一个同样在 B 中、且到其距离小于 h 的点. 为了避免两个点之间互相考虑，我们只考虑那些纵坐标小于 y_i 的点. 显然对于一个合法的点 p_j，$y_i - y_j$ 必须小于 h. 于是我们获得了一个集合 $C(p_i)$:

$$C(p_i) = \{p_j \mid p_j \in B, y_i - h < y_j \leq y_i\}$$

在点集 B 中选一点 p_i，根据 $C(p_i) = \{p_j \mid p_j \in B, y_i - h < y_j \le y_i\}$ 的规则，得到了由红色方框内的黄色点组成的 C 点集.



如果我们将 B 中的点按照 y_i 排序，$C(p_i)$ 将很容易得到，即紧邻 p_i 的连续几个点.

由此我们得到了合并的步骤：

1. 构建集合 B.
2. 将 B 中的点按照 y_i 排序. 通常做法是 $O(n \log n)$，但是我们可以改变策略优化到 $O(n)$（下文讲解）.
3. 对于每个 $p_i \in B$ 考虑 $p_j \in C(p_i)$，对于每对 (p_i, p_j) 计算距离并更新答案（当前所处集合的最近点对）.

注意到我们上文提到了两次排序，因为点坐标全程不变，第一次排序可以只在分治开始前进行一次. 我们令每次递归返回当前点集按 y_i 排序的结果，对于第二次排序，上层直接使用下层的两个分别排序过的点集归并即可.

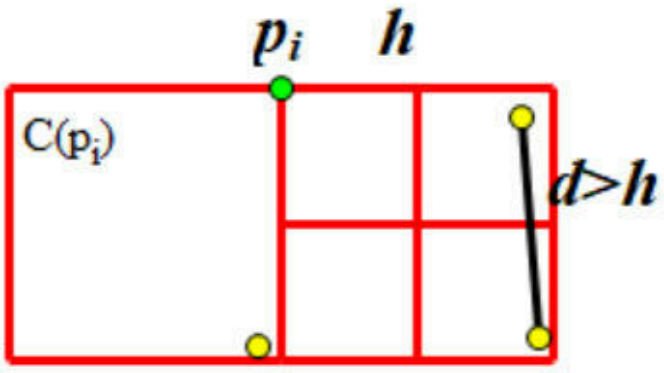
似乎这个算法仍然不优，$|C(p_i)|$ 将处于 $O(n)$ 数量级，导致总复杂度不对. 其实不然，其最大大小为 7，我们给出它的证明：

复杂度证明

我们已经了解到，$C(p_i)$ 中的所有点的纵坐标都在 $(y_i - h, y_i]$ 范围内；且 $C(p_i)$ 中的所有点，和 p_i 本身，横坐标都在 $(x_m - h, x_m + h)$ 范围内. 这构成了一个 $2h \times h$ 的矩形.

我们再将这个矩形拆分为两个 $h \times h$ 的正方形，不考虑 p_i，其中一个正方形中的点为 $C(p_i) \cap A_1$，另一个为 $C(p_i) \cap A_2$，且两个正方形内的任意两点间距离大于 h.（因为它们来自同一下层递归）

我们将一个 $h \times h$ 的正方形拆分为四个 $\frac{h}{2} \times \frac{h}{2}$ 的小正方形. 可以发现，每个小正方形中最多有 1 个点：因为该小正方形中任意两点最大距离是对角线的长度，即 $\frac{h}{\sqrt{2}}$，该数小于 h.



由此，每个正方形中最多有 4 个点，矩形中最多有 8 个点，去掉 p_i 本身，$\max(C(p_i)) = 7$.

参考实现

```cpp
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

const int N = 500000 + 10;

struct point {
  int x, y, id;
};

int n, A, B;
point a[N];
// mindist 是最近距离的平方
long long mindist;

// 更新答案
void upd_ans(const point& a, const point& b) {
  long long dist =
      1LL * (a.x - b.x) * (a.x - b.x) + 1LL * (a.y - b.y) * (a.y - b.y);
  if (dist < mindist) {
    mindist = dist;
    A = a.id;
    B = b.id;
  }
}

// 使用 [l, r) 表示当前分治区间
void DC(int l, int r) {
```

```
  // 当前区间只有一个点，直接返回
  if (l + 1 == r) return;

  int m = (l + r) >> 1;
  int midx = a[m].x;
  DC(l, m);
  DC(m, r);
  // 使用 std::inplace_merge() 进行归并排序
  inplace_merge(a + l, a + m, a + r,
                [&](point a, point b) { return a.y < b.y; });

  vector<point> t;
  for (int i = l; i < r; i++)
    // 距离比较时注意平方，并且比较时不取等号
    if (1LL * (a[i].x - midx) * (a[i].x - midx) < mindist)
t.push_back(a[i]);
  for (int i = 0; i < t.size(); i++)
    for (int j = i + 1; j < t.size(); j++) {
      if (1LL * (t[i].y - t[j].y) * (t[i].y - t[j].y) >= mindist) break;
      upd_ans(t[i], t[j]);
    }
}

void Solve() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &a[i].x, &a[i].y);
    a[i].id = i;
  }
  // 调用前先按横坐标排序
  sort(a, a + n, [&](point x, point y) { return x.x < y.x; });
  mindist = 9'000'000'000'000'000'000'000LL;
  DC(0, n);
  printf("%d %d\n", A, B);
}

int main() {
  int T;
  scanf("%d", &T);
  while (T--) Solve();
  return 0;
}

```

推广：平面最小周长三角形

上述算法有趣地推广到这个问题：在给定的一组点中，选择三个点，使得它们两两的距离之和最小.

算法大体保持不变，每次尝试找到一个比当前答案周长 d 更小的三角形，将所有横坐标与 x_m 的差小于 $\frac{d}{2}$ 的点放入集合 B，尝试更新答案.（周长为 d 的三角形的最长边小于 $\frac{d}{2}$）

非分治算法

其实，除了上面提到的分治算法，还有另一种时间复杂度同样是 $O(n \log n)$ 的非分治算法.

我们可以考虑一种常见的统计序列的思想: 对于每一个元素, 将它和它的左边所有元素的贡献加入到答案中. 平面最近点对问题同样可以使用这种思想.

具体地，我们把所有点按照 x_i 为第一关键字、y_i 为第二关键字排序，并建立一个以 y_i 为关键字的 multiset. 对于每一个位置 i，我们执行以下操作:

1. 将所有满足 $x_i - x_j \geq d$ 的点从集合中删除. 它们不会再对答案有贡献.
2. 对于集合内满足 $|y_i - y_j| < d$ 的所有点，统计它们和 p_i 的距离.
3. 将 p_i 插入到集合中.

由于每个点最多会被插入和删除一次，所以插入和删除点的时间复杂度为 $O(n \log n)$，而统计答案部分的时间复杂度证明与分治算法的时间复杂度证明方法类似，读者不妨一试.

参考实现

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <set>
using namespace std;

constexpr int N = 500000 + 10;

struct point {
  int x, y, id;

  point() {}

  bool operator<(const point& a) const {
    return x < a.x || (x == a.x && y < a.y);
  }
};

int n, A, B;
long long mindist;
point a[N];

void upd_ans(const point& a, const point& b) {
  long long dist =
      1LL * (a.x - b.x) * (a.x - b.x) + 1LL * (a.y - b.y) * (a.y - b.y);
  if (dist < mindist) {
    mindist = dist;
    A = a.id;
    B = b.id;
  }
}

void Solve() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &a[i].x, &a[i].y);
```

```
37      a[i].id = i;
38    }
39    sort(a, a + n);
40    multiset<pair<double, point>> s;
41    mindist = 9'000'000'000'000'000'000LL;
42    for (int i = 0, l = 0; i < n; i++) {
43      for (; l < i && 1LL * (a[i].x - a[l].x) * (a[i].x - a[l].x) >= mindist;
l++)
44        s.erase(s.find({a[l].y, a[l]}));
45      // 需要注意浮点数误差
46      for (auto it =
47                s.lower_bound({(double)a[i].y - sqrt(mindist) + 1e-6,
point()});
48           it != s.end() &&
49           1LL * (it->first - a[i].y) * (it->first - a[i].y) < mindist;
50           it++)
51        upd_ans(it->second, a[i]);
52      s.insert({a[i].y, a[i]});
53    }
54    printf("%d %d\n", A, B);
55  }
56
57  int main() {
58    int T;
59    scanf("%d", &T);
60    while (T--) Solve();
61    return 0;
62  }
63
```

期望线性做法

其实，除了上面提到的时间复杂度为 $O(n \log n)$ 的做法，还有一种 **期望** 复杂度为 $O(n)$ 的算法.

首先将点对 随机打乱→13.6.1@p3252，我们将维护前缀点集的答案. 考虑从前 $i-1$ 个点求出第 i 个点的答案.

记前 $i-1$ 个点的最近点对距离为 s，我们将平面以 s 为边长划分为若干个网格，并存下每个网格内的点（使用 哈希表→10.5@p1995），然后检查第 i 个点所在网格的周围九个网格中的所有点，并更新答案. 注意到需检查的点的个数是 $O(1)$ 的，因为前 $i-1$ 个点的最近点对距离为 s，从而每个网格不超过 4 个点.

如果这一过程中，答案被更新，我们就重构网格图，否则不重构. 在前 i 个点中，最近点对包含 i 的概率为 $O(\frac{1}{i})$，而重构网格的代价为 $O(i)$，从而第 i 个点的期望代价为 $O(1)$. 于是对于 n 个点，该算法期望为 $O(n)$.

习题

- UVa 10245 "The Closest Pair Problem"[难度：低][3]
- SPOJ #8725 CLOPPAIR "Closest Point Pair"[难度：低][4]
- CODEFORCES Team Olympiad Saratov - 2011 "Minimum amount"[难度：中][5]
- SPOJ #7029 CLOSEST "Closest Triple"[难度：中][6]
- Google Code Jam 2009 Final "Min Perimeter"[难度：中][7]

参考资料与拓展阅读

本页面中的分治算法部分主要译自博文 Нахождение пары ближайших точек[8] 与其英文翻译版 Finding the nearest pair of points[9]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

知乎专栏：计算几何 - 最近点对问题[10]

外部参考资料

[1] Franco P. Preparata

[2] Michael Ian Shamos

[3] UVa 10245 "The Closest Pair Problem"[难度：低]

[4] SPOJ #8725 CLOPPAIR "Closest Point Pair"[难度：低]

[5] CODEFORCES Team Olympiad Saratov - 2011 "Minimum amount"[难度：中]

[6] SPOJ #7029 CLOSEST "Closest Triple"[难度：中]

[7] Google Code Jam 2009 Final "Min Perimeter"[难度：中]

[8] Нахождение пары ближайших точек

[9] Finding the nearest pair of points

[10] 知乎专栏：计算几何 - 最近点对问题

12.12 随机增量法

Authors: Ir1d, TianyiQ, GWBailang553

介绍

随机增量算法是计算几何的一个重要算法，它对理论知识要求不高，算法时间复杂度低，应用范围广大.

增量法 (Incremental Algorithm) 的思想与第一数学归纳法类似，它的本质是将一个问题化为规模刚好小一层的子问题．解决子问题后加入当前的对象．写成递归式是：

$$T(n) = T(n-1) + g(n)$$

增量法形式简洁，可以应用于许多的几何题目中.

增量法往往结合随机化，即随机增量法.

随机增量法的思路是：每次加入一个新的对象时，如果改变答案，那么局部重新计算．因此，随机增量法通常要求容易快速判断加入一个元素是否修改答案，且在随机顺序下答案改变次数的期望较少.

随机增量法常用于求解 最小圆覆盖→12.13@p3153 问题，也可以用于求解半空间交、高维凸包、随机 Delaunay 三角剖分等问题.

练习

最小圆覆盖[1]

「HNOI2012」射箭[2]

CodeForces 442E[3]

参考资料与扩展阅读

随机增量算法 - 解轶伦[4]

https://www.cnblogs.com/aininot260/p/9635757.html[5]

https://blog.csdn.net/u014609452/article/details/62039612[6]

随机 Delaunay 三角剖分[7]

外部参考资料

[1] 最小圆覆盖

[2] 「HNOI2012」射箭

[3] CodeForces 442E

[4] 随机增量算法 - 解轶伦

[5] https://www.cnblogs.com/ainiot260/p/9635757.html

[6] https://blog.csdn.net/u014609452/article/details/62039612

[7] 随机 Delaunay 三角剖分

12.13 最小圆覆盖

> **Authors:** Ir1d, Catreap, TianyiQ, Tiphereth-A, Enter-tainer, Henry-ZHR, HeRaNO, iamtwz, ksyx, ouuan, sshwy, Xeonacid, GWBailang553

引入

对平面上的 n $(n \geq 1)$ 个点 $P_1, P_2, \cdots, P_n$，找出一个半径最小的圆，使得所有点都不在圆外．这个圆称为这些点的 **最小覆盖圆**（Minimum Enclosing Circle，MEC）．

本文中，我们允许半径为 0 的圆，即当所有点重合于一点 P 时，我们认为这 n 的点的最小覆盖圆为圆心为 P、半径为 0 的圆．

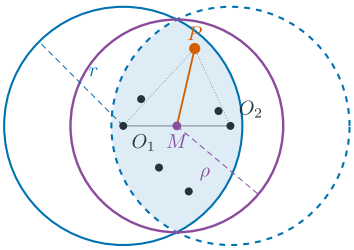
不难发现最小覆盖圆是唯一的．

证明

假设存在两个不同的最小覆盖圆，其圆心分别为 O_1, O_2，半径均为 r．令 M 为 O_1O_2 的中点，则对任意点 P 都有

$$
\begin{aligned}
|P-M|^2 &= \frac{|P-O_1|^2+|P-O_2|^2}{2}-\frac{|O_1-O_2|^2}{4} \\
&\leq r^2-\frac{|O_1-O_2|^2}{4}<r^2.
\end{aligned}
$$

因此，以 M 为圆心可以构造一个半径更小的覆盖圆，产生矛盾．



$|P-M| \leq \rho = \sqrt{r^2-\frac{1}{4}|O_1-O_2|^2} < r$

[1] Welzl, E. (1991). Smallest enclosing disks (balls and ellipsoids). In: Maurer, H. (eds) New Results and New Trends in Computer Science. Lecture Notes in Computer Science, vol 555. Springer, Berlin, Heidelberg.

过程

求解最小圆覆盖问题的常见做法是基于 随机增量法→12.12@p3152 的 Welzl 算法[1]. 该算法的核心操作为：先将点随机打乱，再维护一个点集的最小覆盖圆，并依次向点集加入新点. 虽然算法中包含三层循环，但其期望时间复杂度为 $O(n)$.

加入一个点

我们首先考虑向当前维护的点集中加入新点会发生什么. 设当前圆 C 是已处理点集的最小覆盖圆，现在加入点 P.

- 如果 P 在 C 内部或圆周上，则 C 仍然是最小覆盖圆.
- 如果 P 在 C 外部，此时 P 一定在新最小覆盖圆的圆周上，因此可以将问题转化为：已知 P 在圆周上，求最小覆盖圆的子问题.

为什么圆外的新点一定在新圆的圆周上

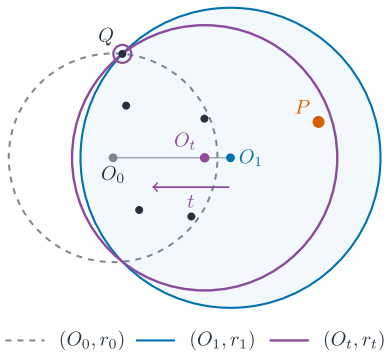
设旧圆、新圆的圆心分别为 O_0, O_1，半径分别为 r_0, r_1，则 $r_0 \le r_1$. 假设 P 严格位于新圆内部. 对 $0 < t < 1$，考虑圆 (O_t, r_t) 满足

$$O_t = (1-t)O_1 + tO_0, \qquad r_t^2 = (1-t)r_1^2 + tr_0^2 - t(1-t)|O_1 - O_0|^2 < r_1^2,$$

此时取充分小的 t 使得 P 仍在圆 (O_t, r_t) 内. 对任意点 Q，我们有

$$|Q - O_t|^2 - r_t^2 = (1-t)(|Q - O_1|^2 - r_1^2) + t(|Q - O_0|^2 - r_0^2).$$

所以旧点集中的点仍能被圆 (O_t, r_t) 覆盖，且原先同时固定在圆 (O_0, r_0) 圆周与圆 (O_1, r_1) 圆周上的点仍在圆 (O_t, r_t) 的圆周上. 由于 P 在圆 (O_t, r_t) 内，所以圆 (O_t, r_t) 是合法的覆盖圆，但是我们有 $r_t < r_1$，与新圆最优矛盾.



上述结论也适用于已经固定了一点或两点在圆周上时加入新点的情况.

三层循环

接下来我们考虑 Welzl 算法的主体. 我们将输入点等概率随机打乱，仍记为 $P_1, P_2, \cdots, P_n$，并令 $S_i = \{P_1, \cdots, P_i\}$，$S_0 = \emptyset$. 维护当前圆 C，初始时圆心为 P_1、半径为 0，圆周上没有固定点.

1. （第一层循环）不固定圆周上的点：依次枚举 $i = 2, \cdots, n$. 若 P_i 已在 C 内则继续；否则在圆周上固定 P_i，重新求覆盖 S_{i-1} 的最小圆.
2. （第二层循环）固定一个圆周点 P_i：先将 C 设为以 P_iP_1 为直径的圆，再依次枚举 $j = 2, \cdots, i-1$. 若 P_j 已在 C 内则继续；否则再在圆周上固定 P_j，重新求覆盖 S_{j-1} 的最小圆.
3. （第三层循环）固定两个圆周点 P_i, P_j：先将 C 设为以 P_iP_j 为直径的圆，再依次枚举 $k = 1, \cdots, j-1$. 若 P_k 已在 C 内则继续；否则将 C 更新为 $\triangle P_iP_jP_k$ 的外接圆.

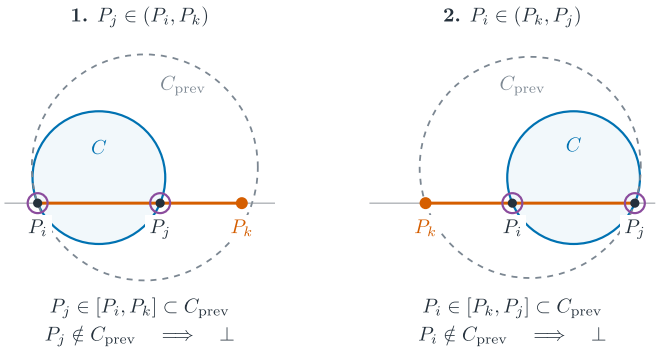
这里「在圆内」均包含圆周，每一行中的当前圆都是满足相应约束的 **最小圆**. 第三层循环结束时覆盖了 S_{j-1}，再加上圆周上的 P_j，恰好得到第二层循环所需的答案；第二层循环结束时，同理得到第一层循

环所需的答案．内层循环的可行性由外层循环保证，而且不难发现第三层循环更新时，三个点 $P_iP_jP_k$ 不会共线，所以该算法是正确的．

第三层循环更新最小圆时，为什么三点不会共线

由于 P_j 曾在经过 P_i 的圆外，因此 $P_i \neq P_j$．若 P_i, P_j, P_k 共线，且 P_k 在当前圆 C 外，则 P_k 必须在线段 P_iP_j 之外．接下来分两种情况讨论：

- 若 P_j 位于 P_i, P_k 之间，则第二层循环枚举到 P_j 时，当前最小圆经过 P_i，且覆盖 S_{j-1} 中的 P_k，因此也覆盖线段 P_iP_k 上的 P_j．这与 P_j 触发第三层循环矛盾．
- 若 P_i 位于 P_j, P_k 之间，则第一层循环枚举到 P_i 时，当前最小圆覆盖 S_{i-1} 中的 P_j, P_k，因此也覆盖线段 P_jP_k 上的 P_i．这与 P_i 触发第二层循环矛盾．



计算外接圆

Welzl 算法涉及的计算几何操作较为常规，主要涉及判定点与圆的位置关系、以两点为直径的圆以及过三点的外接圆这三种，前两种我们略去不表，重点关注一下第三种操作．

给定三个不共线点 $A = (x_A, y_A), B = (x_B, y_B), C = (x_C, y_C)$，它们的外接圆圆心是线段 AB 与 AC 的垂直平分线交点，可以使用 求两条直线的交点→12.2@p3082 中的方法计算．

也可以直接列方程求解．设外心 $O = (x, y)$，则

$$\begin{cases} (x - x_A)^2 + (y - y_A)^2 = (x - x_B)^2 + (y - y_B)^2, \\ (x - x_A)^2 + (y - y_A)^2 = (x - x_C)^2 + (y - y_C)^2, \end{cases}$$

令

$$A_1 = 2(x_B - x_A), B_1 = 2(y_B - y_A), C_1 = x_B^2 + y_B^2 - x_A^2 - y_A^2,$$
$$A_2 = 2(x_C - x_A), B_2 = 2(y_C - y_A), C_2 = x_C^2 + y_C^2 - x_A^2 - y_A^2,$$

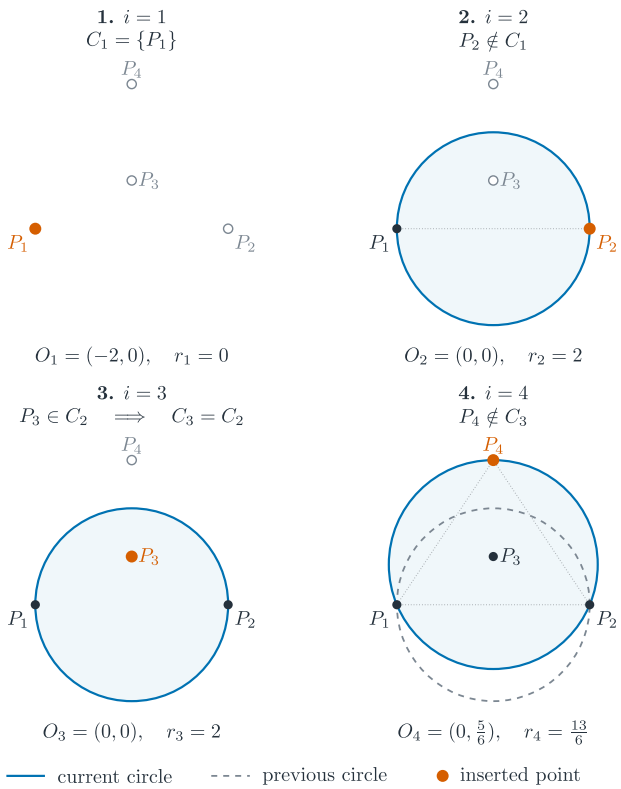
则有

$$x = \frac{C_1B_2 - C_2B_1}{A_1B_2 - A_2B_1}, \quad y = \frac{A_1C_2 - A_2C_1}{A_1B_2 - A_2B_1}.$$

求出圆心后，取 $r = |O - A|$ 即可．分母 $A_1B_2 - A_2B_1$ 非零对应三点不共线．三点接近共线时，该方法的浮点误差可能被放大，实际使用中需要特别留意．

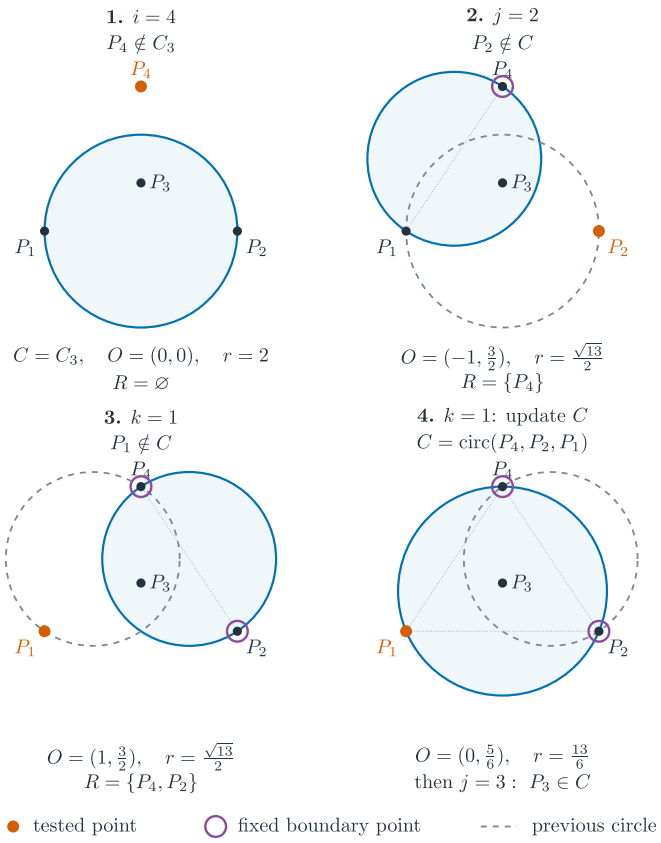
示例

设点 $P_1 = (-2, 0)$、$P_2 = (2, 0)$、$P_3 = (0, 1)$、$P_4 = (0, 3)$．记 C_i 为处理完前 i 个点后的最小覆盖圆，下图展示外层循环的结果．蓝色实线表示当前圆，灰色虚线表示更新前的圆，橙色实心点表示本轮加入的点，灰色空心点表示尚未加入的点．



初始时半径为 0．加入 P_2 后，当前覆盖圆以 P_1P_2 为直径．P_3 在圆内，所以加入它时圆不变．P_4 位于圆外，必须重新计算．重算完成后，圆心为 $(0, 5/6)$、半径为 $13/6$，P_1, P_2, P_4 在圆周上，P_3 在圆内．

下面展开 $i = 4$ 时的重算过程．橙色点表示当前检查的点，紫色外圈标记本层循环固定的圆周点，其集合记为 R．蓝色实线与灰色虚线分别表示当前圆与上一步的圆．



1. P_4 在旧圆 C_3 外，进入第二层循环.
2. 固定 P_4，先取以 P_4P_1 为直径的圆．检查到 $j = 2$ 时，发现 P_2 在这个圆外，进入第三层循环.
3. 固定 P_4, P_2，将当前圆设为以 P_4P_2 为直径的圆．检查到 $k = 1$ 时，发现 P_1 仍在圆外.
4. 更新为 $\triangle P_4P_2P_1$ 的外接圆，完成第三层循环．回到第二层循环后，$j = 3$ 对应的 P_3 已在圆内，因此跳过，得到 C_4.

重算中的临时圆只需满足当前子问题的约束，因此不一定覆盖上一层已经处理过的全部点．例如，第二幅子图中的圆尚未覆盖 P_2，第三幅子图中的圆尚未覆盖 P_1，这正是进入下一层或更新外接圆的原因.

复杂度分析

从流程来看，该算法的最坏时间复杂度上界为 $O(n^3)$，但随机打乱可以降低进入内层循环的概率，使期望时间复杂度降为 $O(n)$.

最小覆盖圆可以由至多三个点确定：只有一个不同点时半径为 0，否则，最小覆盖圆由一对直径端点或三个不共线的点确定．固定一个圆周点后，只需再用至多两个点即可确定最优解.

固定一个圆周点后，为什么至多还需要两个点

将固定点平移到原点．设圆心向量为 $\vec{u}$，则该圆的半径为 $|\vec{u}|$．点 x 被覆盖的条件等价于

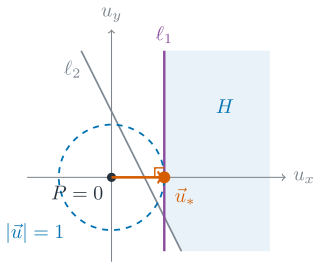
$$|\vec{x} - \vec{u}|^2 \le |\vec{u}|^2 \iff 2\vec{u} \cdot \vec{x} \ge |\vec{x}|^2.$$

除原点外的每个点都会对圆心施加一个半平面约束，可行圆心落在这些 半平面的交→12.10@p3141 中，而最优圆心是其中离原点最近的点．在二维空间中，保留至多两条起作用的约束就能保持这个最近点不变．[2]因此，除了固定点，至多再保留两个输入点就足以保持最优解不变.

下图中，蓝色区域 H 表示所有可行圆心，橙色点 $\vec{u}_*$ 是其中离原点最近的点．去掉灰色边界对应的约束，都不会改变最优圆心.

[2]参见 线性规划→9.15.1@p1774 相关内容.

$$H=\bigcap_{\vec{x}\in S\setminus\{0\}}\{\vec{u}:2\vec{u}\cdot\vec{x}\ge|\vec{x}|^2\},\qquad \vec{u}_*=\arg\min_{\vec{u}\in H}|\vec{u}|$$

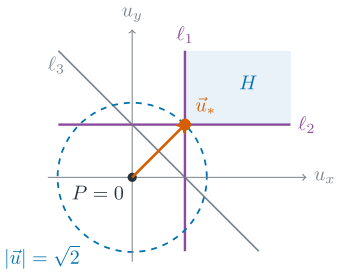


1. $\vec{u}_*=(1,0),\quad r_*=1$

$H_1:\ u_x\ge 1$

$H_2:\ u_x+\frac{1}{2}u_y\ge\frac{5}{8}$

$\arg\min_{\vec{u}\in H_1}|\vec{u}|=\vec{u}_*$



2. $\vec{u}_*=(1,1),\quad r_*=\sqrt{2}$

$H_1:\ u_x\ge 1,\quad H_2:\ u_y\ge 1$

$H_3:\ u_x+u_y\ge 1$

$\arg\min_{\vec{u}\in H_1\cap H_2}|\vec{u}|=\vec{u}_*$

因此，删除后会改变最优圆的点一定属于任意一组这样的确定点. 不固定圆周点时，这种点至多有三个；固定一个圆周点时，这种点至多有两个.

现在进行逆向分析. 先固定 P_i 和第二层循环某一步的无序点集 S_j. 由于初始顺序是随机的，所以 P_j 相当于是从 S_j 中等概率选取的一点. 只有删除 P_j 会改变受约束最优圆时，加入 P_j 才会触发第三层循环，所以触发概率至多为 $\min(1,2/j)$. 第三层扫描需要 $O(j)$ 时间，故整个第二层循环的期望用时为

$$O(1)+\sum_{j=2}^{i-1}\left(O(1)+\frac{2}{j}\cdot O(j)\right)=O(i).$$

类似地，固定第一层的无序点集 S_i，触发第二层重算的点至多有三个，概率至多为 $\min(1,3/i)$. 固定该点以后，之前各点的相对顺序仍然是随机的，因此这次重算的条件期望用时为 $O(i)$. 加上每次判定的开销以及最初打乱的开销，总期望时间复杂度为

$$O(n)+\sum_{i=2}^{n}\left(O(1)+\min\left(1,\frac{3}{i}\right)\cdot O(i)\right)=O(n).$$

例题

洛谷 P1742 最小圆覆盖[1]

给出 N 个点，求包含所有点的最小圆，输出圆的半径和圆心坐标.

下面的实现直接维护圆心与半径，使用 `std::shuffle` 打乱点的顺序. 三层循环分别对应前文的零个、一个、两个固定圆周点的子问题，`geto` 用于求三个点的外心.

参考实现

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <random>

using namespace std;

```

```cpp
int n;
double r;

struct point {
  double x, y;
} p[100005], o;

double sqr(double x) { return x * x; }

double dis(point a, point b) { return sqrt(sqr(a.x - b.x) + sqr(a.y - b.y)); }

bool cmp(double a, double b) { return fabs(a - b) < 1e-8; }

point geto(point a, point b, point c) {
  double a1, a2, b1, b2, c1, c2;
  point ans;
  a1 = 2 * (b.x - a.x), b1 = 2 * (b.y - a.y),
  c1 = sqr(b.x) - sqr(a.x) + sqr(b.y) - sqr(a.y);
  a2 = 2 * (c.x - a.x), b2 = 2 * (c.y - a.y),
  c2 = sqr(c.x) - sqr(a.x) + sqr(c.y) - sqr(a.y);
  if (cmp(a1, 0)) {
    ans.y = c1 / b1;
    ans.x = (c2 - ans.y * b2) / a2;
  } else if (cmp(b1, 0)) {
    ans.x = c1 / a1;
    ans.y = (c2 - ans.x * a2) / b2;
  } else {
    ans.x = (c2 * b1 - c1 * b2) / (a2 * b1 - a1 * b2);
    ans.y = (c2 * a1 - c1 * a2) / (b2 * a1 - b1 * a2);
  }
  return ans;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%lf%lf", &p[i].x, &p[i].y);
  mt19937 rnd(random_device{}());
  shuffle(p + 1, p + n + 1, rnd);
  o = p[1];
  for (int i = 1; i <= n; i++) {
    if (dis(o, p[i]) < r || cmp(dis(o, p[i]), r)) continue;
    o.x = (p[i].x + p[1].x) / 2;
    o.y = (p[i].y + p[1].y) / 2;
    r = dis(p[i], p[1]) / 2;
    for (int j = 2; j < i; j++) {
      if (dis(o, p[j]) < r || cmp(dis(o, p[j]), r)) continue;
      o.x = (p[i].x + p[j].x) / 2;
      o.y = (p[i].y + p[j].y) / 2;
      r = dis(p[i], p[j]) / 2;
      for (int k = 1; k < j; k++) {
        if (dis(o, p[k]) < r || cmp(dis(o, p[k]), r)) continue;
```

```
61          o = geto(p[i], p[j], p[k]);
62          r = dis(o, p[i]);
63        }
64      }
65    }
66    printf("%.10lf\n%.10lf %.10lf", r, o.x, o.y);
67    return 0;
68  }
69
```

练习

- 「POI 2011」 WYK-Plot[2]
- Library Checker - Minimum Enclosing Circle[3]

参考资料与注释

- Smallest-circle problem - Wikipedia[4]
- Minimum Enclosing Circle - Algorithms for Competitive Programming[5]
- Computational Geometry Lecture 4: Smallest enclosing circles and more - University of Florida[6]

[1] 洛谷 P1742 最小圆覆盖

[2] 「POI 2011」 WYK-Plot

[3] Library Checker - Minimum Enclosing Circle

[4] Smallest-circle problem - Wikipedia

[5] Minimum Enclosing Circle - Algorithms for Competitive Programming

[6] Computational Geometry Lecture 4: Smallest enclosing circles and more - University of Florida

12.14 反演变换

Authors: hyp1231, 383494

引入

反演变换适用于题目中存在多个圆/直线之间的相切关系的情况．利用反演变换的性质，在反演空间求解问题，可以大幅简化计算．

定义

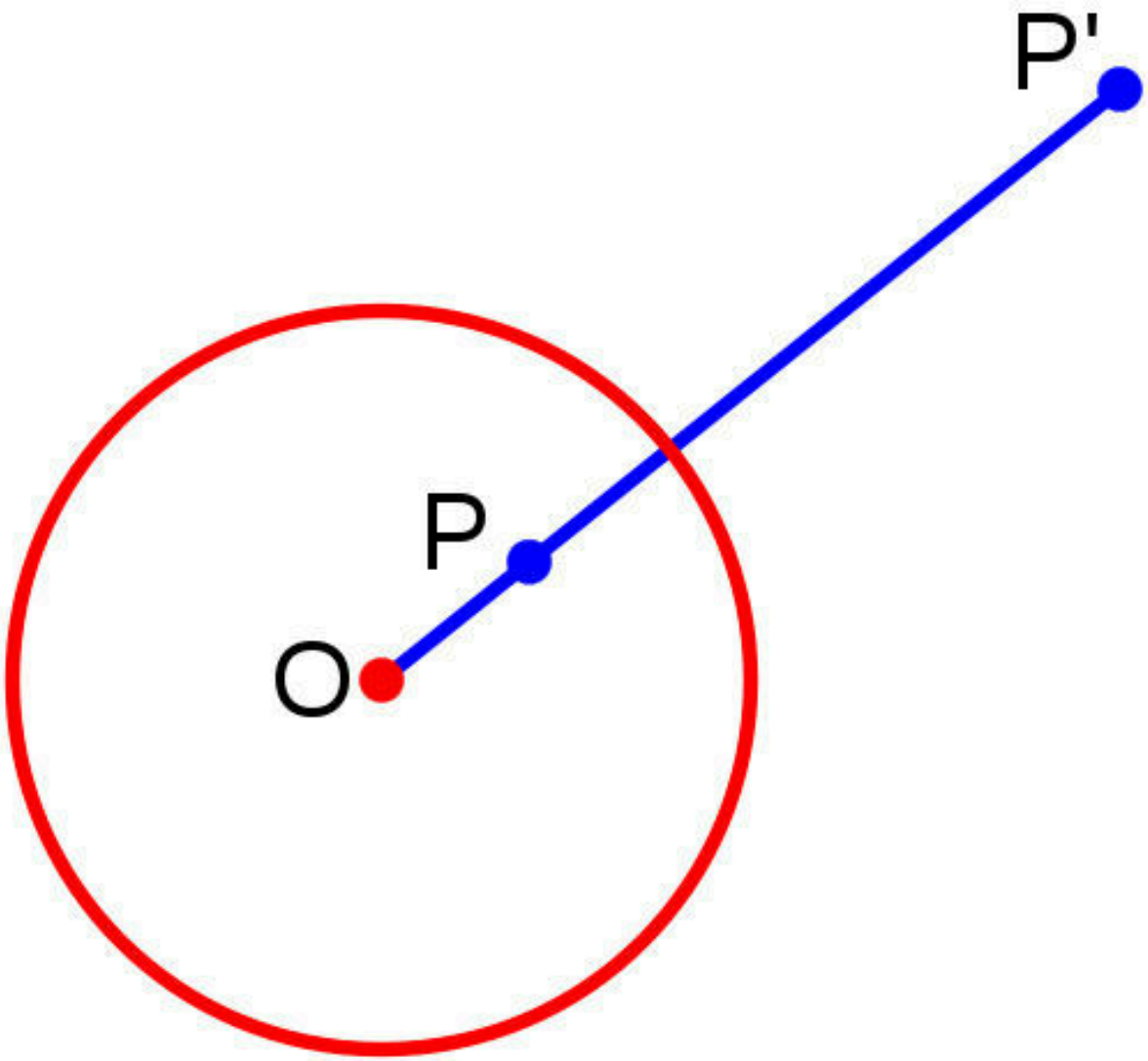
给定反演中心点 O 和反演半径 R．若平面上点 P 和 P' 满足：

- 点 P' 在射线 $\overrightarrow{OP}$ 上
- $|OP| \cdot |OP'| = R^2$

则称点 P 和点 P' 互为反演点．

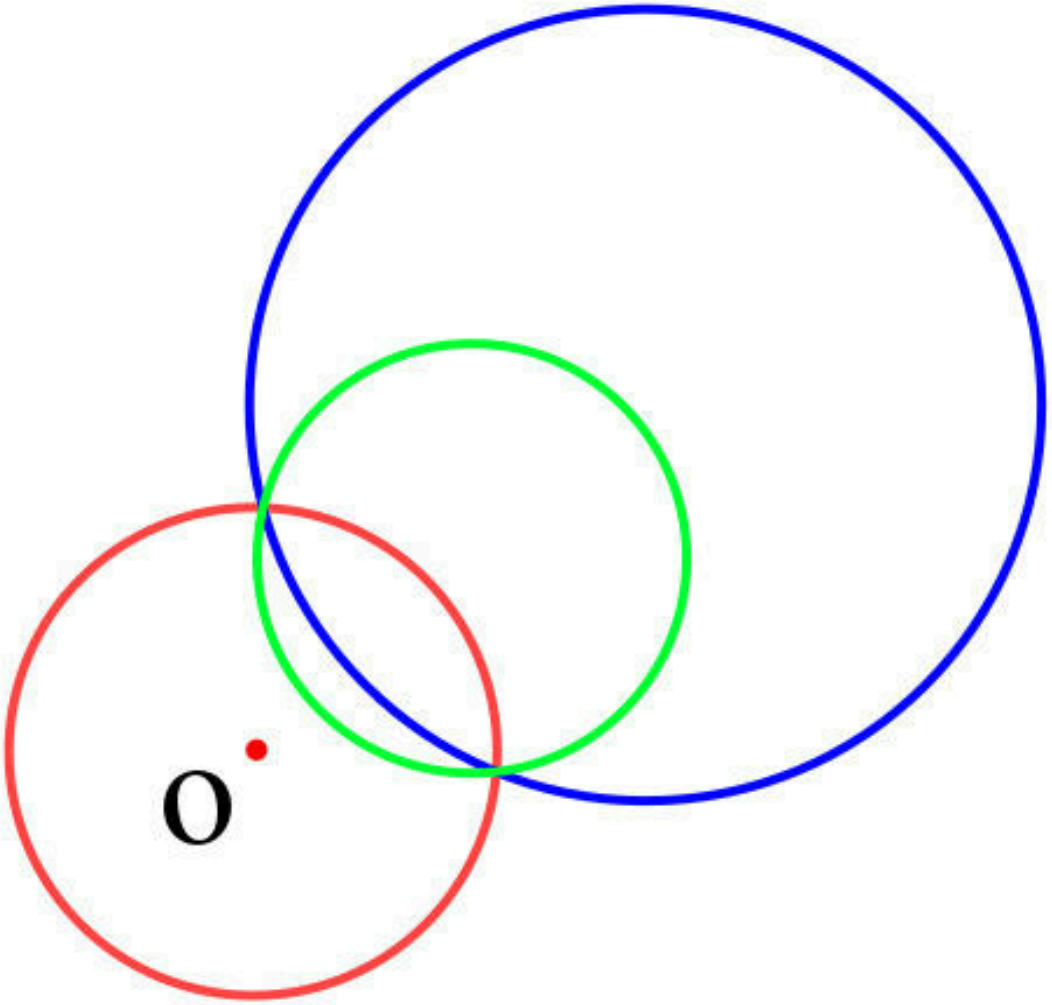
解释

下图所示即为平面上一点 P 的反演：



性质

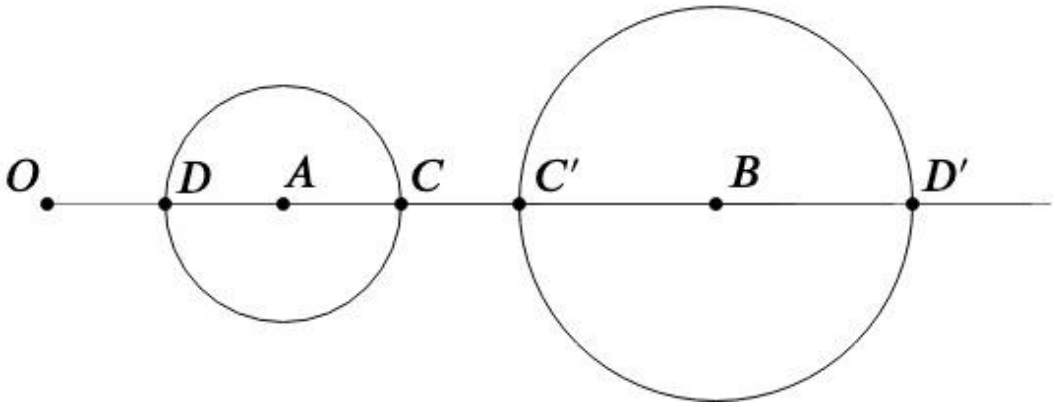
1. 圆 O 外的点的反演点在圆 O 内，反之亦然；圆 O 上的点的反演点为其自身.
2. 不过点 O 的圆 A，其反演图形也是不过点 O 的圆.



- 记圆 A 半径为 r_1，其反演图形圆 B 半径为 r_2，则有：

$$r_2 = \frac{1}{2}\left(\frac{1}{|OA|-r_1} - \frac{1}{|OA|+r_1}\right)R^2$$

证明



根据反演变换定义：

$$|OC|\cdot|OC'|=(|OA|+r_1)\cdot(|OB|-r_2)=R^2$$
$$|OD|\cdot|OD'|=(|OA|-r_1)\cdot(|OB|+r_2)=R^2$$

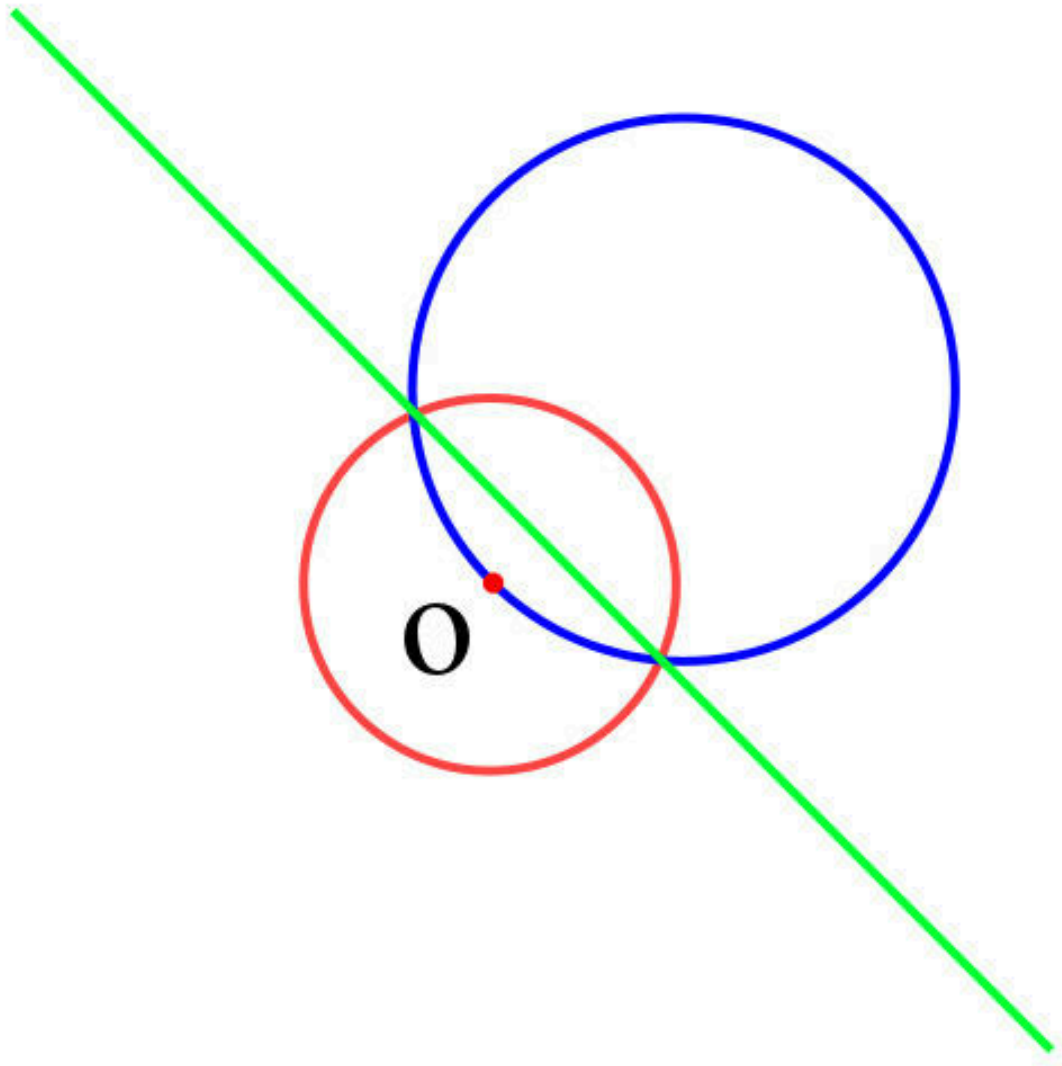
消掉 $|OB|$，解方程即可.

- 记点 O 坐标为 (x_0,y_0)，点 A 坐标为 x_1,y_1，点 B 坐标为 x_2,y_2，则有：

$$x_2=x_0+\frac{|OB|}{|OA|}(x_1-x_0)$$
$$y_2=y_0+\frac{|OB|}{|OA|}(y_1-y_0)$$

其中 $|OB|$ 可在上述求 r_2 的过程中计算得到.

3. 过点 O 的圆 A，其反演图形是不过点 O 的直线. 因为圆 A 上无限接近点 O 的一点，其反演点离点 O 无限远.



4. 两个图形相切且存在不为点 O 的切点，则他们的反演图形也相切.

例题

「ICPC 2013 杭州赛区」Problem of Apollonius[1]

题目大意

求过两圆外一点，且与两圆相切的所有的圆.

解法

首先考虑解析几何解法，似乎很难求解.

考虑以需要经过的点为反演中心进行反演（反演半径任意），所求的圆的反演图形是一条直线（应用性质 3），且与题目给出两圆的反演图形（性质 2）相切（性质 4）.

于是题目经过反演变换后转变为：求两圆的所有公切线.

求出公切线后，反演回原平面即可.

示例代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;

constexpr double EPS = 1e-8;   // 精度系数
const double PI = acos(-1.0);  // π
constexpr int N = 4;

// 点的定义
struct Point {
  double x, y;

  Point(double x = 0, double y = 0) : x(x), y(y) {}

  bool operator<(Point A) const { return x == A.x ? y < A.y : x < A.x; }
};

// 向量的定义
using Vector = Point;

// 向量加法
Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }

// 向量减法
Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }

// 向量数乘
Vector operator*(Vector A, double p) { return Vector(A.x * p, A.y * p); }

// 向量数除
```

```cpp
Vector operator/(Vector A, double p) { return Vector(A.x / p, A.y / p); }

// 与0的关系
int dcmp(double x) {
  if (fabs(x) < EPS) return 0;
  return x < 0 ? -1 : 1;
}

// 向量点乘
double Dot(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }

// 向量长度
double Length(Vector A) { return sqrt(Dot(A, A)); }

// 向量叉乘
double Cross(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }

// 点在直线上投影
Point GetLineProjection(Point P, Point A, Point B) {
  Vector v = B - A;
  return A + v * (Dot(v, P - A) / Dot(v, v));
}

// 圆
struct Circle {
  Point c;
  double r;

  Circle() : c(Point(0, 0)), r(0) {}

  Circle(Point c, double r = 0) : c(c), r(r) {}

  // 输入极角返回点坐标
  Point point(double a) { return Point(c.x + cos(a) * r, c.y + sin(a) * r); }
};

// 两圆公切线 返回切线的条数，-1表示无穷多条切线
// a[i] 和 b[i] 分别是第i条切线在圆A和圆B上的切点
int getTangents(Circle A, Circle B, Point* a, Point* b) {
  int cnt = 0;
  if (A.r < B.r) {
    swap(A, B);
    swap(a, b);
  }
  double d2 =
      (A.c.x - B.c.x) * (A.c.x - B.c.x) + (A.c.y - B.c.y) * (A.c.y - B.c.y);
  double rdiff = A.r - B.r;
  double rsum = A.r + B.r;
  if (dcmp(d2 - rdiff * rdiff) < 0) return 0;  // 内含

  double base = atan2(B.c.y - A.c.y, B.c.x - A.c.x);
```

```
  if (dcmp(d2) == 0 && dcmp(A.r - B.r) == 0) return -1;  // 无限多条切线
  if (dcmp(d2 - rdiff * rdiff) == 0) {  // 内切，一条切线
    a[cnt] = A.point(base);
    b[cnt] = B.point(base);
    ++cnt;
    return 1;
  }
  // 有外公切线
  double ang = acos(rdiff / sqrt(d2));
  a[cnt] = A.point(base + ang);
  b[cnt] = B.point(base + ang);
  ++cnt;
  a[cnt] = A.point(base - ang);
  b[cnt] = B.point(base - ang);
  ++cnt;
  if (dcmp(d2 - rsum * rsum) == 0) {  // 一条内公切线
    a[cnt] = A.point(base);
    b[cnt] = B.point(PI + base);
    ++cnt;
  } else if (dcmp(d2 - rsum * rsum) > 0) {  // 两条内公切线
    double ang = acos(rsum / sqrt(d2));
    a[cnt] = A.point(base + ang);
    b[cnt] = B.point(PI + base + ang);
    ++cnt;
    a[cnt] = A.point(base - ang);
    b[cnt] = B.point(PI + base - ang);
    ++cnt;
  }
  return cnt;
}

// 点 O 在圆 A 外，求圆 A 的反演圆 B，R 是反演半径
Circle Inversion_C2C(Point O, double R, Circle A) {
  double OA = Length(A.c - O);
  double RB = 0.5 * ((1 / (OA - A.r)) - (1 / (OA + A.r))) * R * R;
  double OB = OA * RB / A.r;
  double Bx = O.x + (A.c.x - O.x) * OB / OA;
  double By = O.y + (A.c.y - O.y) * OB / OA;
  return Circle(Point(Bx, By), RB);
}

// 直线反演为过 O 点的圆 B，R 是反演半径
Circle Inversion_L2C(Point O, double R, Point A, Vector v) {
  Point P = GetLineProjection(O, A, A + v);
  double d = Length(O - P);
  double RB = R * R / (2 * d);
  Vector VB = (P - O) / d * RB;
  return Circle(O + VB, RB);
}

// 返回 true 如果 A B 两点在直线同侧
bool theSameSideOfLine(Point A, Point B, Point S, Vector v) {
```

```
  return dcmp(Cross(A - S, v)) * dcmp(Cross(B - S, v)) > 0;
}

int main() {
  int T;
  scanf("%d", &T);
  while (T--) {
    Circle A, B;
    Point P;
    scanf("%lf%lf%lf", &A.c.x, &A.c.y, &A.r);
    scanf("%lf%lf%lf", &B.c.x, &B.c.y, &B.r);
    scanf("%lf%lf", &P.x, &P.y);
    Circle NA = Inversion_C2C(P, 10, A);
    Circle NB = Inversion_C2C(P, 10, B);
    Point LA[N], LB[N];
    Circle ansC[N];
    int q = getTangents(NA, NB, LA, LB), ans = 0;
    for (int i = 0; i < q; ++i)
      if (theSameSideOfLine(NA.c, NB.c, LA[i], LB[i] - LA[i])) {
        if (!theSameSideOfLine(P, NA.c, LA[i], LB[i] - LA[i])) continue;
        ansC[ans++] = Inversion_L2C(P, 10, LA[i], LB[i] - LA[i]);
      }
    printf("%d\n", ans);
    for (int i = 0; i < ans; ++i) {
      printf("%.8f %.8f %.8f\n", ansC[i].c.x, ansC[i].c.y, ansC[i].r);
    }
  }

  return 0;
}
```

练习

「ICPC 2017 南宁赛区网络赛」Finding the Radius for an Inserted Circle[2]

「CCPC 2017 网络赛」The Designer[3]

参考资料与拓展阅读

- Inversive geometry - Wikipedia[4]
- 圆的反演变换 - ACdreamers 的博客[5]

外部参考资料

[1] 「ICPC 2013 杭州赛区」Problem of Apollonius

[2] 「ICPC 2017 南宁赛区网络赛」Finding the Radius for an Inserted Circle

[3] 「CCPC 2017 网络赛」The Designer

[4] Inversive geometry - Wikipedia

[5] 圆的反演变换 - ACdreamers 的博客

第十三章

杂项

13.1 杂项简介

这个板块主要介绍的是一些难以分类的算法及 OI 相关知识.

13.2 离散化

Authors: GavinZhengOI, PlanariaIce

简介

离散化是一种数据处理的技巧，本质上可以看成是一种 哈希→8.5@p894，其保证数据在哈希以后仍然保持原来的 全/偏序→9.20@p1951 关系.

通俗地讲就是当有些数据因为本身很大或者类型不支持，自身无法作为数组的下标来方便地处理，而影响最终结果的只有元素之间的相对大小关系时，我们可以将原来的数据按照排名来处理问题，即离散化.

用来离散化的可以是大整数、浮点数、字符串等等.

实现

将一个数组离散化，并进行查询是比较常用的应用场景.

方法一

通常原数组中会有重复的元素，一般把相同的元素离散化为相同的数据.

方法如下:

1. 创建原数组的副本.
2. 将副本中的值从小到大排序.
3. 将排序好的副本去重.
4. 查找原数组的每一个元素在副本中的位置，位置即为排名，将其作为离散化后的值.

```
// arr[i] 为初始数组,下标范围为 [1, n]

for (int i = 1; i <= n; ++i)  // step 1
  tmp[i] = arr[i];
std::sort(tmp + 1, tmp + n + 1);                          // step 2
int len = std::unique(tmp + 1, tmp + n + 1) - (tmp + 1);  // step 3
for (int i = 1; i <= n; ++i)                              // step 4
  arr[i] = std::lower_bound(tmp + 1, tmp + len + 1, arr[i]) - tmp;
```

参考实现中使用的 STL 算法可参考 STL 算法→4.3.3@p386.

同样地，我们也可以对 std::vector→4.3.2.3@p373 进行离散化:

```
// std::vector<int> arr;
std::vector<int> tmp(arr);  // tmp 是 arr 的一个副本
std::sort(tmp.begin(), tmp.end());
tmp.erase(std::unique(tmp.begin(), tmp.end()), tmp.end());
for (int i = 0; i < n; ++i)
  arr[i] = std::lower_bound(tmp.begin(), tmp.end(), arr[i]) - tmp.begin();
```

方法二

根据题目要求，有时候会把相同的元素根据输入顺序离散化为不同的数据.

此时再用 `std::lower_bound()` 函数实现就有些困难了，需要换一种思路:

1. 创建原数组的副本，同时记录每个元素出现的位置.
2. 将副本按值从小到大排序，当值相同时，按出现顺序从小到大排序.
3. 将离散化后的数字放回原数组.

```
struct Data {
  int idx, val;

  bool operator<(const Data& o) const {
    if (val == o.val)
      return idx < o.idx;  // 当值相同时，先出现的元素离散化后的值更小
    return val < o.val;
  }
} tmp[MAXN];  // 也可以使用 std::pair

for (int i = 1; i <= n; ++i) tmp[i] = Data{i, arr[i]};
std::sort(tmp + 1, tmp + n + 1);
for (int i = 1; i <= n; ++i) arr[tmp[i].idx] = i;
```

复杂度

对于方法一，去重复杂度为 $O(n)$，排序复杂度为 $O(n \log n)$，最后的 n 次查找复杂度为 $O(n \log n)$.

对于方法二，排序复杂度为 $O(n \log n)$.

故两种方法的总时间复杂度都为 $O(n \log n)$.

空间复杂度为 $O(n)$.

习题

- [HAOI2014] 贴海报[1]
- [NOI2015] 程序自动分析[2]

外部参考资料

[1] [HAOI2014] 贴海报  [2] [NOI2015] 程序自动分析 

13.3 双指针

本页面将简要介绍双指针.

引入

双指针是一种简单而又灵活的技巧和思想，单独使用可以轻松解决一些特定问题，和其他算法结合也能发挥多样的用处.

双指针顾名思义，就是同时使用两个指针，在序列、链表结构上指向的是位置，在树、图结构中指向的是节点，通过或同向移动，或相向移动来维护、统计信息.

接下来我们来看双指针的几个具体使用方法.

维护区间信息

如果不和其他数据结构结合使用，双指针维护区间信息的最简单模式就是维护具有一定单调性，新增和删去一个元素都很方便处理的信息，就比如正数的和、正整数的积等等.

例题 1

例题 1 leetcode 713. 乘积小于 K 的子数组[1]

给定一个长度为 n 的正整数数组 $nums$ 和整数 k，找出该数组内乘积小于 k 的连续子数组的个数.

其中，$1 \le n \le 3 \times 10^4, 1 \le nums[i] \le 1000, 0 \le k \le 10^6$.

过程

设两个指针分别为 l, r，另外设置一个变量 tmp 记录 $[l, r]$ 内所有数的乘积. 最开始 l, r 都在最左面，先向右移动 r，直到第一次发现 $tmp \ge k$，这时就固定 r，右移 l，直到 $tmp < k$. 那么对于每个 r，l 是它能延展到的左边界，由于正整数乘积的单调性，此时以 r 为右端点的满足题目条件的区间个数为 $r - l + 1$ 个.

实现

```cpp
int numSubarrayProductLessThanK(vector<int>& nums, int k) {
  long long ji = 1ll, ans = 0;
  int l = 0;
  for (int i = 0; i < nums.size(); ++i) {
    ji *= nums[i];
    while (l <= i && ji >= k) ji /= nums[l++];
    ans += i - l + 1;
  }
  return ans;
}
```

使用双指针维护区间信息也可以与其他数据结构比如差分、单调队列、线段树、主席树等等结合使用. 另外将双指针技巧融入算法的还有莫队，莫队中将询问离线排序后，一般也都是用两个指针记录当前要处理的区间，随着指针一步步移动逐渐更新区间信息.

例题 2

接下来看一道在树上使用双指针并结合树上差分的例题：

例题 2 luogu P3066 Running Away From the Barn G[2]

给定一颗 n 个点的有根树，边有边权，节点从 1 至 n 编号，1 号节点是这棵树的根. 再给出一个参数 t，对于树上的每个节点 u，请求出 u 的子树中有多少节点满足该节点到 u 的距离不大于 t. 数据范围：$1 \le n \le 2 \times 10^5, 1 \le t \le 10^{18}, 1 \le p_i < i, 1 \le w_i \le 10^{12}$

过程

从根开始用 dfs 遍历整棵树，使用一个栈来记录根到当前节点的树链，设一个指针 u 指向当前节点，另一个指针 p 指向与 u 距离不大于 t 的节点中深度最小的节点. 记录到根的距离，每次二分查找确定 p. 此时 u 对 p 到 u 路径上的所有节点都有一个贡献，可以用树上差分来记录.

注意不能直接暴力移动 p，否则时间复杂度可能会退化至 $O(n^2)$.

习题

leetcode 1438. 绝对差不超过限制的最长连续子数组[3]

子序列匹配

例题 3 leetcode 524. 通过删除字母匹配到字典里最长单词[4]

给定一个字符串 s 和一个字符串数组 $dictionary$ 作为字典，找出并返回字典中最长的字符串，该字符串可以通过删除 s 中的某些字符得到.

过程

此类问题需要将字符串 s 与 t 进行匹配，判断 t 是否为 s 的子序列. 解决这种问题只需先将两个指针一个 i 放在 s 开始位置，一个 j 放在 t 开始位置，如果 $s[i] = t[j]$ 说明 t 的第 j 位已经在 s 中找到了第一个对应，可以进而检测后面的部分了，那么 i 和 j 同时加一. 如果上述等式不成立，则 t 的第 j 位仍然没有被匹配上，所以只给 i 加一，在 s 的后面部分再继续寻找. 最后，如果 j 已经移到了超尾位置，说明整个字符串都可以被匹配上，也就是 t 是 s 的一个子序列，否则不是.

实现

```
string findLongestWord(string s, vector<string>& dictionary) {
  sort(dictionary.begin(), dictionary.end());
  int mx = 0, r = 0;
  string ans = "";
  for (int i = dictionary.size() - 1; i >= 0; i--) {
    r = 0;
    for (int j = 0; j < s.length(); ++j) {
      if (s[j] == dictionary[i][r]) r++;
    }
    if (r == dictionary[i].length()) {
      if (r >= mx) {
        mx = r;
        ans = dictionary[i];
      }
    }
  }
  return ans;
}
```

这种两个指针指向不同对象然后逐步进行比对的方法还可以用在一些 dp 中.

利用序列有序性

很多时候在序列上使用双指针之所以能够正确地达到目的，是因为序列的某些性质，最常见的就是利用序列的有序性.

例题 4 leetcode 167. 两数之和 II - 输入有序数组[5]

给定一个已按照 **升序排列** 的整数数组 `numbers`，请你从数组中找出两个数满足相加之和等于目标数 `target`.

过程

这种问题也是双指针的经典应用了，虽然二分也很方便，但时间复杂度上多一个 $\log n$，而且代码不够简洁.

接下来介绍双指针做法：既然要找到两个数，且这两个数不能在同一位置，那其位置一定是一左一右. 由于两数之和固定，那么两数之中的小数越大，大数越小. 考虑到这些性质，那我们不妨从两边接近它们.

首先假定答案就是 1 和 n，如果发现 $num[1]+num[n]>target$，说明我们需要将其中的一个元素变小，而 $num[1]$ 已经不能再变小了，所以我们把指向 n 的指针减一，让大数变小.

同理如果发现 $num[1]+num[n]<target$，说明我们要将其中的一个元素变大，但 $num[n]$ 已经不能再变大了，所以将指向 1 的指针加一，让小数变大.

推广到一般情形，如果此时我们两个指针分别指在 l,r 上，且 $l<r$, 如果 $num[l]+num[r]>target$，就将 r 减一，如果 $num[l]+num[r]<target$，就将 l 加一. 这样 l 不断右移，r 不断左移，最后两者各逼近到一个答案.

实现

```cpp
vector<int> twoSum(vector<int>& numbers, int target) {
  int r = numbers.size() - 1, l = 0;
  vector<int> ans;
  ans.clear();
  while (l < r) {
    if (numbers[l] + numbers[r] > target)
      r--;
    else if (numbers[l] + numbers[r] == target) {
      ans.push_back(l + 1), ans.push_back(r + 1);
      return ans;
    } else
      l++;
  }
  return ans;
}
```

在归并排序中，在 $O(n+m)$ 时间内合并两个有序数组，也是保证数组的有序性条件下使用的双指针法.

习题

leetcode 15. 三数之和[6]

在单向链表中找环

过程

在单向链表中找环也是有多种办法, 不过快慢双指针方法是其中最为简洁的方法之一, 接下来介绍这种方法.

首先两个指针都指向链表的头部, 令一个指针一次走一步, 另一个指针一次走两步, 如果它们相遇了, 证明有环，否则无环，时间复杂度 $O(n)$.

如果有环的话，怎么找到环的起点呢?

我们列出式子来观察一下，设相遇时，慢指针一共走了 k 步，在环上走了 l 步（快慢指针在环上相遇时，慢指针一定没走完一圈）. 快指针走了 $2k$ 步，设环长为 C，则有

$$2k=n\times C+l+(k-l)$$

$$k=n\times C$$

第一次相遇时 n 取最小正整数 1. 也就是说 $k=C$. 那么利用这个等式，可以在两个指针相遇后，将其中一个指针移到表头，让两者都一步一步走，再度相遇的位置即为环的起点.

实现

```cpp
pair<bool, vector<int>> findCycle(
```

```
    vector<int>
        nxt)  // nxt[i]表示i在单向链表中指向的节点（0-indexed），-1表示没有指向任何节点
{
  int fast = 0, slow = 0;
  do  // 先判断有没有环
  {
    if (nxt[fast] == -1 || nxt[nxt[fast]] == -1) {
      return make_pair(false, vector<int>());  // 没有环
    }
    fast = nxt[nxt[fast]];
    slow = nxt[slow];
  } while (fast != slow);
  slow = 0;
  while (slow != fast)  // 再找入环位置
  {
    slow = nxt[slow];
    fast = nxt[fast];
  }
  vector<int> cycle;
  do  // 最后找出整个环
  {
    cycle.push_back(slow);
    slow = nxt[slow];
  } while (slow != fast);
  return make_pair(true, cycle);
}

```

时间复杂度 $O(n)$.

外部参考资料

[1] leetcode 713. 乘积小于 K 的子数组

[2] luogu P3066 Running Away From the Barn G

[3] leetcode 1438. 绝对差不超过限制的最长连续子数组

[4] leetcode 524. 通过删除字母匹配到字典里最长单词

[5] leetcode 167. 两数之和 II - 输入有序数组

[6] leetcode 15. 三数之和

13.4 离线算法

13.4.1 离线算法简介

本章将介绍离线算法（Offline Algorithm）的思想、常见算法及优化.

离线算法是基于「**求解前已知所有数据**」这一假设来设计的，适用于有多组询问的题目. 相对的还有在线算法[1]（Online Algorithm）.

例如 选择排序→5.7.2@p568 必须知道数组的全局最小元素才能执行，所以是离线算法，而 插入排序→5.7.4@p572 可以动态接收数据进行排序，不强制要求执行前已知全部数据，所以是在线算法.

对于相同的问题，在设计难度等方面，离线算法往往优于在线算法. 为了阻止选手使用离线算法，有时题目会使用「强制在线」的方式，常见的有需要前一个询问的答案才能得到下一个询问的参数（交互题→2.3.1@p76 与 通信题→2.3.1@p76 也属于此类）.

离线算法的常见思路包括将询问统一求解（如 CDQ 分治→13.4.2@p3176）、通过一个询问的答案求出另外相似询问的答案（如 整体二分→13.4.3@p3199 和 莫队算法→13.4.4.1@p3209）等.

由于离线算法是一种思想而并不是某种具体的算法，因此它会搭配各种各样的数据结构或算法一起使用，与之相关的题目种类也更为繁杂.

外部参考资料

[1] 在线算法 

13.4.2 CDQ 分治

本页面将介绍 CDQ 分治.

简介

CDQ 分治是一种思想而不是具体的算法，与 动态规划→7.1@p714 类似. 目前这个思想的拓展十分广泛，依原理与写法的不同，大致分为三类：

- 解决和点对有关的问题.
- 1D 动态规划的优化与转移.
- 通过 CDQ 分治，将一些动态问题转化为静态问题.

CDQ 分治的思想最早由 IOI2008 金牌得主陈丹琦在高中时整理并总结，它也因此得名. [1]

解决和点对有关的问题

这类问题多数类似于「给定一个长度为 n 的序列，统计有一些特性的点对 (i,j) 的数量」或「给定一个长度为 n 的序列，找到一对点 (i,j) 使得一些函数的值最大」.

CDQ 分治解决这类问题的算法流程如下：

1. 找到这个序列的中点 mid；
2. 将所有点对 (i,j) 划分为 3 类：
    1. $1\le i\le mid, 1\le j\le mid$ 的点对；
    2. $1\le i\le mid, mid+1\le j\le n$ 的点对；
    3. $mid+1\le i\le n, mid+1\le j\le n$ 的点对.
3. 将 $(1,n)$ 这个序列拆成两个序列 $(1,mid)$ 和 $(mid+1,n)$. 此时第一类点对和第三类点对都在这两个序列之中；
4. 递归地处理这两类点对；
5. 设法处理第二类点对.

可以看到 CDQ 分治的思想就是不断地把点对通过递归的方式分给左右两个区间.

在实际应用时，我们通常使用一个函数 `solve(l,r)` 处理 $l\le i\le r, l\le j\le r$ 的点对. 上述算法流程中的递归部分便是通过 `solve(l,mid)` 与 `solve(mid+1,r)` 来实现的. 剩下的第二类点对则需要额外设计算法解决.

例题

三维偏序[2]

给定一个序列，每个点有 a_i,b_i,c_i 三个属性，试求：这个序列里有多少对点对 (i,j) 满足 $a_j\le a_i$ 且 $b_j\le b_i$ 且 $c_j\le c_i$ 且 $j\ne i$.

[1] 从《Cash》谈一类分治算法的应用[1]

解题思路

三维偏序是 CDQ 分治的经典问题.

题目要求统计序列里点对的个数，那试一下用 CDQ 分治.

首先将序列按 a 排序.

假设我们现在写好了 `solve(l,r)`，并且通过递归搞定了 `solve(l,mid)` 和 `solve(mid+1,r)`. 现在我们要做的，就是统计满足 $l \le i \le mid$，$mid+1 \le j \le r$ 的点对 (i,j) 中，有多个点对还满足 $a_i \le a_j$，$b_i \le b_j$，$c_i \le c_j$ 的限制条件.

稍微思考一下就会发现，那个 $a_i \le a_j$ 的限制条件没啥用了：既然 i 比 mid 小，j 比 mid 大，那 i 肯定比 j 要小；已经将序列按 a 排序，就一定有 $a_i \le a_j$. 现在还剩下两个限制条件：$b_i \le b_j$ 与 $c_i \le c_j$. 根据这个限制条件我们就可以枚举 j, 求出有多少个满足条件的 i.

为了方便枚举，我们把 (l, mid) 和 $(mid+1, r)$ 中的点全部按照 b 的值从小到大排个序. 之后我们依次枚举每一个 j, 把所有 $b_i \le b_j$ 的点 i 全部插入到某种数据结构里（这里我们选择 树状数组→10.12@p2078）. 此时只要查询树状数组里有多少个点的 c 值是小于等于 c_j 的，我们就求出了对于这个点 j，有多少个 i 可以合法匹配它了.

当我们插入一个 c 值等于 x 的点时，我们就令树状数组的 x 这个位置单点加一，而查询树状数组里有多少个点小于 x 的操作实际上就是在求 前缀和→5.8@p617，只要我们事先对于所有的 c 值做了离散化→13.2@p3170，我们的复杂度就是对的.

对于每一个 j，我们都需要将所有 $b_i \le b_j$ 的点 i 插入树状数组中. 由于所有的 i 和 j 都已事先按照 b 值排好序，这样的话只要以双指针的方式在树状数组里插入点，则对树状数组的插入操作就能从 $O(n^2)$ 次降到 $O(n)$ 次.

通过这样一个算法流程，我们就用 $O(n \log n)$ 的时间处理完了关于第二类点对的信息了. 此时算法的时间复杂度是 $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n \log n) = O(n \log^2 n)$.

示例代码

```cpp
#include <algorithm>
#include <iostream>

constexpr int MAXN = 1e5 + 10;
constexpr int MAXK = 2e5 + 10;

int n, k;

struct Element {
  int a, b, c;
  int cnt;
  int res;

  bool operator!=(Element other) const {
    if (a != other.a) return true;
    if (b != other.b) return true;
    if (c != other.c) return true;
    return false;
  }
};
```

```
21
22 Element e[MAXN];
23 Element ue[MAXN];
24 int m, t;
25 int res[MAXN];
26
27 struct BinaryIndexedTree {
28   int node[MAXK];
29
30   int lowbit(int x) { return x & -x; }
31
32   void Add(int pos, int val) {
33     while (pos <= k) {
34       node[pos] += val;
35       pos += lowbit(pos);
36     }
37     return;
38   }
39
40   int Ask(int pos) {
41     int res = 0;
42     while (pos) {
43       res += node[pos];
44       pos -= lowbit(pos);
45     }
46     return res;
47   }
48 } BIT;
49
50 bool cmpA(Element x, Element y) {
51   if (x.a != y.a) return x.a < y.a;
52   if (x.b != y.b) return x.b < y.b;
53   return x.c < y.c;
54 }
55
56 bool cmpB(Element x, Element y) {
57   if (x.b != y.b) return x.b < y.b;
58   return x.c < y.c;
59 }
60
61 void CDQ(int l, int r) {
62   if (l == r) return;
63   int mid = (l + r) / 2;
64   CDQ(l, mid);
65   CDQ(mid + 1, r);
66   std::sort(ue + l, ue + mid + 1, cmpB);
67   std::sort(ue + mid + 1, ue + r + 1, cmpB);
68   int i = l;
69   int j = mid + 1;
70   while (j <= r) {
71     while (i <= mid && ue[i].b <= ue[j].b) {
72       BIT.Add(ue[i].c, ue[i].cnt);
```

```
      i++;
    }
    ue[j].res += BIT.Ask(ue[j].c);
    j++;
  }
  for (int k = l; k < i; k++) BIT.Add(ue[k].c, -ue[k].cnt);
  return;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> k;
  for (int i = 1; i <= n; i++) cin >> e[i].a >> e[i].b >> e[i].c;
  std::sort(e + 1, e + n + 1, cmpA);
  for (int i = 1; i <= n; i++) {
    t++;
    if (e[i] != e[i + 1]) {
      m++;
      ue[m].a = e[i].a;
      ue[m].b = e[i].b;
      ue[m].c = e[i].c;
      ue[m].cnt = t;
      t = 0;
    }
  }
  CDQ(1, m);
  for (int i = 1; i <= m; i++) res[ue[i].res + ue[i].cnt - 1] += ue[i].cnt;
  for (int i = 0; i < n; i++) cout << res[i] << '\n';
  return 0;
}

```

CQOI2011 动态逆序对[3]

对于序列 a，它的逆序对数定义为集合 $\{(i,j)|\ i<j\wedge a_i>a_j\}$ 中的元素个数.

现在给出 $1\sim n$ 的一个排列，按照某种顺序依次删除 m 个元素，你的任务是在每次删除一个元素之前统计整个序列的逆序对数.

示例代码

```
// 仔细推一下就是和三维偏序差不多的式子了，基本就是一个三维偏序的板子
#include <algorithm>
#include <iostream>
using namespace std;
using ll = long long;
int n;
int m;
```

```
 8
 9 struct treearray {
10   int ta[200010];
11
12   void ub(int& x) { x += x & (-x); }
13
14   void db(int& x) { x -= x & (-x); }
15
16   void c(int x, int t) {
17     for (; x <= n + 1; ub(x)) ta[x] += t;
18   }
19
20   int sum(int x) {
21     int r = 0;
22     for (; x > 0; db(x)) r += ta[x];
23     return r;
24   }
25 } ta;
26
27 struct data_ {
28   int val;
29   int del;
30   int ans;
31 } a[100010];
32
33 int rv[100010];
34 ll res;
35
36 // 重写两个比较
37 bool cmp1(const data_& a, const data_& b) { return a.val < b.val; }
38
39 bool cmp2(const data_& a, const data_& b) { return a.del < b.del; }
40
41 void solve(int l, int r) {  // 底下是具体的式子，套用
42   if (r - l == 1) {
43     return;
44   }
45   int mid = (l + r) / 2;
46   solve(l, mid);
47   solve(mid, r);
48   int i = l + 1;
49   int j = mid + 1;
50   while (i <= mid) {
51     while (a[i].val > a[j].val && j <= r) {
52       ta.c(a[j].del, 1);
53       j++;
54     }
55     a[i].ans += ta.sum(m + 1) - ta.sum(a[i].del);
56     i++;
57   }
58   i = l + 1;
59   j = mid + 1;
```

```
  while (i <= mid) {
    while (a[i].val > a[j].val && j <= r) {
      ta.c(a[j].del, -1);
      j++;
    }
    i++;
  }
  i = mid;
  j = r;
  while (j > mid) {
    while (a[j].val < a[i].val && i > l) {
      ta.c(a[i].del, 1);
      i--;
    }
    a[j].ans += ta.sum(m + 1) - ta.sum(a[j].del);
    j--;
  }
  i = mid;
  j = r;
  while (j > mid) {
    while (a[j].val < a[i].val && i > l) {
      ta.c(a[i].del, -1);
      i--;
    }
    j--;
  }
  sort(a + l + 1, a + r + 1, cmp1);
  return;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    cin >> a[i].val;
    rv[a[i].val] = i;
  }
  for (int i = 1; i <= m; i++) {
    int p;
    cin >> p;
    a[rv[p]].del = i;
  }
  for (int i = 1; i <= n; i++) {
    if (a[i].del == 0) a[i].del = m + 1;
  }
  for (int i = 1; i <= n; i++) {
    res += ta.sum(n + 1) - ta.sum(a[i].val);
    ta.c(a[i].val, 1);
  }
  for (int i = 1; i <= n; i++) {
    ta.c(a[i].val, -1);
  }
```

```
112   solve(0, n);
113   sort(a + 1, a + n + 1, cmp2);
114   for (int i = 1; i <= m; i++) {
115     cout << res << '\n';
116     res -= a[i].ans;
117   }
118   return 0;
119 }
120
```

CDQ 分治优化 1D/1D 动态规划的转移

相关内容：CDQ 分治优化 DP→7.15.1@p810

1D/1D 动态规划指的是一类特定的 DP 问题，该类题目的特征是 DP 数组是一维的，转移是 $O(n)$ 的. 如果条件良好的话，有时可以通过 CDQ 分治来把它们的时间复杂度由 $O(n^2)$ 降至 $O(n\log^2 n)$.

例如, 给定一个序列, 每个元素有两个属性 a, b. 我们希望计算一个 DP 式子的值, 它的转移方程如下:

$dp_i = 1 + \max_{j=1}^{i-1} dp_j[a_j < a_i][b_j < b_i]$

这是一个二维最长上升子序列的 DP 方程, 即只有 $j < i, a_j < a_i, b_j < b_i$ 的点 j 可以更新点 i 的 DP 值.

直接转移显然是 $O(n^2)$ 的. 以下是使用 CDQ 分治优化转移过程的讲解.

我们发现 dp_j 转移到 dp_i 这种转移关系也是一种点对间的关系，所以我们用类似 CDQ 分治处理点对关系的方式来处理它.

这个转移过程相对来讲比较套路. 假设现在正在处理的区间是 (l, r)，算法流程大致如下:

1. 如果 $l = r$，说明 dp_r 值的 max 部分已经被计算好了，直接令 $dp_r \leftarrow dp_r + 1$ 然后返回即可;
2. 递归使用 `solve(l,mid)`;
3. 处理所有 $l \le j \le mid$，$mid + 1 \le i \le r$ 的转移关系;
4. 递归使用 `solve(mid+1,r)`.

第三步的做法与 CDQ 分治求三维偏序差不多. 处理 $l \le j \le mid$，$mid + 1 \le i \le r$ 的转移关系的时候，我们会发现已经不用管 $j < i$ 这个限制条件了. 因此，我们依然先将所有的点 i 和点 j 按 a 值进行排序处理，然后用双指针的方式将 j 点插入到树状数组里，最后查一下前缀最大值更新一下 dp_i 就可以了.

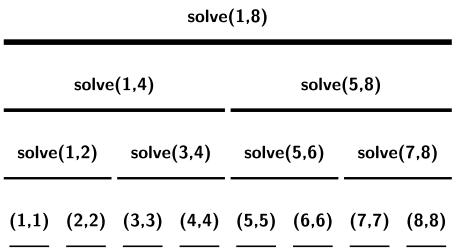
转移过程的正确性证明

该 CDQ 写法和处理点对间关系的 CDQ 写法最大的不同就是处理 $l \le j \le mid$，$mid + 1 \le i \le r$ 的点对这一部分. 处理点对间关系的 CDQ 写法中，这一部分放到哪里都是可以的. 但是，在用 CDQ 分治优化 DP 的时候，这个流程却必须夹在 $solve(l, mid)$,$solve(mid + 1, r)$ 的中间. 原因是 DP 的转移是 **有序的**，它必须满足两个条件，否则就是不对的:

1. 用来计算 dp_i 的所有 dp_j 值都必须是已经计算完毕的，不能存在「半成品」;
2. 用来计算 dp_i 的所有 dp_j 值都必须能更新到 dp_i，不能存在没有更新到的 dp_j 值.

上述两个条件可能在 $O(n^2)$ 暴力的时候是相当容易满足的，但是使用 CDQ 分治后，转移顺序很显然已经乱掉了，所以有必要考察转移的正确性.

CDQ 分治的递归树如下所示.



执行刚才的算法流程的话，以 8 这个点为例，它的 DP 值是在 `solve(1,8)`、`solve(5,8)`、`solve(7,8)` 这 3 个函数中更新完成的，而三次用来更新它的点分别是 $(1,4)$、$(5,6)$、$(7,7)$ 这三个不相交的区间；又以 5 这个点为例，它的 DP 值是在 `solve(1,4)` 函数中解决的，更新它的区间是 $(1,4)$．仔细观察就会发现，一个 i 点的 DP 值被更新了 log 次，而且，更新它的区间刚好是 $(1,i)$ 在线段树上被拆分出来的 log 个区间．因此，我们的确保证了所有合法的 j 都更新过点 i，满足第 2 个条件．

接着分析我们算法的执行流程：

1. 第一个结束的函数是 `solve(1,1)`．此时我们发现 dp_1 的值已经计算完毕了；
2. 第一个执行转移过程的函数是 `solve(1,2)`．此时我们发现 dp_2 的值已经被转移好了；
3. 第二个结束的函数是 `solve(2,2)`．此时我们发现 dp_2 的值已经计算完毕了；
4. 接下来 `solve(1,2)` 结束，$(1,2)$ 这段区间的 dp 值均被计算好；
5. 下一个执行转移流程的函数是 `solve(1,4)`．这次转移结束之后我们发现 dp_3 的值已经被转移好了；
6. 接下来结束的函数是 `solve(3,3)`．我们会发现 dp_3 的 dp 值被计算好了；
7. 接下来执行的转移是 `solve(3,4)`．此时 dp_4 在 `solve(1,4)` 中被 $(1,2)$ 转移了一次，这次又被 $(3,3)$ 转移了，因此 dp_4 的值也被转移好了；
8. `solve(4,4)` 结束，dp_4 的值计算完毕；
9. `solve(3,4)` 结束，$(3,4)$ 的值计算完毕；
10. `solve(1,4)` 结束，$(1,4)$ 的值计算完毕．
11. ……

通过模拟函数流程，我们发现一件事：每次 `solve(l,r)` 结束的时候，(l,r) 区间的 DP 值会被全部计算好．由于我们每一次执行转移函数的时候，`solve(l,mid)` 已经结束，因此我们每一次执行的转移过程都是合法的，满足第 1 个条件．

在刚才的过程我们发现，如果将 CDQ 分治的递归树看成一颗线段树，那么 CDQ 分治就是这个线段树的 **中序遍历函数**，因此我们相当于按顺序处理了所有的 DP 值，只是转移顺序被拆开了而已，所以算法是正确的．

例题

SDOI2011 拦截导弹[4]

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统．但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度、并且能够拦截任意速度的导弹，但是以后每一发炮弹都不能高于前一发的高度，其拦截的导弹的飞行速度也不能大于前一发．某天，雷达捕捉到敌国的导弹来袭．由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹．

在不能拦截所有的导弹的情况下，我们当然要选择使国家损失最小、也就是拦截导弹的数量最多的方案．但是拦截导弹数量的最多的方案有可能有多个，如果有多个最优方案，那么我们会随机选取一个作为最终的拦截导弹行动蓝图．

我方间谍已经获取了所有敌军导弹的高度和速度，你的任务是计算出在执行上述决策时，每枚导弹被拦截掉的概率.

示例代码

```cpp
// 一道二维最长上升子序列的题
// 为了确定某一个元素是否在最长上升子序列中可以正反跑两遍 CDQ
#include <algorithm>
#include <iomanip>
#include <iostream>
using namespace std;
using db = double;
constexpr int N = 1e6 + 10;

struct data_ {
  int h;
  int v;
  int p;
  int ma;
  db ca;
} a[2][N];

int n;
bool tr;

// 底下是重写比较
bool cmp1(const data_& a, const data_& b) {
  if (tr)
    return a.h > b.h;
  else
    return a.h < b.h;
}

bool cmp2(const data_& a, const data_& b) {
  if (tr)
    return a.v > b.v;
  else
    return a.v < b.v;
}

bool cmp3(const data_& a, const data_& b) {
  if (tr)
    return a.p < b.p;
  else
    return a.p > b.p;
}

bool cmp4(const data_& a, const data_& b) { return a.v == b.v; }

struct treearray {
  int ma[2 * N];
  db ca[2 * N];
```

```
48
49    void c(int x, int t, db c) {
50      for (; x <= n; x += x & (-x)) {
51        if (ma[x] == t) {
52          ca[x] += c;
53        } else if (ma[x] < t) {
54          ca[x] = c;
55          ma[x] = t;
56        }
57      }
58    }
59
60    void d(int x) {
61      for (; x <= n; x += x & (-x)) {
62        ma[x] = 0;
63        ca[x] = 0;
64      }
65    }
66
67    void q(int x, int& m, db& c) {
68      for (; x > 0; x -= x & (-x)) {
69        if (ma[x] == m) {
70          c += ca[x];
71        } else if (m < ma[x]) {
72          c = ca[x];
73          m = ma[x];
74        }
75      }
76    }
77  } ta;
78
79  int rk[2][N];
80
81  void solve(int l, int r, int t) {  // 递归跑
82    if (r - l == 1) {
83      return;
84    }
85    int mid = (l + r) / 2;
86    solve(l, mid, t);
87    sort(a[t] + mid + 1, a[t] + r + 1, cmp1);
88    int p = l + 1;
89    for (int i = mid + 1; i <= r; i++) {
90      for (; (cmp1(a[t][p], a[t][i]) || a[t][p].h == a[t][i].h) && p <= mid;
91           p++) {
92        ta.c(a[t][p].v, a[t][p].ma, a[t][p].ca);
93      }
94      db c = 0;
95      int m = 0;
96      ta.q(a[t][i].v, m, c);
97      if (a[t][i].ma < m + 1) {
98        a[t][i].ma = m + 1;
99        a[t][i].ca = c;
```

```
    } else if (a[t][i].ma == m + 1) {
      a[t][i].ca += c;
    }
  }
  for (int i = l + 1; i <= mid; i++) {
    ta.d(a[t][i].v);
  }
  sort(a[t] + mid, a[t] + r + 1, cmp3);
  solve(mid, r, t);
  sort(a[t] + l + 1, a[t] + r + 1, cmp1);
}

void ih(int t) {
  sort(a[t] + 1, a[t] + n + 1, cmp2);
  rk[t][1] = 1;
  for (int i = 2; i <= n; i++) {
    rk[t][i] = (cmp4(a[t][i], a[t][i - 1])) ? rk[t][i - 1] : i;
  }
  for (int i = 1; i <= n; i++) {
    a[t][i].v = rk[t][i];
  }
  sort(a[t] + 1, a[t] + n + 1, cmp3);
  for (int i = 1; i <= n; i++) {
    a[t][i].ma = 1;
    a[t][i].ca = 1;
  }
}

int len;
db ans;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; i++) {
    cin >> a[0][i].h >> a[0][i].v;
    a[0][i].p = i;
    a[1][i].h = a[0][i].h;
    a[1][i].v = a[0][i].v;
    a[1][i].p = i;
  }
  ih(0);
  solve(0, n, 0);
  tr = true;
  ih(1);
  solve(0, n, 1);
  tr = true;
  sort(a[0] + 1, a[0] + n + 1, cmp3);
  sort(a[1] + 1, a[1] + n + 1, cmp3);
  for (int i = 1; i <= n; i++) {
    len = max(len, a[0][i].ma);
  }
```

```
152   cout << len << '\n';
153   for (int i = 1; i <= n; i++) {
154     if (a[0][i].ma == len) {
155       ans += a[0][i].ca;
156     }
157   }
158   cout << fixed << setprecision(5);
159   for (int i = 1; i <= n; i++) {
160     if (a[0][i].ma + a[1][i].ma - 1 == len) {
161       cout << (a[0][i].ca * a[1][i].ca) / ans << ' ';
162     } else {
163       cout << "0.00000 ";
164     }
165   }
166   return 0;
167 }
168
```

将动态问题转化为静态问题

前两种情况使用 CDQ 分治的目的是将序列折半之后递归处理点对间的关系，来获得良好的复杂度. 不过在本节中，折半的不是一般的序列，而是时间序列.

它适用于一些「需要支持做 xxx 修改然后做 xxx 询问」的数据结构题. 该类题目有两个特点：

- 如果把询问 离线→13.4.1@p3175，所有操作会按照时间自然地排成一个序列.
- 每一个修改均与之后的询问操作息息相关. 而这样的「修改 - 询问」关系一共会有 $O(n^2)$ 对.

我们可以使用 CDQ 分治对于这个操作序列进行分治，处理修改和询问之间的关系.

与处理点对关系的 CDQ 分治类似，假设正在分治的序列是 (l,r)，我们先递归地处理 (l,mid) 和 (mid,r) 之间的修改 - 询问关系，再处理所有 $l \le i \le mid$，$mid+1 \le j \le r$ 的修改 - 询问关系，其中 i 是一个修改，j 是一个询问.

注意，如果各个修改之间是 **独立** 的话，我们无需处理 $l \le i \le mid$ 和 $mid+1 \le j \le r$，以及 `solve(l,mid)` 和 `solve(mid+1,r)` 之间的时序关系（比如普通的加减法问题）. 但是如果各个修改之间并不独立（比如说赋值操作），做完这个修改后，序列长什么样可能依赖于之前的序列. 此时处理所有跨越 mid 的修改 - 询问关系的步骤就必须放在 `solve(l,mid)` 和 `solve(mid+1,r)` 之间. 理由和 CDQ 分治优化 1D/1D 动态规划的原因是一样的：按照中序遍历序进行分治才能保证每一个修改都是严格按照时间顺序执行的.

例题

矩形加矩形求和

维护一个二维数组，支持在一个矩形区域内加一个数字，每次询问一个矩形区域的和.

解题思路

对于这个问题的无修版本，即「给定一个二维数组，多次询问一个矩形区域的和」，有一个扫描线配合线段树的经典做法. 具体的做法是先将每个矩形拆成插入和删除两个操作，接着将每个询问拆成二维前缀和相减的形式，最后离线. 然而，原题目是带修改的，不能直接使用这种做法.

尝试对其使用 CDQ 分治. 我们将所有的询问和修改操作全部离线. 这些操作形成了一个序列, 并且有 $O(N^2)$ 对修改 - 询问的关系. 依然使用 CDQ 分治的一般流程，将所有的关系分成三类，在这一层分治过程当中只处理跨越 mid 的修改 - 询问关系，剩下的修改 - 询问关系通过递归的方式来解决.

我们发现，所有的修改在询问之前就已完成. 这时，原问题等价于「平面上有静态的一些矩形，不停地询问一个矩形区域的和」.

使用一个扫描线在 $O(n \log n)$ 的时间内处理好所有跨越 mid 的修改 - 询问关系，剩下的事情就是递归地分治左右两侧的修改 - 询问关系了.

在这样实现的 CDQ 分治中，同一个询问被处理了 $O(\log n)$ 次. 不过没有关系，因为每次贡献这个询问的修改是互不相交的. 全套流程的时间复杂度为 $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n \log n) = O(n \log^2 n)$.

观察上述的算法流程, 我们发现一开始我们只能解决静态的矩形加矩形求和问题, 但只是简单地使用 CDQ 分治后，我们就可以离线地解决一个动态的矩形加矩形求和问题了. 将动态问题转化为静态问题的精髓就在于 CDQ 分治每次仅仅处理跨越某一个点的修改和询问关系，这样的话我们就只需要考虑「所有询问都在修改之后」这个简单的问题了. 也正是因为这一点，CDQ 分治被称为「动态问题转化为静态问题的工具」.

[Ynoi2016] 镜中的昆虫[5]

维护一个长为 n 的序列 a_i，有 m 次操作.

1. 将区间 $[l, r]$ 的值修改为 x;
2. 询问区间 $[l, r]$ 出现了多少种不同的数，也就是说同一个数出现多次只算一个.

一句话题意：区间赋值区间数颜色.

解题思路

维护一下每个位置左侧第一个同色点的位置, 记为 pre_i, 此时区间数颜色就被转化为了一个经典的二维数点问题.

通过将连续的一段颜色看成一个点的方式，可以证明 pre 的变化量是 $O(n + m)$ 的，即单次操作仅仅引起 $O(1)$ 的 pre 值变化，那么我们可以用 CDQ 分治来解决动态的单点加矩形求和问题.

pre 数组的具体变化可以使用 `std::set` 来进行处理. 这个用 set 维护连续的区间的技巧也被称为 old driver tree→13.18@p3332.

示例代码

```cpp
#include <algorithm>
#include <iostream>
#include <map>
#include <set>
#define SNI set<nod>::iterator
#define SDI set<data>::iterator
using namespace std;
constexpr int N = 1e5 + 10;
int n;
int m;
int pre[N];
```

```
int npre[N];
int a[N];
int tp[N];
int lf[N];
int rt[N];
int co[N];

struct modi {
  int t;
  int pos;
  int pre;
  int va;

  friend bool operator<(modi a, modi b) { return a.pre < b.pre; }
} md[10 * N];

int tp1;

struct qry {
  int t;
  int l;
  int r;
  int ans;

  friend bool operator<(qry a, qry b) { return a.l < b.l; }
} qr[N];

int tp2;
int cnt;

bool cmp(const qry& a, const qry& b) { return a.t < b.t; }

void modify(int pos, int co)  // 修改函数
{
  if (npre[pos] == co) return;
  md[++tp1] = modi{++cnt, pos, npre[pos], -1};
  md[++tp1] = modi{++cnt, pos, npre[pos] = co, 1};
}

namespace prew {
int lst[2 * N];
map<int, int> mp;  // 提前离散化

void prew() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i], mp[a[i]] = 1;
  for (int i = 1; i <= m; i++) {
    cin >> tp[i] >> lf[i] >> rt[i];
    if (tp[i] == 1) cin >> co[i], mp[co[i]] = 1;
  }
  map<int, int>::iterator it, it1;
```

```
64    for (it = mp.begin(), it1 = it, ++it1; it1 != mp.end(); ++it, ++it1)
65      it1->second += it->second;
66    for (int i = 1; i <= n; i++) a[i] = mp[a[i]];
67    for (int i = 1; i <= n; i++)
68      if (tp[i] == 1) co[i] = mp[co[i]];
69    for (int i = 1; i <= n; i++) pre[i] = lst[a[i]], lst[a[i]] = i;
70    for (int i = 1; i <= n; i++) npre[i] = pre[i];
71  }
72  }  // namespace prew
73
74  namespace colist {
75  struct data {
76    int l;
77    int r;
78    int x;
79
80    friend bool operator<(data a, data b) { return a.r < b.r; }
81  };
82
83  set<data> s;
84
85  struct nod {
86    int l;
87    int r;
88
89    friend bool operator<(nod a, nod b) { return a.r < b.r; }
90  };
91
92  set<nod> c[2 * N];
93  set<int> bd;
94
95  void split(int mid) {  // 将一个节点拆成两个节点
96    SDI it = s.lower_bound(data{0, mid, 0});
97    data p = *it;
98    if (mid == p.r) return;
99    s.erase(p);
100   s.insert(data{p.l, mid, p.x});
101   s.insert(data{mid + 1, p.r, p.x});
102   c[p.x].erase(nod{p.l, p.r});
103   c[p.x].insert(nod{p.l, mid});
104   c[p.x].insert(nod{mid + 1, p.r});
105 }
106
107 void del(set<data>::iterator it) {  // 删除一个迭代器
108   bd.insert(it->l);
109   SNI it1, it2;
110   it1 = it2 = c[it->x].find(nod{it->l, it->r});
111   ++it2;
112   if (it2 != c[it->x].end()) bd.insert(it2->l);
113   c[it->x].erase(it1);
114   s.erase(it);
115 }
```

```
116
117 void ins(data p) {  // 插入一个节点
118   s.insert(p);
119   SNI it = c[p.x].insert(nod{p.l, p.r}).first;
120   ++it;
121   if (it != c[p.x].end()) {
122     bd.insert(it->l);
123   }
124 }
125
126 void stv(int l, int r, int x) {  // 区间赋值
127   if (l != 1) split(l - 1);
128   split(r);
129   int p = l;  // split两下之后删掉所有区间
130   while (p != r + 1) {
131     SDI it = s.lower_bound(data{0, p, 0});
132     p = it->r + 1;
133     del(it);
134   }
135   ins(data{l, r, x});  // 扫一遍set处理所有变化的pre值
136   for (set<int>::iterator it = bd.begin(); it != bd.end(); ++it) {
137     SDI it1 = s.lower_bound(data{0, *it, 0});
138     if (*it != it1->l)
139       modify(*it, *it - 1);
140     else {
141       SNI it2 = c[it1->x].lower_bound(nod{0, *it});
142       if (it2 != c[it1->x].begin())
143         --it2, modify(*it, it2->r);
144       else
145         modify(*it, 0);
146     }
147   }
148   bd.clear();
149 }
150
151 void ih() {
152   int nc = a[1];
153   int ccnt = 1;  // 将连续的一段插入到set中
154   for (int i = 2; i <= n; i++)
155     if (nc != a[i]) {
156       s.insert(data{i - ccnt, i - 1, nc}), c[nc].insert(nod{i - ccnt, i
- 1});
157       nc = a[i];
158       ccnt = 1;
159     } else {
160       ccnt++;
161     }
162   s.insert(data{n - ccnt + 1, n, a[n]}), c[a[n]].insert(nod{n - ccnt + 1,
n});
163 }
164 }  // namespace colist
165
```

```
namespace CDQ {
struct treearray  // 树状数组
{
  int ta[N];

  void c(int x, int t) {
    for (; x <= n; x += x & (-x)) ta[x] += t;
  }

  void d(int x) {
    for (; x <= n; x += x & (-x)) ta[x] = 0;
  }

  int q(int x) {
    int r = 0;
    for (; x; x -= x & (-x)) r += ta[x];
    return r;
  }

  void clear() {
    for (int i = 1; i <= n; i++) ta[i] = 0;
  }
} ta;

int srt[N];

bool cmp1(const int& a, const int& b) { return pre[a] < pre[b]; }

void solve(int l1, int r1, int l2, int r2, int L, int R) {  // CDQ
  if (l1 == r1 || l2 == r2) return;
  int mid = (L + R) / 2;
  int mid1 = l1;
  while (mid1 != r1 && md[mid1 + 1].t <= mid) mid1++;
  int mid2 = l2;
  while (mid2 != r2 && qr[mid2 + 1].t <= mid) mid2++;
  solve(l1, mid1, l2, mid2, L, mid);
  solve(mid1, r1, mid2, r2, mid, R);
  if (l1 != mid1 && mid2 != r2) {
    sort(md + l1 + 1, md + mid1 + 1);
    sort(qr + mid2 + 1, qr + r2 + 1);
    for (int i = mid2 + 1, j = l1 + 1; i <= r2; i++) {  // 考虑左侧对右侧贡献
      while (j <= mid1 && md[j].pre < qr[i].l) ta.c(md[j].pos, md[j].va),
j++;
      qr[i].ans += ta.q(qr[i].r) - ta.q(qr[i].l - 1);
    }
    for (int i = l1 + 1; i <= mid1; i++) ta.d(md[i].pos);
  }
}

void mainsolve() {
  colist::ih();
  for (int i = 1; i <= m; i++)
```

```
217     if (tp[i] == 1)
218       colist::stv(lf[i], rt[i], co[i]);
219     else
220       qr[++tp2] = qry{++cnt, lf[i], rt[i], 0};
221   sort(qr + 1, qr + tp2 + 1);
222   for (int i = 1; i <= n; i++) srt[i] = i;
223   sort(srt + 1, srt + n + 1, cmp1);
224   for (int i = 1, j = 1; i <= tp2; i++) {  // 初始化一下每个询问的值
225     while (j <= n && pre[srt[j]] < qr[i].l) ta.c(srt[j], 1), j++;
226     qr[i].ans += ta.q(qr[i].r) - ta.q(qr[i].l - 1);
227   }
228   ta.clear();
229   sort(qr + 1, qr + tp2 + 1, cmp);
230   solve(0, tp1, 0, tp2, 0, cnt);
231   sort(qr + 1, qr + tp2 + 1, cmp);
232   for (int i = 1; i <= tp2; i++) cout << qr[i].ans << '\n';
233 }
234 }  // namespace CDQ
235
236 int main() {
237   prew::prew();
238   CDQ::mainsolve();
239   return 0;
240 }
241
```

[HNOI2010] 城市建设[6]

PS 国是一个拥有诸多城市的大国．国王 Louis 为城市的交通建设可谓绞尽脑汁．Louis 可以在某些城市之间修建道路，在不同的城市之间修建道路需要不同的花费．

Louis 希望建造最少的道路使得国内所有的城市连通．但是由于某些因素，城市之间修建道路需要的花费会随着时间而改变．Louis 会不断得到某道路的修建代价改变的消息．他希望每得到一条消息后能立即知道使城市连通的最小花费总和．Louis 决定求助于你来完成这个任务．

一句话题意：给定一张图支持动态的修改边权，要求在每次修改边权之后输出这张图的最小生成树的最小代价和．

解题思路

事实上，有一个线段树分治套 lct 的做法可以解决这个问题，但是这个实现方式的常数过大，可能需要精妙的卡常技巧才可以通过本题，因此不妨考虑 CDQ 分治来解决这个问题．

和一般的 CDQ 分治解决的问题不同，此时使用 CDQ 分治的时候并没有修改和询问的关系来让我们进行分治，因为无法单独考虑「修改一个边对整张图的最小生成树有什么贡献」．传统的 CDQ 分治思路似乎不是很好使．

通过刚才的例题可以发现，一般的 CDQ 分治和线段树有着特殊的联系：我们在 CDQ 分治的过程中其实隐式地建了一棵线段树出来（因为 CDQ 分治的递归树就是一颗线段树）．通常的 CDQ 是考虑线段树左右儿子之间的联系．而对于这道题，我们需要考虑的是父亲和孩子之间的关系；换句话来

讲，我们在 `$solve(l,r)$` 这段区间的时候，如果可以想办法使图的规模变成和区间长度相关的一个变量的话，就可以解决这个问题了.

那么具体来讲如何设计算法呢?

假设我们正在构造 (l,r) 这段区间的最小生成树边集，并且我们已知它父亲最小生成树的边集. 我们将在 (l,r) 这段区间中发生变化的边分别赋与 $+\infty$ 和 $-\infty$ 的边权，并各跑一边 kruskal，求出在最小生成树里的那些边.

对于一条边来讲:

- 如果最小生成树里所有被修改的边权都被赋成了 $+\infty$，而它未出现在树中，则证明它不可能出现在 (l,r) 这些询问的最小生成树当中. 所以我们仅仅在 (l,r) 的边集中加入最小生成树的树边.
- 如果最小生成树里所有被修改的边权都被赋成了 $-\infty$，而它出现在树中，则证明它一定会出现 (l,r) 这段的区间的最小生成树当中. 这样的话我们就可以使用并查集将这些边对应的点缩起来，并且将答案加上这些边的边权.

这样我们就将 (l,r) 这段区间的边集构造出来了. 用这些边求出来的最小生成树和直接求原图的最小生成树等价.

那么为什么我们的复杂度是对的呢?

首先，修改过的边一定会加进我们的边集，这些边的数目是 $O(len)$ 级别的.

接下来我们需要证明边集当中不会有过多的未被修改的边. 我们只会加入所有边权取 $+\infty$ 最小生成树的树边，因此我们加入的边数目不会超过当前图的点数.

现在我们只需证明每递归一层图的点数是 $O(len)$ 级别的，就可以说明图的边数是 $O(len)$ 级别的了.

证明点数是 $O(len)$ 几倍就变得十分简单了. 我们每次向下递归的时候缩掉的边是在 $-\infty$ 生成树中出现的未被修改边，反过来想就是，我们割掉了出现在 $-\infty$ 生成树当中的所有的被修改边. 显然我们最多割掉 len 条边，整张图最多分裂成 $O(len)$ 个连通块，这样的话新图点数就是 $O(len)$ 级别的了. 所以我们就证明了每次我们用来跑 kruskal 的图都是 $O(len)$ 级别的了，从而每一层的时间复杂度都是 $O(n\log n)$ 了.

时间复杂度是 $T(n)=T(\lfloor\frac{n}{2}\rfloor)+T(\lceil\frac{n}{2}\rceil)+O(n\log n)=O(n\log^2 n)$.

代码实现上可能会有一些难度. 需要注意的是并查集不能使用路径压缩，否则就不支持回退操作了. 执行缩点操作的时候也没有必要真的执行，而是每一层的 kruskal 都在上一层的并查集里直接做就可以了.

示例代码

```cpp
#include <algorithm>
#include <iostream>
#include <stack>
#include <vector>
using namespace std;
using ll = long long;
int n;
int m;
int ask;

```

```
struct bcj {
  int fa[20010];
  int size[20010];

  struct opt {
    int u;
    int v;
  };

  stack<opt> st;

  void ih() {
    for (int i = 1; i <= n; i++) fa[i] = i, size[i] = 1;
  }

  int f(int x) { return (fa[x] == x) ? x : f(fa[x]); }

  void u(int x, int y) {  // 带撤回
    int u = f(x);
    int v = f(y);
    if (u == v) return;
    if (size[u] < size[v]) swap(u, v);
    size[u] += size[v];
    fa[v] = u;
    opt o;
    o.u = u;
    o.v = v;
    st.push(o);
  }

  void undo() {
    opt o = st.top();
    st.pop();
    fa[o.v] = o.v;
    size[o.u] -= size[o.v];
  }

  void clear(int tim) {
    while (st.size() > tim) {
      undo();
    }
  }
} s, s1;

struct edge  // 静态边
{
  int u;
  int v;
  ll val;
  int mrk;

  friend bool operator<(edge a, edge b) { return a.val < b.val; }
```

```
} e[50010];

struct moved {
  int u;
  int v;
};  // 动态边

struct query {
  int num;
  ll val;
  ll ans;
} q[50010];

bool book[50010];  // 询问
vector<edge> ve[30];
vector<moved> vq;
vector<edge> tr;
ll res[30];
int tim[30];

void pushdown(int dep)  // 缩边
{
  tr.clear();  // 这里要复制一份，以免无法回撤操作
  for (int i = 0; i < ve[dep].size(); i++) {
    tr.push_back(ve[dep][i]);
  }
  sort(tr.begin(), tr.end());
  for (int i = 0; i < tr.size(); i++) {  // 无用边
    if (s1.f(tr[i].u) == s1.f(tr[i].v)) {
      tr[i].mrk = -1;
      continue;
    }
    s1.u(tr[i].u, tr[i].v);
  }
  s1.clear(0);
  res[dep + 1] = res[dep];
  for (int i = 0; i < vq.size(); i++) {
    s1.u(vq[i].u, vq[i].v);
  }
  vq.clear();
  for (int i = 0; i < tr.size(); i++) {  // 必须边
    if (tr[i].mrk == -1 || s1.f(tr[i].u) == s1.f(tr[i].v)) continue;
    tr[i].mrk = 1;
    s1.u(tr[i].u, tr[i].v);
    s.u(tr[i].u, tr[i].v);
    res[dep + 1] += tr[i].val;
  }
  s1.clear(0);
  ve[dep + 1].clear();
  for (int i = 0; i < tr.size(); i++) {  // 缩边
    if (tr[i].mrk != 0) continue;
    edge p;
```

```
    p.u = s.f(tr[i].u);
    p.v = s.f(tr[i].v);
    if (p.u == p.v) continue;
    p.val = tr[i].val;
    p.mrk = 0;
    ve[dep + 1].push_back(p);
  }
  return;
}

void solve(int l, int r, int dep) {
  tim[dep] = s.st.size();
  int mid = (l + r) / 2;
  if (r - l == 1) {  // 终止条件
    edge p;
    p.u = s.f(e[q[r].num].u);
    p.v = s.f(e[q[r].num].v);
    p.val = q[r].val;
    e[q[r].num].val = q[r].val;
    p.mrk = 0;
    ve[dep].push_back(p);
    pushdown(dep);
    q[r].ans = res[dep + 1];
    s.clear(tim[dep - 1]);
    return;
  }
  for (int i = l + 1; i <= mid; i++) {
    book[q[i].num] = true;
  }
  for (int i = mid + 1; i <= r; i++) {  // 动转静
    if (book[q[i].num]) continue;
    edge p;
    p.u = s.f(e[q[i].num].u);
    p.v = s.f(e[q[i].num].v);
    p.val = e[q[i].num].val;
    p.mrk = 0;
    ve[dep].push_back(p);
  }
  for (int i = l + 1; i <= mid; i++) {  // 询问转动态
    moved p;
    p.u = s.f(e[q[i].num].u);
    p.v = s.f(e[q[i].num].v);
    vq.push_back(p);
  }
  pushdown(dep);  // 下面的是回撤
  for (int i = mid + 1; i <= r; i++) {
    if (book[q[i].num]) continue;
    ve[dep].pop_back();
  }
  for (int i = l + 1; i <= mid; i++) {
    book[q[i].num] = false;
  }
```

```
167   solve(l, mid, dep + 1);
168   for (int i = 0; i < ve[dep].size(); i++) {
169     ve[dep][i].mrk = 0;
170   }
171   for (int i = mid + 1; i <= r; i++) {
172     book[q[i].num] = true;
173   }
174   for (int i = l + 1; i <= mid; i++) {  // 动转静
175     if (book[q[i].num]) continue;
176     edge p;
177     p.u = s.f(e[q[i].num].u);
178     p.v = s.f(e[q[i].num].v);
179     p.val = e[q[i].num].val;
180     p.mrk = 0;
181     ve[dep].push_back(p);
182   }
183   for (int i = mid + 1; i <= r; i++) {  // 询问转动
184     book[q[i].num] = false;
185     moved p;
186     p.u = s.f(e[q[i].num].u);
187     p.v = s.f(e[q[i].num].v);
188     vq.push_back(p);
189   }
190   pushdown(dep);
191   solve(mid, r, dep + 1);
192   s.clear(tim[dep - 1]);
193   return;  // 时间倒流至上一层
194 }
195
196 int main() {
197   cin.tie(nullptr)->sync_with_stdio(false);
198   cin >> n >> m >> ask;
199   s.ih();
200   s1.ih();
201   for (int i = 1; i <= m; i++) {
202     cin >> e[i].u >> e[i].v >> e[i].val;
203   }
204   for (int i = 1; i <= ask; i++) {
205     cin >> q[i].num >> q[i].val;
206   }
207   for (int i = 1; i <= ask; i++) {  // 初始动态边
208     book[q[i].num] = true;
209     moved p;
210     p.u = e[q[i].num].u;
211     p.v = e[q[i].num].v;
212     vq.push_back(p);
213   }
214   for (int i = 1; i <= m; i++) {  // 初始静态
215     if (book[i]) continue;
216     ve[1].push_back(e[i]);
217   }
218   for (int i = 1; i <= ask; i++) {
```

```
219     book[q[i].num] = false;
220   }
221   solve(0, ask, 1);
222   for (int i = 1; i <= ask; i++) {
223     cout << q[i].ans << '\n';
224   }
225   return 0;
226 }
227
```

参考资料与注释

[1] 从《Cash》谈一类分治算法的应用 

[2] 三维偏序 

[3] CQOI2011 动态逆序对 

[4] SDOI2011 拦截导弹 

[5] [Ynoi2016] 镜中的昆虫 

[6] [HNOI2010] 城市建设 

13.4.3 整体二分

引入

在信息学竞赛中，有一部分题目可以使用二分的办法来解决. 但是当这种题目有多次询问且我们每次查询都直接二分可能导致 TLE 时，就会用到整体二分. 整体二分的主体思路就是把多个查询一起解决.（所以这是一个离线算法）

可以使用整体二分解决的题目需要满足以下性质[2]：

1. 询问的答案具有可二分性
2. **修改对判定答案的贡献互相独立**，修改之间互不影响效果
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律，结合律，具有可加性
5. 题目允许使用离线算法

解释

记 $[l,r]$ 为答案的值域，$[L,R]$ 为答案的定义域.（也就是说求答案时仅考虑下标在区间 $[L,R]$ 内的操作和询问，这其中询问的答案在 $[l,r]$ 内）

- 我们首先把所有操作 **按时间顺序** 存入数组中，然后开始分治.
- 在每一层分治中，利用数据结构（常见的是树状数组）统计当前查询的答案和 mid 之间的关系.
- 根据查询出来的答案和 mid 间的关系（小于等于 mid 和大于 mid）将当前处理的操作序列分为 $q1$ 和 $q2$ 两份，并分别递归处理.
- 当 $l=r$ 时，找到答案，记录答案并返回即可.

需要注意的是，在整体二分过程中，若当前处理的值域为 $[l,r]$，则此时最终答案范围不在 $[l,r]$ 的询问会在其他时候处理.

过程

> **注**
>
> 1. 为可读性，文中代码或未采用实际竞赛中的常见写法.

[2] 许昊然. 浅谈数据结构题的几个非经典解法. 2013 年信息学奥林匹克中国国家队侯选队员论文集[1].

2. 若觉得某段代码有难以理解之处, 请先参考之前题目的解释, 因为节省篇幅解释过的内容不再赘述.

从普通二分说起:

查询全局第 k 小

题 1

在一个数列中查询第 k 小的数.

解法

当然可以直接排序. 如果用二分法呢? 可以用数据结构记录每个大小范围内有多少个数, 然后用二分法猜测, 利用数据结构检验.

题 2

在一个数列中多次查询第 k 小的数.

解法

可以对于每个询问进行一次二分; 但是, 也可以把所有的询问放在一起二分.

先考虑二分的本质: 假设要猜一个 $[l,r]$ 之间的数, 猜测之后会知道是猜大了, 猜小了还是刚好. 当然可以从 l 枚举到 r, 但更优秀的方法是二分: 猜测答案是 $m=\lfloor\frac{l+r}{2}\rfloor$, 然后去验证 m 的正确性, 再调整边界. 这样做每次询问的复杂度为 $O(\log n)$, 若询问次数为 q, 则时间复杂度为 $O(q\log n)$.

回过头来, 对于当前的所有询问, 可以去猜测所有询问的答案都是 mid, 然后去依次验证每个询问的答案应该是小于等于 mid 的还是大于 mid 的, 并将询问分为两个部分 (不大于/大于), 对于每个部分继续二分. 注意: 如果一个询问的答案是大于 mid 的, 则在将其划至右侧前需更新它的 k, 即, 如果当前数列中小于等于 mid 的数有 t 个, 则将询问划分后实际是在右区间询问第 $k-t$ 小数. 如果一个部分的 $l=r$ 了, 则结束这个部分的二分. 利用线段树的相关知识, 我们每次将整个答案可能在的区间 $[1,n]$ (假设已经离散化) 划分成了若干个部分, 这样的划分共进行了 $O(\log n)$ 次, 一次划分会将整个操作序列操作一次. 若对整个序列进行操作, 并支持对应的查询的时间复杂度为 $O(T)$, 则整体二分的时间复杂度为 $O(T\log n)$.

参考代码

```
struct Query {
  int id, k;  // 这个询问的编号, 这个询问的 k
};

int ans[N], a[N];  // ans[i] 表示编号为i的询问的答案, a 为原数列
int val[N], cnt[N];  // 离散化后, 记录对应的值及其计数 (假设已经处理好)

// 返回原数列中值域在 [l,r] 中的数的个数
int check(int l, int r) {
  int res = 0;
  for (int i = l; i <= r; i++) {
```

```
    res += cnt[i];
  }
  return res;
}

// 整体二分
void solve(int l, int r, vector<Query> q) {
  int m = (l + r) / 2;
  if (l == r) {
    for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = val[l];
    return;
  }
  vector<Query> q1, q2;
  int t = check(l, m);
  for (unsigned i = 0; i < q.size(); i++) {
    if (q[i].k <= t)
      q1.push_back(q[i]);
    else
      q[i].k -= t, q2.push_back(q[i]);
  }
  solve(l, m, q1), solve(m + 1, r, q2);
  return;
}
```

查询区间第 k 小

题 3

在一个数列中多次查询区间第 k 小的数.

解法

涉及到给定区间的查询，再按之前的方法进行二分就会导致 `check` 函数的时间复杂度爆炸. 仍然考虑询问与值域中点 m 的关系：若询问区间内小于等于 m 的数有 t 个，询问的是区间内的 k 小数，则当 $k \le t$ 时，答案应小于等于 m；否则，答案应大于 m.（注意边界问题）此处需记录一个区间小于等于指定数的数的数量，即单点加，求区间和，可用树状数组快速处理. 为提高效率，只对数列中值在值域区间 $[l,r]$ 的数进行统计，即，在进一步递归之前，不仅将询问划分，将当前处理的数按值域范围划为两半.

参考代码（关键部分）

```
struct Num {
  int p, x;
};  // 位于数列中第 p 项的数的值为 x

struct Query {
  int l, r, k, id;
};  // 一个编号为 id, 询问 [l,r] 中第 k 小数的询问

```

```
int ans[N];
void add(int p, int x);  // 树状数组, 在 p 位置加上 x
int query(int p);        // 树状数组, 求 [1,p] 的和
void clear();            // 树状数组, 清空

void solve(int l, int r, vector<Num> a, vector<Query> q)
// a中为给定数列中值在值域区间 [l,r] 中的数
{
  int m = (l + r) / 2;
  if (l == r) {
    for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = l;
    return;
  }
  vector<Num> a1, a2;
  vector<Query> q1, q2;
  for (unsigned i = 0; i < a.size(); i++)
    if (a[i].x <= m)
      a1.push_back(a[i]), add(a[i].p, 1);
    else
      a2.push_back(a[i]);
  for (unsigned i = 0; i < q.size(); i++) {
    int t = query(q[i].r) - query(q[i].l - 1);
    if (q[i].k <= t)
      q1.push_back(q[i]);
    else
      q[i].k -= t, q2.push_back(q[i]);
  }
  clear();
  solve(l, m, a1, q1), solve(m + 1, r, a2, q2);
  return;
}
```

下面提供【模板】可持久化线段树 2[2] 一题使用整体二分的，偏向竞赛风格的写法.

参考代码

```
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 200020;
int n, m;
int ans[N];
// BIT begin
int t[N];
int a[N];

int sum(int p) {
  int ans = 0;
  while (p) {
    ans += t[p];
    p -= p & (-p);
  }
```

```
  return ans;
}

void add(int p, int x) {
  while (p <= n) {
    t[p] += x;
    p += p & (-p);
  }
}

// BIT end
int tot = 0;

struct Query {
  int l, r, k, id, type;  // set values to -1 when they are not used!
} q[N * 2], q1[N * 2], q2[N * 2];

void solve(int l, int r, int ql, int qr) {
  if (ql > qr) return;
  if (l == r) {
    for (int i = ql; i <= qr; i++)
      if (q[i].type == 2) ans[q[i].id] = l;
    return;
  }
  int mid = (l + r) / 2, cnt1 = 0, cnt2 = 0;
  for (int i = ql; i <= qr; i++) {
    if (q[i].type == 1) {
      if (q[i].l <= mid) {
        add(q[i].id, 1);
        q1[++cnt1] = q[i];
      } else
        q2[++cnt2] = q[i];
    } else {
      int x = sum(q[i].r) - sum(q[i].l - 1);
      if (q[i].k <= x)
        q1[++cnt1] = q[i];
      else {
        q[i].k -= x;
        q2[++cnt2] = q[i];
      }
    }
  }
  // rollback changes
  for (int i = 1; i <= cnt1; i++)
    if (q1[i].type == 1) add(q1[i].id, -1);
  // move them to the main array
  for (int i = 1; i <= cnt1; i++) q[i + ql - 1] = q1[i];
  for (int i = 1; i <= cnt2; i++) q[i + cnt1 + ql - 1] = q2[i];
  solve(l, mid, ql, cnt1 + ql - 1);
  solve(mid + 1, r, cnt1 + ql, qr);
}

```

```cpp
pair<int, int> b[N];
int toRaw[N];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  // read and discrete input data
  for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    b[i].first = x;
    b[i].second = i;
  }
  sort(b + 1, b + n + 1);
  int cnt = 0;
  for (int i = 1; i <= n; i++) {
    if (b[i].first != b[i - 1].first) cnt++;
    a[b[i].second] = cnt;
    toRaw[cnt] = b[i].first;
  }
  for (int i = 1; i <= n; i++) {
    q[++tot] = {a[i], -1, -1, i, 1};
  }
  for (int i = 1; i <= m; i++) {
    int l, r, k;
    cin >> l >> r >> k;
    q[++tot] = {l, r, k, i, 2};
  }
  solve(0, cnt + 1, 1, tot);
  for (int i = 1; i <= m; i++) cout << toRaw[ans[i]] << '\n';
}

```

带修区间第 k 小

题 4（Dynamic Rankings[3]）

给定一个数列，要支持单点修改，区间查第 k 小.

解法

修改操作可以直接理解为从原数列中删去一个数再添加一个数，为方便起见，将询问和修改统称为「操作」. 因后面的操作会依附于之前的操作，不能如题 3 一样将统计和处理询问分开，故可将所有操作存于一个数组，用标识区分类型，依次处理每个操作. 为便于处理树状数组，修改操作可分拆为擦除操作和插入操作.

优化

1. 注意到每次对于操作进行分类时，只会更改操作顺序，故可直接在原数组上操作. 具体实现，在二分时将记录操作的 q, a 数组换为一个大的全局数组，二分时记录信息变为 L, R，即当前处理的操作是全局数组上的哪个区间. 利用临时数组记录当前的分类情况，进一步递归前将临时数组信息写回原数组.

2. 树状数组每次清空会导致时间复杂度爆炸，可采用每次使用树状数组时记录当前修改位置（这已由 1 中提到的临时数组实现），本次操作结束后在原位置加 -1 的方法快速清零.
3. 一开始对于数列的初始化操作可简化为插入操作.

参考代码（关键部分）

```
struct Opt {
  int x, y, k, type, id;
  // 对于询问, type = 1, x, y 表示区间左右边界, k 表示询问第 k 小
  // 对于修改, type = 0, x 表示修改位置, y 表示修改后的值,
  // k 表示当前操作是插入(1)还是擦除(-1), 更新树状数组时使用.
  // id 记录每个操作原先的编号, 因二分过程中操作顺序会被打散
};

Opt q[N], q1[N], q2[N];
// q 为所有操作,
// 二分过程中, 分到左边的操作存到 q1 中, 分到右边的操作存到 q2 中.
int ans[N];
void add(int p, int x);
int query(int p);  // 树状数组函数, 含义见题3

void solve(int l, int r, int L, int R)
// 当前的值域范围为 [l,r], 处理的操作的区间为 [L,R]
{
  if (l > r || L > R) return;
  int cnt1 = 0, cnt2 = 0, m = (l + r) / 2;
  // cnt1, cnt2 分别为分到左边, 分到右边的操作数
  if (l == r) {
    for (int i = L; i <= R; i++)
      if (q[i].type == 1) ans[q[i].id] = l;
    return;
  }
  for (int i = L; i <= R; i++)
    if (q[i].type == 1) {  // 是询问: 进行分类
      int t = query(q[i].y) - query(q[i].x - 1);
      if (q[i].k <= t)
        q1[++cnt1] = q[i];
      else
        q[i].k -= t, q2[++cnt2] = q[i];
    } else
      // 是修改: 更新树状数组 & 分类
      if (q[i].y <= m)
        add(q[i].x, q[i].k), q1[++cnt1] = q[i];
      else
        q2[++cnt2] = q[i];
  for (int i = 1; i <= cnt1; i++)
    if (q1[i].type == 0) add(q1[i].x, -q1[i].k);  // 清空树状数组
  for (int i = 1; i <= cnt1; i++) q[L + i - 1] = q1[i];
  for (int i = 1; i <= cnt2; i++)
    q[L + cnt1 + i - 1] = q2[i];  // 将临时数组中的元素合并回原数组
  solve(l, m, L, L + cnt1 - 1), solve(m + 1, r, L + cnt1, R);
  return;
```

```
47 }
```

针对静态序列的优化

题 5（【模板】可持久化线段树 2[4]）

给定一个序列，区间查询第 k 小.

解法

树套树和整体二分实现带修区间第 k 小问题的复杂度都为 $O(n\log^2 n)$，但静态区间第 k 小问题可以使用可持久化线段树在 $O(n\log n)$ 时间复杂度内解决，而几乎所有整体二分实现的静态区间第 k 小问题代码时间复杂度都是 $O(n\log^2 n)$，面对大数据范围时存在 TLE 的风险.（这里默认值域与序列长度同阶，值域与序列长不同阶的情况可以通过离散化转化为同阶情况）

优化

1. 对于每一轮划分，如果当前数列中小于等于 mid 的数有 t 个，则将询问划分后实际是在右区间询问第 $k-t$ 小数，因此对划分到右区间的询问做出了修改. 如果答案的原始值域为 $[L,R]$，某次划分的答案值域为 $[l,r]$，那么对于参与此次划分的询问，$[L,l)$ 中所有数值对它们的影响已经在之前被消除了.
2. 由于需要使每轮划分都仅和当前答案值域 $[l,r]$ 有关，树状数组需要多次载入和清空.

如果划分不仅仅和当前答案值域有关呢?

由此可以得到一个与全局序列有关的优化方法: 维护一个指针 pos 追踪每轮划分的 mid（分治中心），将所有 $\le pos$ 的元素对应的下标在树状数组中置为 1，树状数组的其余位置置为 0. 每次划分之前移动 pos 并更新树状数组. 指针 pos 移动的次数与 $n\log n$ 同阶. 划分时对每一个询问查询树状数组中对应区间的值，满足则划分至左区间，否则划分至右区间，**不需要对询问做出修改**.

由于要追踪分治中心，需要让 pos 准确地更新树状数组. 在整体二分之前将序列按元素大小排序并记录元素对应下标，指针移动时在树状数组中对下标进行相应修改. 对于绝大多数 **可以用整体二分解决并且不带修改的问题**，都可以应用此种优化以大幅降低数据结构的使用次数.

由于减少了很多树状数组的载入和清空操作，应用这种优化通常情况下会明显提升整体二分的效率（即使只是常数优化），对于静态区间第 k 小值问题而言效率完全不差于时间复杂度更优的可持久化线段树. 值得注意的是，对于静态区间第 k 小值问题也存在时间复杂度 $O(n\log n)$ 的整体二分实现.

参考代码（关键部分）

```
struct Query {
  int i, l, r, k;
};  // 第 i 次询问查询区间 [l,r] 的第 k 小值

Query s[200005], t1[200005], t2[200005];
int n, m, cnt, pos, p[200005], ans[200005];
pair<int, int> a[200005];

void add(int x, int y);  // 树状数组 位置 x 加 y
int sum(int x);          // 树状数组 [1,x] 前缀和
```

```
11
12 // 当前处理的询问为 [l,r],答案值域为 [ql,qr]
13 void overall_binary(int l, int r, int ql, int qr) {
14   if (l > r) return;
15   if (ql == qr) {
16     for (int i = l; i <= r; i++) ans[s[i].i] = ql;
17     return;
18   }
19   int cnt1 = 0, cnt2 = 0, mid = (ql + qr) >> 1;
20   // 追踪分治中心,认为 [1,pos] 的值已经载入树状数组
21   while (pos <= n - 1 && a[pos + 1].first <= mid)
22     add(a[pos + 1].second, 1), ++pos;
23   while (pos >= 1 && a[pos].first > mid) add(a[pos].second, -1), --pos;
24
25   for (int i = l; i <= r; i++) {
26     int now = sum(s[i].r) - sum(s[i].l - 1);
27     if (s[i].k <= now)
28       t1[++cnt1] = s[i];
29     else
30       t2[++cnt2] = s[i];  // 注意 不应修改询问信息
31   }
32   for (int i = 1; i <= cnt1; i++) s[l + i - 1] = t1[i];
33   for (int i = 1; i <= cnt2; i++) s[l + cnt1 + i - 1] = t2[i];
34
35   overall_binary(l, l + cnt1 - 1, ql, mid);
36   overall_binary(l + cnt1, r, mid + 1, qr);
37 }
38
39 int main() {
40   scanf("%d%d", &n, &m);
41   for (int i = 1; i <= n; i++) {
42     scanf("%d", &a[i].first);
43     a[i].second = i;
44     p[++cnt] = a[i].first;
45   }
46   sort(a + 1, a + n + 1);  // 对序列排序 离散化
47   sort(p + 1, p + n + 1);
48   cnt = unique(p + 1, p + n + 1) - p - 1;
49   for (int i = 1; i <= n; i++)
50     a[i].first = lower_bound(p + 1, p + cnt + 1, a[i].first) - p;
51   // 省略读入询问
52   overall_binary(1, m, 1, cnt);
53   for (int i = 1; i <= n; i++) printf("%d\n", p[ans[i]]);
54   return 0;
55 }
```

区间前驱后继

题 6

在一个数列中多次查询 k 在区间中的前驱（严格小于 k，且最大的数）或后继（严格大于 k，且最小的数），保证存在这样的数.

解法

以前驱为例，使用数据结构解决此种问题的方法一般是先查询区间内有多少严格小于 k 的数（设它们的数量为 x），再查询区间第 x 小的数．后继则是查询区间内有多少不大于 k 的数（数量为 x），然后查询区间第 $x+1$ 小的数．

考虑使用整体二分解决这个问题：整体二分是一种高效求解区间第 k 小的离线算法，而 CDQ 分治→13.4.2@p3176 可以离线高效求解区间内的排名．先跑一遍 CDQ 分治求出排名就可以使用整体二分得到区间内部的前驱和后继了．

此问题还可以用 CDQ 分治套线段树离线一遍解决，但效率远低于跑两遍的 CDQ 分治 + 整体二分．

构造单调性序列

题 7（Sequence[5]）

给定一个序列，每次操作可以把某个数 $+1$ 或 -1．要求把序列变成单调不降的，并且修改后的数列只能出现修改前的数，输出最小操作次数．

解法

此类题目也可以使用动态规划或反悔贪心解决．

在满足操作次数最小化的前提下，一定存在一种方案使得最后序列中的每个数都是序列修改前存在的，这个结论可以使用数学归纳法证明．由于题目并不需要最终序列的信息，问题转化为求出最小操作次数．

由于要求最终的序列单调不降，可以使用整体二分．每轮整体二分判定最终序列区间 $[l,r]$ 的值域，此时答案的值域为 $[ql,qr]$．令 $mid=\left\lfloor\frac{ql+qr}{2}\right\rfloor$，每轮二分开始时默认将所有数划分至 $[mid+1,qr]$（要划分到 $[ql,mid]$ 的数设为 0 个），初始代价设为将序列区间 $[l,r]$ 全部置为 $mid+1$ 的操作次数．依次枚举区间 $[l,r]$ 中的数 i 并且计算将 $[l,i]$ 置为 mid、将 $[i+1,r]$ 置为 $mid+1$ 的操作次数之和，如果优于之前的操作次数则更新最少操作次数和要划分到 $[ql,mid]$ 的数的个数．

划分时已经保证了最终序列的单调性不被破坏，同时因为每次都取最小操作次数，最终被划分至左区间的数取 mid 一定比取 $mid+1$ 更优，故整体二分得到的序列一定是单调不降且操作次数最小的．计算操作次数输出即可．

参考代码（关键部分）

```cpp
int a[500005], ans[500005];  // a:原序列 ans:构造的序列

void overall_binary(int l, int r, int ql, int qr) {
  if (l > r) return;
  if (ql == qr) {
    for (int i = l; i <= r; i++) ans[i] = ql;
    return;
  }
  int cnt = 0,
      mid = ql + ((qr - ql) >> 1);  // 默认开始都填 mid+1 全部划分到右区间
  long long res = 0ll, sum = 0ll;
```

```
  for (int i = l; i <= r; i++) sum += abs(a[i] - (mid + 1));
  res = sum;
  for (int i = l; i <= r;
       i++) {  // 尝试把 [l,i] 从 mid+1 换成 mid 并且划分到左区间
    sum -= abs(a[i] - (mid + 1));
    sum += abs(a[i] - mid);
    if (sum < res) cnt = i - l + 1, res = sum;  // 发现 [l,i] 取 mid 更优,更新
  }
  overall_binary(l, l + cnt - 1, ql, mid);
  overall_binary(l + cnt, r, mid + 1, qr);
}
```

参考习题

- 「国家集训队」矩阵乘法[6]
- 「POI2011 R3 Day2」流星 Meteors[7]
- 二逼平衡树[8]
- [BalticOI 2004] Sequence 数字序列[9]

参考资料与注释

- 许昊然. 浅谈数据结构题的几个非经典解法. 2013 年信息学奥林匹克中国国家队侯选队员论文集[10].

[1] 2013 年信息学奥林匹克中国国家队侯选队员论文集

[2] 【模板】可持久化线段树 2

[3] Dynamic Rankings

[4] 【模板】可持久化线段树 2

[5] Sequence

[6] 「国家集训队」矩阵乘法

[7] 「POI2011 R3 Day2」流星 Meteors

[8] 二逼平衡树

[9] [BalticOI 2004] Sequence 数字序列

[10] 2013 年信息学奥林匹克中国国家队侯选队员论文集

13.4.4 莫队算法

13.4.4.1 莫队算法简介

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan

莫队算法是由莫涛提出的算法. 在莫涛提出莫队算法之前，莫队算法已经在 Codeforces 的高手圈里小范围流传，但是莫涛是第一个对莫队算法进行详细归纳总结的人. 莫涛提出莫队算法时，只分析了普通莫队算法，但是经过 OIer 和 ACMer 的集体智慧改造，莫队有了多种扩展版本.

莫队算法可以解决一类离线区间询问问题，适用性极为广泛. 同时将其加以扩展，便能轻松处理树上路径询问以及支持修改操作.

13.4.4.2 普通莫队算法

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan

形式

假设 $n=m$，那么对于序列上的区间询问问题，如果从 $[l,r]$ 的答案能够 $O(1)$ 扩展到 $[l-1,r],[l+1,r],[l,r+1],[l,r-1]$（即与 $[l,r]$ 相邻的区间）的答案，那么可以在 $O(n\sqrt{n})$ 的复杂度内求出所有询问的答案.

解释

离线后排序，顺序处理每个询问，暴力从上一个区间的答案转移到下一个区间答案（一步一步移动即可）.

排序方法

对于区间 $[l,r]$, 以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序.

实现

```
void move(int pos, int sign) {
  // update nowAns
}

void solve() {
  BLOCK_SIZE = int(ceil(pow(n, 0.5)));
  sort(querys, querys + m);
  for (int i = 0; i < m; ++i) {
    const query &q = querys[i];
    while (l > q.l) move(--l, 1);
    while (r < q.r) move(++r, 1);
    while (l < q.l) move(l++, -1);
    while (r > q.r) move(r--, -1);
    ans[q.id] = nowAns;
  }
}
```

复杂度分析

以下的情况在 n 和 m 同阶的前提下讨论.

首先是分块这一步，这一步的时间复杂度是 $O(\sqrt{n}\cdot\sqrt{n}\log\sqrt{n}+n\log n)=O(n\log n)$.

接着就到了莫队算法的精髓了，下面我们用通俗易懂的初中方法来证明它的时间复杂度是 $O(n\sqrt{n})$.

证明

证：令每一块中 L 的最大值为 $\max_1,\max_2,\max_3,\cdots,\max_{\lceil\sqrt{n}\rceil}$.

由第一次排序可知，$\max_1\le\max_2\le\cdots\le\max_{\lceil\sqrt{n}\rceil}$.

显然，对于每一块暴力求出第一个询问的时间复杂度为 $O(n)$.

考虑最坏的情况, 在每一块中, R 的最大值均为 n, 每次修改操作均要将 L 由 $\max_{i-1}$ 修改至 $\max_i$ 或由 $\max_i$ 修改至 $\max_{i-1}$.

考虑 R：因为 R 在块中已经排好序，所以在同一块修改完它的时间复杂度为 $O(n)$. 对于所有块就是 $O(n\sqrt{n})$.

重点分析 L：因为每一次改变的时间复杂度都是 $O(\max_i-\max_{i-1})$ 的，所以在同一块中时间复杂度为 $O(\sqrt{n}\cdot(\max_i-\max_{i-1}))$.

将每一块 L 的时间复杂度合在一起，可以得到:

对于 L 的总时间复杂度为

$$O(\sqrt{n}(\max_1-1)+\sqrt{n}(\max_2-\max_1)+\sqrt{n}(\max_3-\max_2)+\cdots+\sqrt{n}(\max_{\lceil\sqrt{n}\rceil}-\max_{\lceil\sqrt{n}\rceil-1}))$$

$$O(\sqrt{n}\cdot(\max_1-1+\max_2-\max_1+\max_3-\max_2+\cdots+\max_{\lceil\sqrt{n}\rceil-1}-\max_{\lceil\sqrt{n}\rceil-2}+\max_{\lceil\sqrt{n}\rceil}-\max_{\lceil\sqrt{n}\rceil-1}$$

$$O(\sqrt{n}\cdot(\max_{\lceil\sqrt{n}\rceil}-1))$$

（裂项求和）

由题可知 $\max_{\lceil\sqrt{n}\rceil}$ 最大为 n，所以 L 的总时间复杂度最坏情况下为 $O(n\sqrt{n})$.

综上所述，莫队算法的时间复杂度为 $O(n\sqrt{n})$.

但是对于 m 的其他取值，如 $m<n$，分块方式需要改变才能变的更优.

怎么分块呢?

我们设块长度为 S，那么对于任意多个在同一块内的询问，挪动的距离就是 n，一共 $\frac{n}{S}$ 个块，移动的总次数就是 $\frac{n^2}{S}$，移动可能跨越块，所以还要加上一个 mS 的复杂度，总复杂度为 $O\left(\frac{n^2}{S}+mS\right)$，我们要让这个值尽量小，那么就要将这两个项尽量相等，发现 S 取 $\frac{n}{\sqrt{m}}$ 是最优的，此时复杂度为 $O\left(\frac{n^2}{\frac{n}{\sqrt{m}}}+m\left(\frac{n}{\sqrt{m}}\right)\right)=O(n\sqrt{m})$.

事实上，如果块长度的设定不准确，则莫队的时间复杂度会受到很大影响. 例如，如果 m 与 $\sqrt{n}$ 同阶，并且块长误设为 $\sqrt{n}$，则可以很容易构造出一组数据使其时间复杂度为 $O(n\sqrt{n})$ 而不是正确的 $O(n^{5/4})$.

莫队算法看起来十分暴力，很大程度上是因为莫队算法的分块排序方法看起来很粗糙. 我们会想到通过看上去更精细的排序方法对所有区间排序. 一种方法是把所有区间 $[l,r]$ 看成平面上的点 (l,r)，并对所有点建立曼哈顿最小生成树，每次沿着曼哈顿最小生成树的边在询问之间转移答案. 这样看起来可以改善莫队算法的时间复杂度，但是实际上对询问分块排序的方法的时间复杂度上界已经是最优的了.

假设 n,m 同阶且 n 是完全平方数. 我们考虑形如 $[a\sqrt{n},b\sqrt{n}](1\le a,b\le\sqrt{n})$ 的区间，这样的区间一共有 n 个. 如果把所有的区间看成平面上的点，则两点之间的曼哈顿距离恰好为两区间的转移代价，并且任意两个区间之间的最小曼哈顿距离为 $\sqrt{n}$，所以处理所有询问的时间复杂度最小为 $O(n\sqrt{n})$. 其它情况的数据构造方法与之类似.

莫队算法还有一个特点：当 n 不变时，m 越大，处理每次询问的平均转移代价就越小. 一些其他的离线算法也具有同样的特点（如求 LCA 的 Tarjan 算法），但是莫队算法的平均转移代价随 m 的变化最明显.

例题 & 代码

例题 「国家集训队」小 Z 的袜子[1]

题目大意：

有一个长度为 n 的序列 $\{c_i\}$. 现在给出 m 个询问，每次给出两个数 l,r，从编号在 l 到 r 之间的数中随机选出两个不同的数，求两个数相等的概率.

过程

思路：莫队算法模板题.

对于区间 $[l,r]$，以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序.

然后从序列的第一个询问开始计算答案，第一个询问通过直接暴力算出，复杂度为 $O(n)$，后面的询问在前一个询问的基础上得到答案.

具体做法：

对于区间 $[i,i]$，由于区间只有一个元素，我们很容易就能知道答案. 然后一步一步从当前区间（已知答案）向下一个区间靠近.

我们设 $col[i]$ 表示当前颜色 i 出现了多少次，ans 表示当前共有多少种可行的配对方案（有多少种可以选到一双颜色相同的袜子）. 然后每次移动的时候更新答案：设当前颜色为 k，如果是增长区间就是 ans 加上 $\binom{col[k]+1}{2}-\binom{col[k]}{2}$；如果是缩短就是 ans 减去 $\binom{col[k]}{2}-\binom{col[k]-1}{2}$. 这个询问的答案就是 $\dfrac{ans}{\binom{r-l+1}{2}}$.

这里有个优化：$\binom{a}{2}=\dfrac{a(a-1)}{2}$.

所以 $\binom{a+1}{2}-\binom{a}{2}=\dfrac{(a+1)a}{2}-\dfrac{a(a-1)}{2}=\dfrac{a}{2}\cdot(a+1-a+1)=\dfrac{a}{2}\cdot 2=a$.

所以 $\binom{col[k]+1}{2}-\binom{col[k]}{2}=col[k]$.

算法总复杂度：$O(n\sqrt{n})$

下面的代码中 `deno` 表示答案的分母 (denominator)，`nume` 表示分子 (numerator)，`sqn` 表示块的大小：$\sqrt{n}$，`arr` 是输入的数组，`node` 是存储询问的结构体，`tab` 是询问序列（排序后的），`col` 同上所述.

注意：由于 `++l` 和 `--r` 的存在，下面代码中的移动区间的 4 个 while 循环的位置很关键，不能随意改变它们之间的位置关系.

关于四个循环位置的讨论

莫队区间的移动过程，就相当于加入了 $[1,r]$ 的元素，并删除了 $[1,l-1]$ 的元素. 因此，

- 对于 $l\le r$ 的情况，$[1,l-1]$ 的元素相当于被加入了一次又被删除了一次，$[l,r]$ 的元素被加入一次，$[r+1,+\infty)$ 的元素没有被加入. 这个区间是合法区间.
- 对于 $l=r+1$ 的情况，$[1,r]$ 的元素相当于被加入了一次又被删除了一次，$[r+1,+\infty)$ 的元素没有被加入. 这时这个区间表示空区间.
- 对于 $l>r+1$ 的情况，那么 $[r+1,l-1]$（这个区间非空）的元素被删除了一次但没有被加入，因此这个元素被加入的次数是负数.

因此，如果某时刻出现 $l>r+1$ 的情况，那么会存在一个元素，它的加入次数是负数. 这在某些题目会出现问题，例如我们如果用一个 `set` 维护区间中的所有数，就会出现「需要删除 `set` 中不存在的元素」的问题.

代码中的四个 while 循环一共有 $4!=24$ 种排列顺序. 不妨设第一个循环用于操作左端点，就有以下 12 种排列（另外 12 种是对称的）. 下表列出了这 12 种写法的正确性，还给出了错误写法的反例.

循环顺序	正确性	反例或注释
`l--,l++,r--,r++`	错误	$l<r<l'<r'$
`l--,l++,r++,r--`	错误	$l<r<l'<r'$
`l--,r--,l++,r++`	错误	$l<r<l'<r'$
`l--,r--,r++,l++`	正确	证明较繁琐
`l--,r++,l++,r--`	正确	
`l--,r++,r--,l++`	正确	
`l++,l--,r--,r++`	错误	$l<r<l'<r'$

`l++,l--,r++,r--`	错误	$l < r < l' < r'$
`l++,r++,l--,r--`	错误	$l < r < l' < r'$
`l++,r++,r--,l--`	错误	$l < r < l' < r'$
`l++,r--,l--,r++`	错误	$l < r < l' < r'$
`l++,r--,r++,l--`	错误	$l < r < l' < r'$

全部 24 种排列中只有 6 种是正确的，其中有 2 种的证明较繁琐，这里只给出其中 4 种的证明.

这 4 种正确写法的共同特点是，前两步先扩大区间（`l--` 或 `r++`），后两步再缩小区间（`l++` 或 `r--`）. 这样写，前两步是扩大区间，可以保持 $l \le r+1$；执行完前两步后，$l \le l' \le r' \le r$ 一定成立，再执行后两步只会把区间缩小到 $[l', r']$，依然有 $l \le r+1$，因此这样写是正确的.

实现

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;
constexpr int N = 50005;
int n, m, maxn;
int c[N];
long long sum;
int cnt[N];
long long ans1[N], ans2[N];

struct query {
  int l, r, id;

  bool operator<(const query &x) const {  // 重载<运算符
    if (l / maxn != x.l / maxn) return l < x.l;
    return (l / maxn) & 1 ? r < x.r : r > x.r;
  }
} a[N];

void add(int i) {
  sum += cnt[i];
  cnt[i]++;
}

void del(int i) {
  cnt[i]--;
  sum -= cnt[i];
}

long long gcd(long long a, long long b) { return b ? gcd(b, a % b) : a; }

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
```

```
  maxn = sqrt(n);
  for (int i = 1; i <= n; i++) cin >> c[i];
  for (int i = 0; i < m; i++) cin >> a[i].l >> a[i].r, a[i].id = i;
  sort(a, a + m);
  for (int i = 0, l = 1, r = 0; i < m; i++) {  // 具体实现
    if (a[i].l == a[i].r) {
      ans1[a[i].id] = 0, ans2[a[i].id] = 1;
      continue;
    }
    while (l > a[i].l) add(c[--l]);
    while (r < a[i].r) add(c[++r]);
    while (l < a[i].l) del(c[l++]);
    while (r > a[i].r) del(c[r--]);
    ans1[a[i].id] = sum;
    ans2[a[i].id] = (long long)(r - l + 1) * (r - l) / 2;
  }
  for (int i = 0; i < m; i++) {
    if (ans1[i] != 0) {
      long long g = gcd(ans1[i], ans2[i]);
      ans1[i] /= g, ans2[i] /= g;
    } else
      ans2[i] = 1;
    cout << ans1[i] << '/' << ans2[i] << '\n';
  }
  return 0;
}
```

普通莫队的优化

过程

我们看一下下面这组数据

```
// 设块的大小为 2 (假设)
1 1
2 100
3 1
4 100
```

手动模拟一下可以发现，r 指针的移动次数大概为 300 次，我们处理完第一个块之后，$l = 2, r = 100$，此时只需要移动两次 l 指针就可以得到第四个询问的答案，但是我们却将 r 指针移动到 1 来获取第三个询问的答案，再移动到 100 获取第四个询问的答案，这样多了九十几次的指针移动．我们怎么优化这个地方呢？这里我们就要用到奇偶化排序．

什么是奇偶化排序？奇偶化排序即对于属于奇数块的询问，r 按从小到大排序，对于属于偶数块的排序，r 从大到小排序，这样我们的 r 指针在处理完这个奇数块的问题后，将在返回的途中处理偶数块的问题，再向 n 移动处理下一个奇数块的问题，优化了 r 指针的移动次数，一般情况下，这种优化能让程序快 30% 左右．

实现

排序代码：

- 压行

```cpp
// clang-format off
// 这里有个小细节等下会讲
int unit; // 块的大小
struct node {
  int l, r, id;
  bool operator < (const node &x) const {
    return l / unit == x.l / unit ? (r == x.r ? 0 : ((l / unit) & 1) ^ (r < x.r)) : l < x.l;
  }
};
```

- 不压行

```cpp
struct node {
  int l, r, id;

  bool operator<(const node &x) const {
    if (l / unit != x.l / unit) return l < x.l;
    // 注意下面两行不能写小于（大于）等于，否则会出错（详见下面的小细节）
    if ((l / unit) & 1) return r < x.r;
    return r > x.r;
  }
};
```

> ⚠ **小细节**
>
> 如果使用 `sort` 比较两个结构体，不能出现 $a < b$ 和 $b < a$ 同时为真的情况，否则会运行错误，详见 常见错误→2.6.3@p110.

对于压行版，如果没有 `r == x.r` 的特判，当 l 属于同一奇数块且 r 相等时，会出现上面小细节中的问题（自己手动模拟一下），对于不压行版，如果写成小于（大于）等于，则也会出现同样的问题.

参考资料

- 莫队算法学习笔记 | Sengxian's Blog[2]

外部参考资料

[1]「国家集训队」小 Z 的袜子  [2] 莫队算法学习笔记 | Sengxian's Blog 

13.4.4.3 带修改莫队

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan, renbaoshuo, Lixuannan

请确保您已经会普通莫队算法了．如果您还不会，请先阅读前面的 普通莫队算法→13.4.4.2@p3209.

特点

普通莫队是不能带修改的.

我们可以强行让它可以修改，就像 DP 一样，可以强行加上一维 **时间维**, 表示这次操作的时间.

时间维表示经历的修改次数.

即把询问 $[l, r]$ 变成 $[l, r, \text{time}]$.

那么我们的坐标也可以在时间维上移动，即 $[l, r, \text{time}]$ 多了一维可以移动的方向，可以变成:

- $[l-1, r, \text{time}]$

- $[l+1,r,\text{time}]$
- $[l,r-1,\text{time}]$
- $[l,r+1,\text{time}]$
- $[l,r,\text{time}-1]$
- $[l,r,\text{time}+1]$

这样的转移也是 $O(1)$ 的，但是我们排序又多了一个关键字，再搞搞就行了.

可以用和普通莫队类似的方法排序转移，做到 $O(n^{5/3})$.

这一次我们排序的方式是以 $n^{2/3}$ 为一块，分成了 $n^{1/3}$ 块，第一关键字是左端点所在块，第二关键字是右端点所在块，第三关键字是时间.

最优块长以及时间复杂度分析

我们设序列长为 n，m 个询问，t 个修改.

带修莫队排序的第二关键字是右端点所在块编号，不同于普通莫队.

想一想，如果不把右端点分块:

- 乱序的右端点对于每个询问会移动 n 次.
- 有序的右端点会带来乱序的时间，每次询问会移动 t 次.

无论哪一种情况，带来的时间开销都无法接受.

接下来分析时间复杂度.

设块长为 s，则有 $\frac{n}{s}$ 个块. 对于块 i 和块 j，记有 $q_{i,j}$ 个询问的左端点位于块 i，右端点位于块 j.

每「组」左右端点不换块的询问 (i,j)，端点每次移动 $O(s)$ 次，时间单调递增，$O(t)$.

左右端点换块的时间忽略不计.

表示一下就是:

$$\sum_{i=1}^{n/s}\sum_{j=i+1}^{n/s}(q_{i,j}\cdot s+t)$$

$$=ms+\left(\frac{n}{s}\right)^2 t$$

$$=ms+\frac{n^2t}{s^2}$$

考虑求导求此式极小值. 设 $f(s)=ms+\frac{n^2t}{s^2}$. 那 $f'(s)=m-\frac{2n^2t}{s^3}=0$.

得 $s=\sqrt[3]{\frac{2n^2t}{m}}=\frac{2^{1/3}n^{2/3}t^{1/3}}{m^{1/3}}=s_0$.

也就是当块长取 $\frac{n^{2/3}t^{1/3}}{m^{1/3}}$ 时有最优时间复杂度 $O\left(n^{2/3}m^{2/3}t^{1/3}\right)$.

常说的 $O\left(n^{5/3}\right)$ 便是把 n,m,t 当做同数量级的时间复杂度.

实际操作中还是推荐设定 $n^{2/3}$ 为块长.

例题

例题 「国家集训队」数颜色/维护队列[1]

题目大意：给你一个序列，M 个操作，有两种操作:

1. 修改序列上某一位的数字
2. 询问区间 $[l,r]$ 中数字的种类数（多个相同的数字只算一个）

我们不难发现，如果不带操作 1（修改）的话，我们就能轻松用普通莫队解决.

但是题目还带单点修改，所以用 **带修改的莫队**.

过程

先考虑普通莫队的做法：

- 每次扩大区间时，每加入一个数字，则统计它已经出现的次数，如果加入前这种数字出现次数为 0，则说明这是一种新的数字，答案 $+1$. 然后这种数字的出现次数 $+1$.
- 每次减小区间时，每删除一个数字，则统计它删除后的出现次数，如果删除后这种数字出现次数为 0，则说明这种数字已经从当前的区间内删光了，也就是当前区间减少了一种颜色，答案 -1. 然后这种数字的出现次数 -1.

现在再来考虑修改：

- 单点修改，把某一位的数字修改掉. 假如我们是从一个经历修改次数为 i 的询问转移到一个经历修改次数为 j 的询问上，且 $i < j$ 的话，我们就需要把第 $i+1$ 个到第 j 个修改强行加上.
- 假如 $j < i$ 的话，则需要把第 i 个到第 $j+1$ 个修改强行还原.

怎么强行加上一个修改呢？假设一个修改是修改第 pos 个位置上的颜色，原本 pos 上的颜色为 a，修改后颜色为 b，还假设当前莫队的区间扩展到了 $[l, r]$.

- 加上这个修改：我们首先判断 pos 是否在区间 $[l, r]$ 内. 如果是的话，我们等于是从区间中删掉颜色 a，加上颜色 b，并且当前颜色序列的第 pos 项的颜色改成 b. 如果不在区间 $[l, r]$ 内的话，我们就直接修改当前颜色序列的第 pos 项为 b.
- 还原这个修改：等于加上一个修改第 pos 项、把颜色 b 改成颜色 a 的修改.

因此这道题就这样用带修改莫队轻松解决啦！

实现

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;

long long qsize;

struct query {
  long long id, t, l, r;

  bool operator<(query b) const {
    if (l / qsize != b.l / qsize) {
      return l / qsize > b.l / qsize;
    } else if (r / qsize != b.r / qsize) {
      return r / qsize > b.r / qsize;
    } else {
      return t > b.t;
    }
  }
} q[150009];

struct operation {
  long long p, x;
} r[150009];
```

```
25
26 char op;
27 long long n, m, x, y, cur, qcnt, rcnt, mp[1500009], a[150009], ans[150009];
28
29 void add(long long x) {
30   if (!mp[x]) {
31     cur += 1;
32   }
33   mp[x] += 1;
34 }
35
36 void del(long long x) {
37   mp[x] -= 1;
38   if (!mp[x]) {
39     cur -= 1;
40   }
41 }
42
43 void process() {
44   sort(q + 1, q + qcnt + 1);
45   long long L = 1, R = 0, last = 0;
46   for (long long i = 1; i <= qcnt; i++) {
47     while (R < q[i].r) {
48       add(a[++R]);
49     }
50     while (R > q[i].r) {
51       del(a[R--]);
52     }
53     while (L > q[i].l) {
54       add(a[--L]);
55     }
56     while (L < q[i].l) {
57       del(a[L++]);
58     }
59     while (last < q[i].t) {
60       last += 1;
61       if (r[last].p >= L && r[last].p <= R) {
62         add(r[last].x);
63         del(a[r[last].p]);
64       }
65       swap(a[r[last].p], r[last].x);
66     }
67     while (last > q[i].t) {
68       if (r[last].p >= L && r[last].p <= R) {
69         add(r[last].x);
70         del(a[r[last].p]);
71       }
72       swap(a[r[last].p], r[last].x);
73       last -= 1;
74     }
75     ans[q[i].id] = cur;
76   }
```

```
}

signed main() {
  cin.tie(nullptr);
  ios::sync_with_stdio(false);
  cin >> n >> m;
  qsize = pow(n, 2.0 / 3.0);
  for (long long i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (long long i = 1; i <= m; i++) {
    cin >> op >> x >> y;
    if (op == 'Q') {
      ++qcnt, q[qcnt] = {qcnt, rcnt, x, y};
    } else if (op == 'R') {
      r[++rcnt] = {x, y};
    }
  }
  process();
  for (long long i = 1; i <= qcnt; i++) {
    cout << ans[i] << '\n';
  }
}

```

外部参考资料

[1] 「国家集训队」数颜色/维护队列 

13.4.4.4 树上莫队

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan, Linky, yinqf

括号序树上莫队

一般的莫队只能处理线性问题，我们要把树强行压成序列.

我们可以将树的括号序跑下来，把括号序分块，在括号序上跑莫队.

具体怎么做呢?

过程

我们通过 DFS 维护树的括号序列（进入节点 x 时向序列加入 x，离开节点 x 时再次向序列加入 x，序列总长为 $2N$）.

我们令 st_x 为 x 进入的时间，ed_x 为 x 离开的时间.

根据括号序列的性质，对于树上路径 $u \to v$（不妨设 $st_u \le st_v$）:

- 若 $\mathrm{LCA}(u,v)=u$，路径上的节点恰好在区间 $[st_u, st_v]$ 中出现 1 次；不在路径上的节点出现 0 次或 2 次.
- 若 $\mathrm{LCA}(u,v)\neq u$，路径上除 $\mathrm{LCA}(u,v)$ 外的节点恰好在区间 $[ed_u, st_v]$ 中出现 1 次；不在路径上的节点出现 0 次或 2 次. 注意特殊考虑 $\mathrm{LCA}(u,v)$，它出现次数为 0.

因此，进行莫队时，无论遇到的节点是入栈还是出栈，其本质都是改变了该节点在当前区间的出现次数．我们只需要维护每个节点 x 出现了几次，出现次数为 0 或 2 的不在当前维护的路径集合中，出现次数为 1 的在当前维护的路径集合中．因此莫队指针卡到 $[st_u, st_v]$ 时就几乎是我们树上路径 $u \to v$ 的节点集合．

实际实现中，我们可以考虑维护每个节点 x 出现次数对 2 取模的结果，这个就可以指示这个节点 x 有没有出现．那么我们移动指针的时候，无论是加入还是删除更新就相当于翻转这个节点 x 的出现状态．

注意在处理 $\mathrm{LCA} \neq u$ 的查询时，莫队指针移动到位后，我们需要临时手动将 $\mathrm{LCA}(u,v)$ 加入集合（状态翻转），计算并记录该查询的答案，然后再将 $\mathrm{LCA}(u,v)$ 移出集合（再次翻转），以保证不干扰后续指针的移动．

例题

例题 「WC2013」糖果公园[1]

题意：给你一棵树，树上第 i 个点颜色为 c_i，每次询问一条路径 u_i,v_i, 求这条路径上的 $\sum_c val_c \sum_{i=1}^{cnt_c} w_i$

其中：val 表示该颜色的价值，cnt 表示颜色出现的次数，w 表示该颜色出现 i 次后的价值

过程

先把树变成序列，然后每次添加/删除一个点，这个点的对答案的贡献是可以在 $O(1)$ 时间内获得的，即 $val_c \times w_{cnt_{c+1}}$

发现因为他会把起点的子树也扫了一遍，产生多余的贡献，怎么办呢?

因为扫的过程中起点的子树里的点肯定会被扫两次，但贡献为 0.

所以可以开一个 vis 数组，每次扫到点 x，就把 vis_x 异或上 1.

如果 $vis_x = 0$，那这个点的贡献就可以不计.

所以可以用树上莫队来求.

修改的话，加上一维时间维即可，变成带修改树上莫队.

然后因为所包含的区间内可能没有 LCA，对于没有的情况要将多余的贡献删除，然后就完事了.

实现

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

constexpr int MAXN = 200010;

int f[MAXN], g[MAXN], id[MAXN], head[MAXN], cnt, last[MAXN], dep[MAXN],
    fa[MAXN][22], v[MAXN], w[MAXN];
int block, index, n, m, q;
int pos[MAXN], col[MAXN], app[MAXN];
bool vis[MAXN];
long long ans[MAXN], cur;

```

```
struct edge {
  int to, nxt;
} e[MAXN];

int cnt1 = 0, cnt2 = 0;  // 时间戳

struct query {
  int l, r, t, id;

  bool operator<(const query &b) const {
    return (pos[l] < pos[b.l]) || (pos[l] == pos[b.l] && pos[r] < pos[b.r])
||
           (pos[l] == pos[b.l] && pos[r] == pos[b.r] && t < b.t);
  }
} a[MAXN], b[MAXN];

void addedge(int x, int y) {
  e[++cnt] = edge{y, head[x]};
  head[x] = cnt;
}

void dfs(int x) {
  id[f[x] = ++index] = x;
  for (int i = head[x]; i; i = e[i].nxt) {
    if (e[i].to != fa[x][0]) {
      fa[e[i].to][0] = x;
      dep[e[i].to] = dep[x] + 1;
      dfs(e[i].to);
    }
  }
  id[g[x] = ++index] = x;  // 括号序
}

int lca(int x, int y) {
  if (dep[x] < dep[y]) swap(x, y);
  if (dep[x] != dep[y]) {  // 爬到同一高度
    int dis = dep[x] - dep[y];
    for (int i = 20; i >= 0; i--)
      if (dis >= (1 << i)) dis -= 1 << i, x = fa[x][i];
  }
  if (x == y) return x;
  for (int i = 20; i >= 0; i--) {
    if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
  }
  return fa[x][0];
}

void add(int x) {
  if (vis[x])
    cur -= (long long)v[col[x]] * w[app[col[x]]--];
  else
    cur += (long long)v[col[x]] * w[++app[col[x]]];
```

```
  vis[x] ^= 1;
}

// 在时间维上移动
void modify(int x, int t) {
  if (vis[x]) {
    add(x);
    col[x] = t;
    add(x);
  } else
    col[x] = t;
}

int main() {
  scanf("%d%d%d", &n, &m, &q);
  for (int i = 1; i <= m; i++) scanf("%d", &v[i]);
  for (int i = 1; i <= n; i++) scanf("%d", &w[i]);
  for (int i = 1; i < n; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    addedge(x, y);
    addedge(y, x);
  }
  for (int i = 1; i <= n; i++) {
    scanf("%d", &last[i]);
    col[i] = last[i];
  }
  dfs(1);
  for (int j = 1; j <= 20; j++)
    for (int i = 1; i <= n; i++)
      fa[i][j] = fa[fa[i][j - 1]][j - 1];  // 预处理祖先
  int block = pow(index, 2.0 / 3);
  for (int i = 1; i <= index; i++) {
    pos[i] = (i - 1) / block;
  }
  while (q--) {
    int opt, x, y;
    scanf("%d%d%d", &opt, &x, &y);
    if (opt == 0) {
      b[++cnt2].l = x;
      b[cnt2].r = last[x];
      last[x] = b[cnt2].t = y;
    } else {
      if (f[x] > f[y]) swap(x, y);
      a[++cnt1] = query{lca(x, y) == x ? f[x] : g[x], f[y], cnt2, cnt1};
    }
  }
  sort(a + 1, a + cnt1 + 1);
  int L, R, T;  // 指针坐标
  L = R = 0;
  T = 1;
  for (int i = 1; i <= cnt1; i++) {
```

```
118     while (T <= a[i].t) {
119       modify(b[T].l, b[T].t);
120       T++;
121     }
122     while (T > a[i].t) {
123       modify(b[T].l, b[T].r);
124       T--;
125     }
126     while (L > a[i].l) {
127       L--;
128       add(id[L]);
129     }
130     while (L < a[i].l) {
131       add(id[L]);
132       L++;
133     }
134     while (R > a[i].r) {
135       add(id[R]);
136       R--;
137     }
138     while (R < a[i].r) {
139       R++;
140       add(id[R]);
141     }
142     int x = id[L], y = id[R];
143     int llca = lca(x, y);
144     if (x != llca && y != llca) {
145       add(llca);
146       ans[a[i].id] = cur;
147       add(llca);
148     } else
149       ans[a[i].id] = cur;
150   }
151   for (int i = 1; i <= cnt1; i++) {
152     printf("%lld\n", ans[i]);
153   }
154   return 0;
155 }
```

真·树上莫队

上面的树上莫队只是将树转化成了链，下面的才是真正的树上莫队.

由于莫队相关的问题都是模板题，因此实现部分不做太多解释

询问的排序

首先我们知道莫队的是基于分块的算法，所以我们需要找到一种树上的分块方法来保证时间复杂度.

条件:

- 属于同一块的节点之间的距离不超过给定块的大小
- 每个块中的节点不能太多也不能太少
- 每个节点都要属于一个块
- 编号相邻的块之间的距离不能太大

了解了这些条件后，我们看到这样一道题 「SCOI2005」王室联邦[2].

在这道题的基础上我们只要保证最后一个条件就可以解决分块的问题了.

思路

令 lim 为希望块的大小，首先，对于整个树 dfs，当子树的大小大于 lim 时，就将它们分在一块，容易想到：对于根，可能会剩下一些点，于是将这些点分在最后一个块里.

做法：用栈维护当前节点作为父节点访问它的子节点，当从栈顶到父节点的距离大于希望块的大小时，弹出这部分元素分为一块，最后剩余的一块单独作为一块.

最后的排序方法：若第一维时间戳大于第二维，交换它们，按第一维所属块为第一关键字，第二维时间戳为第二关键字排序.

指针的移动

过程

容易想到，我们可以标记被计入答案的点，让指针直接向目标移动，同时取反路径上的点.

但是，这样有一个问题，若指针一开始都在 x 上，显然 x 被标记，当两个指针向同一子节点移动（还有许多情况）时，x 应该不被标记，但实际情况是 x 被标记，因为两个指针分别标记了一次，抵消了.

如何解决呢?

有一个很显然的性质：这些点肯定是某些 LCA，因为 LCA 处才有可能被重复撤销导致撤销失败.

所以我们每次不标记 LCA，到需要询问答案时再将 LCA 标记，然后再撤销.

实现

```
// 取反路径上除LCA以外的所有节点
void move(int x, int y) {
  if (dp[x] < dp[y]) swap(x, y);
  while (dp[x] > dp[y]) update(x), x = fa[x];
  while (x != y) update(x), update(y), x = fa[x], y = fa[y];
  // x!=y保证LCA没被取反
}
```

对于求 LCA，我们可以用树剖，然后我们就可以把分块的步骤放到树剖的第一次 dfs 里面，时间戳也可以直接用第二次 dfs 的 dfs 序.

```
int bl[100002], bls = 0;  // 属于的块，块的数量
unsigned step;            // 块大小
int fa[100002], dp[100002], hs[100002] = {0}, sz[100002] = {0};
// 父节点，深度，重儿子，大小
stack<int> sta;

void dfs1(int x) {
  sz[x] = 1;
  unsigned ss = sta.size();
  for (int i = head[x]; i; i = nxt[i])
    if (ver[i] != fa[x]) {
      fa[ver[i]] = x;
      dp[ver[i]] = dp[x] + 1;
      dfs1(ver[i]);
      sz[x] += sz[ver[i]];
      if (sz[ver[i]] > sz[hs[x]]) hs[x] = ver[i];
      if (sta.size() - ss >= step) {
```

```
        bls++;
        while (sta.size() != ss) bl[sta.top()] = bls, sta.pop();
      }
    }
  sta.push(x);
}

// main
if (!sta.empty()) {
  bls++;  // 这一行可写可不写
  while (!sta.empty()) bl[sta.top()] = bls, sta.pop();
}
```

时间复杂度

重点到了，这里关系到块的大小取值.

设块的大小为 $unit$:

- 对于 x 指针，由于每个块中节点的距离在 $unit$ 左右，每个块中 x 指针移动 $unit^2$ 次 $(unit \times dis_{\max})$，共计 $n \times unit$ 次 $(unit^2 \times (\frac{n}{unit}))$;
- 对于 y 指针，每个块中最多移动 $O(n)$ 次，共计 $\frac{n^2}{unit}$ 次 $(n \times (\frac{n}{unit}))$.

加起来大概在根号处取得最小值（由于树上莫队块的大小不固定，所以不一定要严格按照）.

例题「WC2013」糖果公园

由于多了时间维，块的大小取到 $n^{0.6}$ 的样子就差不多了.

参考代码

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <stack>
using namespace std;

int gi() {
  int x, c, op = 1;
  while (c = getchar(), c < '0' || c > '9')
    if (c == '-') op = -op;
  x = c ^ 48;
  while (c = getchar(), c >= '0' && c <= '9')
    x = (x << 3) + (x << 1) + (c ^ 48);
  return x * op;
}

int head[100002], nxt[200004], ver[200004], tot = 0;

void add(int x, int y) {
  ver[++tot] = y, nxt[tot] = head[x], head[x] = tot;
  ver[++tot] = x, nxt[tot] = head[y], head[y] = tot;
}

int bl[100002], bls = 0;
unsigned step;
int fa[100002], dp[100002], hs[100002] = {0}, sz[100002] = {0}, top[100002],
```

```
                              id[100002];
stack<int> sta;

void dfs1(int x) {
  sz[x] = 1;
  unsigned ss = sta.size();
  for (int i = head[x]; i; i = nxt[i])
    if (ver[i] != fa[x]) {
      fa[ver[i]] = x, dp[ver[i]] = dp[x] + 1;
      dfs1(ver[i]);
      sz[x] += sz[ver[i]];
      if (sz[ver[i]] > sz[hs[x]]) hs[x] = ver[i];
      if (sta.size() - ss >= step) {
        bls++;
        while (sta.size() != ss) bl[sta.top()] = bls, sta.pop();
      }
    }
  sta.push(x);
}

int cnt = 0;

void dfs2(int x, int hf) {
  top[x] = hf, id[x] = ++cnt;
  if (!hs[x]) return;
  dfs2(hs[x], hf);
  for (int i = head[x]; i; i = nxt[i])
    if (ver[i] != fa[x] && ver[i] != hs[x]) dfs2(ver[i], ver[i]);
}

int lca(int x, int y) {
  while (top[x] != top[y]) {
    if (dp[top[x]] < dp[top[y]]) swap(x, y);
    x = fa[top[x]];
  }
  return dp[x] < dp[y] ? x : y;
}

struct qu {
  int x, y, t, id;

  bool operator<(const qu a) const {
    return bl[x] == bl[a.x] ? (bl[y] == bl[a.y] ? t < a.t : bl[y] < bl[a.y])
                            : bl[x] < bl[a.x];
  }
} q[100001];

int qs = 0;

struct ch {
  int x, y, b;
} upd[100001];
```

```
int ups = 0;
long long ans[100001];
int b[100001] = {0};
int a[100001];
long long w[100001];
long long v[100001];
long long now = 0;
bool vis[100001] = {false};

void back(int t) {
  if (vis[upd[t].x]) {
    now -= w[b[upd[t].y]--] * v[upd[t].y];
    now += w[++b[upd[t].b]] * v[upd[t].b];
  }
  a[upd[t].x] = upd[t].b;
}

void change(int t) {
  if (vis[upd[t].x]) {
    now -= w[b[upd[t].b]--] * v[upd[t].b];
    now += w[++b[upd[t].y]] * v[upd[t].y];
  }
  a[upd[t].x] = upd[t].y;
}

void update(int x) {
  if (vis[x])
    now -= w[b[a[x]]--] * v[a[x]];
  else
    now += w[++b[a[x]]] * v[a[x]];
  vis[x] ^= 1;
}

void move(int x, int y) {
  if (dp[x] < dp[y]) swap(x, y);
  while (dp[x] > dp[y]) update(x), x = fa[x];
  while (x != y) update(x), update(y), x = fa[x], y = fa[y];
}

int main() {
  int n = gi(), m = gi(), k = gi();
  step = (int)pow(n, 0.6);
  for (int i = 1; i <= m; i++) v[i] = gi();
  for (int i = 1; i <= n; i++) w[i] = gi();
  for (int i = 1; i < n; i++) add(gi(), gi());
  for (int i = 1; i <= n; i++) a[i] = gi();
  for (int i = 1; i <= k; i++)
    if (gi())
      q[++qs].x = gi(), q[qs].y = gi(), q[qs].t = ups, q[qs].id = qs;
    else
      upd[++ups].x = gi(), upd[ups].y = gi();
```

```
    for (int i = 1; i <= ups; i++) upd[i].b = a[upd[i].x], a[upd[i].x] =
upd[i].y;
  for (int i = ups; i; i--) back(i);
  fa[1] = 1;
  dfs1(1), dfs2(1, 1);
  if (!sta.empty()) {
    bls++;
    while (!sta.empty()) bl[sta.top()] = bls, sta.pop();
  }
  for (int i = 1; i <= n; i++)
    if (id[q[i].x] > id[q[i].y]) swap(q[i].x, q[i].y);
  sort(q + 1, q + qs + 1);
  int x = 1, y = 1, t = 0;
  for (int i = 1; i <= qs; i++) {
    if (x != q[i].x) move(x, q[i].x), x = q[i].x;
    if (y != q[i].y) move(y, q[i].y), y = q[i].y;
    int f = lca(x, y);
    update(f);
    while (t < q[i].t) change(++t);
    while (t > q[i].t) back(t--);
    ans[q[i].id] = now;
    update(f);
  }
  for (int i = 1; i <= qs; i++) printf("%lld\n", ans[i]);
  return 0;
}
```

外部参考资料

[1] 「WC2013」糖果公园  [2] 「SCOI2005」王室联邦 

13.4.4.5 回滚莫队

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan, YOYO-UIAT

引入

有些题目在区间转移时，可能会出现增加或者删除无法实现的问题．在只有增加不可实现或者只有删除不可实现的时候，就可以使用回滚莫队在 $O(n\sqrt{m})$ 的时间内解决问题．回滚莫队的核心思想就是：既然只能实现一个操作，那么就只使用一个操作，剩下的交给回滚解决．

回滚莫队分为只使用增加操作的回滚莫队和只使用删除操作的回滚莫队．以下仅介绍只使用增加操作的回滚莫队，只使用删除操作的回滚莫队和只使用增加操作的回滚莫队只在算法实现上有一点区别，故不再赘述．

例题 JOISC 2014 Day1 历史研究[1]

给你一个长度为 n 的数组 A 和 m 个询问 $(1 \leq n, m \leq 10^5)$，每次询问一个区间 $[L, R]$ 内重要度最大的数字，要求 **输出其重要度**．一个数字 i 重要度的定义为 i 乘上 i 在区间内出现的次数．

在这个问题中，在增加的过程中更新答案是很好实现的，但是在删除的过程中更新答案是不好实现的．因为如果增加会影响答案，那么新答案必定是刚刚增加的数字的重要度，而如果删除过后区间重要度最大的数字改变，我们很难确定新的重要度最大的数字是哪一个．所以，普通的莫队很难解决这个问题．

过程

- 对原序列进行分块，对询问按以左端点所属块编号升序为第一关键字，右端点升序为第二关键字的方式排序.
- 按顺序处理询问：
  - 如果询问左端点所属块 B 和上一个询问左端点所属块的不同，那么将莫队区间的左端点初始化为 B 的右端点加 1，将莫队区间的右端点初始化为 B 的右端点；
  - 如果询问的左右端点所属的块相同，那么直接扫描区间回答询问；
  - 如果询问的左右端点所属的块不同：
    - 如果询问的右端点大于莫队区间的右端点，那么不断扩展右端点直至莫队区间的右端点等于询问的右端点；
    - 不断扩展莫队区间的左端点直至莫队区间的左端点等于询问的左端点；
    - 回答询问；
    - 撤销莫队区间左端点的改动，使莫队区间的左端点回滚到 B 的右端点加 1.

复杂度证明

假设回滚莫队的分块大小是 b：

- 对于左、右端点在同一个块内的询问，可以在 $O(b)$ 时间内计算；
- 对于其他询问，考虑左端点在相同块内的询问，它们的右端点单调递增，移动右端点的时间复杂度是 $O(n)$，而左端点单次询问的移动不超过 b，因为有 $\frac{n}{b}$ 个块，所以总复杂度是 $O(mb+\frac{n^2}{b})$，取 $b=\frac{n}{\sqrt{m}}$ 最优，时间复杂度为 $O(n\sqrt{m})$.

实现

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;
using ll = long long;
constexpr int N = 1e5 + 5;
int n, q;
int x[N], t[N], m;

struct Query {
  int l, r, id;
} Q[N];

int pos[N], L[N], R[N], sz, tot;
int cnt[N], __cnt[N];
ll ans[N];

bool cmp(const Query& A, const Query& B) {
  if (pos[A.l] == pos[B.l]) return A.r < B.r;
  return pos[A.l] < pos[B.l];
}

void build() {
  sz = sqrt(n);
  tot = n / sz;
```

```
  for (int i = 1; i <= tot; i++) {
    L[i] = (i - 1) * sz + 1;
    R[i] = i * sz;
  }
  if (R[tot] < n) {
    ++tot;
    L[tot] = R[tot - 1] + 1;
    R[tot] = n;
  }
}

void Add(int v, ll& Ans) {
  ++cnt[v];
  Ans = max(Ans, 1LL * cnt[v] * t[v]);
}

void Del(int v) { --cnt[v]; }

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> q;
  for (int i = 1; i <= n; i++) cin >> x[i], t[++m] = x[i];
  for (int i = 1; i <= q; i++) cin >> Q[i].l >> Q[i].r, Q[i].id = i;

  build();

  // 对询问进行排序
  for (int i = 1; i <= tot; i++)
    for (int j = L[i]; j <= R[i]; j++) pos[j] = i;
  sort(Q + 1, Q + 1 + q, cmp);

  // 离散化
  sort(t + 1, t + 1 + m);
  m = unique(t + 1, t + 1 + m) - (t + 1);
  for (int i = 1; i <= n; i++) x[i] = lower_bound(t + 1, t + 1 + m, x[i])
- t;

  int l = 1, r = 0, last_block = 0, __l;
  ll Ans = 0, tmp;
  for (int i = 1; i <= q; i++) {
    // 询问的左右端点同属于一个块则暴力扫描回答
    if (pos[Q[i].l] == pos[Q[i].r]) {
      for (int j = Q[i].l; j <= Q[i].r; j++) ++__cnt[x[j]];
      for (int j = Q[i].l; j <= Q[i].r; j++)
        ans[Q[i].id] = max(ans[Q[i].id], 1LL * t[x[j]] * __cnt[x[j]]);
      for (int j = Q[i].l; j <= Q[i].r; j++) --__cnt[x[j]];
      continue;
    }

    // 访问到了新的块则重新初始化莫队区间
    if (pos[Q[i].l] != last_block) {
      while (r > R[pos[Q[i].l]]) Del(x[r]), --r;
```

```
77        while (l < R[pos[Q[i].l]] + 1) Del(x[l]), ++l;
78        Ans = 0;
79        last_block = pos[Q[i].l];
80      }
81
82      // 扩展右端点
83      while (r < Q[i].r) ++r, Add(x[r], Ans);
84      __l = l;
85      tmp = Ans;
86
87      // 扩展左端点
88      while (__l > Q[i].l) --__l, Add(x[__l], tmp);
89      ans[Q[i].id] = tmp;
90
91      // 回滚
92      while (__l < l) Del(x[__l]), ++__l;
93    }
94    for (int i = 1; i <= q; i++) cout << ans[i] << '\n';
95    return 0;
96  }
```

参考资料

- 回滚莫队及其简单运用 | Parsnip's Blog[2]

外部参考资料

[1] JOISC 2014 Day1 历史研究

[2] 回滚莫队及其简单运用 | Parsnip's Blog

13.4.4.6 二维莫队

二维莫队，顾名思义就是每个状态有四个方向可以扩展.

二维莫队每次移动指针要操作一行或者一列的数，具体实现方式与普通的一维莫队类似，这里不再赘述. 这里重点讲块长选定部分.

块长选定

记询问次数为 q，当前矩阵的左上角坐标为 (x_1, y_1)，右下角坐标为 (x_2, y_2)，取块长为 B.

那么指针 x_1 移动了 $\Theta(q \cdot B)$ 次，而指针 y_2 移动了 $\Theta(n^4 \cdot B^{-3})$ 次.

所以只需令 $q \cdot B = n^4 \cdot B^{-3}$，即 $B = n \cdot q^{-\frac{1}{4}}$ 即可.

注意这样计算 B 的结果 **可能为** 0，**注意特判**.

最终，计算部分时间复杂度是 $\Theta(n^2 \cdot q^{\frac{3}{4}})$，加上对询问的排序过程，总时间复杂度为 $\Theta(n^2 \cdot q^{\frac{3}{4}} + q \log q)$.

例题 1

BZOJ 2639 矩形计算[1]

输入一个 $n \times m$ 的矩阵，矩阵的每一个元素都是一个整数，然后有 q 个询问，每次询问一个子矩阵的权值．矩阵的权值是这样定义的，对于一个整数 x，如果它在该矩阵中出现了 p 次，那么它给该矩阵的权值就贡献 p^2．

数据范围：$1 \le n, m \le 200$，$0 \le q \le 10^5$，| 矩阵元素大小 | $\le 2 \times 10^9$．

解题思路

先离散化，二维莫队时用一个数组记录每个数当前出现的次数即可．

示例代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;

int n, m, q, a[201][201];
long long ans[100001];
int disc[250001], cntdisc;  // 离散化用

int blocklen, counts[40001];
long long now;

struct Question {
  int x1, y1, x2, y2, qid;

  bool operator<(Question tmp) const {
    if (x1 / blocklen != tmp.x1 / blocklen) return x1 < tmp.x1;
    if (y1 / blocklen != tmp.y1 / blocklen) return y1 < tmp.y1;
    if (x2 / blocklen != tmp.x2 / blocklen) return x2 < tmp.x2;
    return y2 < tmp.y2;
  }
} Q[100001];

int Qcnt;

void mo_algo_row(int id, int val, int Y1, int Y2) {
  for (int i = Y1; i <= Y2; i++)
    now -= (long long)counts[a[id][i]] * counts[a[id][i]],
        counts[a[id][i]] += val,
        now += (long long)counts[a[id][i]] * counts[a[id][i]];
}

void mo_algo_column(int id, int val, int X1, int X2) {
  for (int i = X1; i <= X2; i++)
    now -= (long long)counts[a[i][id]] * counts[a[i][id]],
        counts[a[i][id]] += val,
        now += (long long)counts[a[i][id]] * counts[a[i][id]];
```

```
}

void mo_algo() {
  blocklen = pow(n * m, 0.5) / pow(q, 0.25);
  if (blocklen < 1) blocklen = 1;
  sort(Q + 1, Q + 1 + Qcnt);

  int X1 = 1, Y1 = 1, X2 = 0, Y2 = 0;
  for (int i = 1; i <= Qcnt; i++) {
    while (X1 > Q[i].x1) mo_algo_row(--X1, 1, Y1, Y2);
    while (X2 < Q[i].x2) mo_algo_row(++X2, 1, Y1, Y2);
    while (Y1 > Q[i].y1) mo_algo_column(--Y1, 1, X1, X2);
    while (Y2 < Q[i].y2) mo_algo_column(++Y2, 1, X1, X2);
    while (X1 < Q[i].x1) mo_algo_row(X1++, -1, Y1, Y2);
    while (X2 > Q[i].x2) mo_algo_row(X2--, -1, Y1, Y2);
    while (Y1 < Q[i].y1) mo_algo_column(Y1++, -1, X1, X2);
    while (Y2 > Q[i].y2) mo_algo_column(Y2--, -1, X1, X2);
    ans[Q[i].qid] = now;
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++) cin >> a[i][j], disc[++cntdisc] = a[i][j];
  sort(disc + 1, disc + 1 + cntdisc);
  cntdisc = unique(disc + 1, disc + cntdisc + 1) - disc - 1;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      a[i][j] = lower_bound(disc + 1, disc + 1 + cntdisc, a[i][j]) - disc;
  cin >> q;
  for (int i = 1; i <= q; i++) {
    int x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;
    if (x1 > x2) swap(x1, x2);
    if (y1 > y2) swap(y1, y2);
    Q[++Qcnt] = {x1, y1, x2, y2, i};
  }

  mo_algo();
  for (int i = 1; i <= q; ++i) cout << ans[i] << '\n';
  return 0;
}
```

例题 2

洛谷 P1527 [国家集训队] 矩阵乘法[2]

给你一个 $n \times n$ 的矩阵，q 次询问，每次询问一个子矩形的第 k 小数.

数据范围：$1 \leq n \leq 500$，$1 \leq q \leq 6 \times 10^4$，$0 \leq a_{i,j} \leq 10^9$.

首先和上一题一样，需要离散化整个矩阵．但是需要注意，本题除了需要对数值进行分块，还需要对数值的值域进行分块，才能求出答案．

这里还需要用到奇偶化排序进行优化，具体内容请见 普通莫队算法→13.4.4.2@p3209．

对于本题而言，时间限制不那么宽，注意代码常数的处理．取的块长计算值普遍较小，n, q 都取最大值时块长大约在 11 左右，可以直接设定为常数来节约代码耗时．

示例代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int n, q, a[501][501], ans[60001];
int disc[250001], cntdisc;  // 离散化用
int nn;

int blockId[501], blocklen;                // 分块
int rangeblockId[250001], rangeblocklen;  // 值域分块
int counts[250001], countsum[501];         // 该值次数及值域块总和

struct Position {
  int x, y;
};

vector<Position> pos[250001];

struct Question {
  int x1, y1, x2, y2, k, qid;

  bool operator<(Question tmp) const {
    if (blockId[x1] != blockId[tmp.x1]) return blockId[x1] < blockId[tmp.x1];
    if (blockId[y1] != blockId[tmp.y1])
      return blockId[x1] & 1 ? y1 < tmp.y1 : y1 > tmp.y1;
    if (blockId[y2] != blockId[tmp.y2])
      return blockId[y1] & 1 ? y2 < tmp.y2 : y2 > tmp.y2;
    else
      return blockId[y2] & 1 ? x2 < tmp.x2 : x2 > tmp.x2;
  }
} Q[60001];

int Qcnt;

void mo_algo() {
  blocklen = 11;
  for (int i = 1; i <= n; ++i) blockId[i] = (i - 1) / blocklen + 1;
  rangeblocklen = n + 1;
  for (int i = 1; i <= nn; ++i) rangeblockId[i] = (i - 1) / rangeblocklen + 1;
  counts[a[1][1]] = countsum[rangeblockId[a[1][1]]] = 1;
  sort(Q + 1, Q + 1 + Qcnt);
```

```
43
44    int L = 1, R = 1, D = 1, U = 1;
45    for (int i = 1; i <= q; ++i) {
46      while (R < Q[i].y2) {
47        ++R;
48        for (int i = U; i <= D; ++i)
49          ++counts[a[i][R]], ++countsum[rangeblockId[a[i][R]]];
50      }
51      while (L > Q[i].y1) {
52        --L;
53        for (int i = U; i <= D; ++i)
54          ++counts[a[i][L]], ++countsum[rangeblockId[a[i][L]]];
55      }
56      while (D < Q[i].x2) {
57        ++D;
58        for (int i = L; i <= R; ++i)
59          ++counts[a[D][i]], ++countsum[rangeblockId[a[D][i]]];
60      }
61      while (U > Q[i].x1) {
62        --U;
63        for (int i = L; i <= R; ++i)
64          ++counts[a[U][i]], ++countsum[rangeblockId[a[U][i]]];
65      }
66      while (R > Q[i].y2) {
67        for (int i = U; i <= D; ++i)
68          --counts[a[i][R]], --countsum[rangeblockId[a[i][R]]];
69        --R;
70      }
71      while (L < Q[i].y1) {
72        for (int i = U; i <= D; ++i)
73          --counts[a[i][L]], --countsum[rangeblockId[a[i][L]]];
74        ++L;
75      }
76      while (D > Q[i].x2) {
77        for (int i = L; i <= R; ++i)
78          --counts[a[D][i]], --countsum[rangeblockId[a[D][i]]];
79        --D;
80      }
81      while (U < Q[i].x1) {
82        for (int i = L; i <= R; ++i)
83          --counts[a[U][i]], --countsum[rangeblockId[a[U][i]]];
84        ++U;
85      }
86      int res = 1, cnt = 0;
87      while (cnt + countsum[res] < Q[i].k && res <= rangeblockId[nn])
88        cnt += countsum[res], ++res;
89      res = (res - 1) * rangeblocklen + 1;
90      while (cnt + counts[res] < Q[i].k && res <= nn) cnt += counts[res],
  ++res;
91      ans[Q[i].qid] = disc[res];
92    }
93  }
```

```cpp

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> q;
  nn = n * n;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j) {
      int x;
      cin >> x;
      a[i][j] = disc[++cntdisc] = x;
    }
  sort(disc + 1, disc + 1 + cntdisc);
  cntdisc = unique(disc + 1, disc + cntdisc + 1) - disc - 1;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j)
      a[i][j] = lower_bound(disc + 1, disc + 1 + cntdisc, a[i][j]) - disc;
  for (int i = 1; i <= q; ++i) {
    int x1, y1, x2, y2, k;
    cin >> x1 >> y1 >> x2 >> y2 >> k;
    Q[++Qcnt] = {x1, y1, x2, y2, k, i};
  }

  mo_algo();
  for (int i = 1; i <= q; ++i) cout << ans[i] << '\n';
  return 0;
}
```

外部参考资料

[1] BZOJ 2639 矩形计算



[2] 洛谷 P1527 [国家集训队] 矩阵乘法



13.4.4.7 莫队二次离线

Authors: Lyccrius, AtomAlpaca

综述

有时我们会遇见一些题目，这些题目看起来很适合使用莫队算法处理，但是他们的单次转移并非是 $O(1)$ 的，这时直接使用莫队即使调整块长，也会导致复杂度不正确.

此时，如果每次转移对答案的贡献可以进行差分，我们就可以将这些转移拆开离线下来，使用其它算法批量处理.

我们用 $f(x,l,r)$ 表示 x 关于 $[l,r]$ 产生的贡献.

如我们在将当前区间 $[l,r]$ 扩展到 $[l,r+1]$ 时，我们要求的是 $f(a_{r+1},l,r)$. 如果可以差分，我们可以将其写成 $f(a_{r+1},1,r)-f(a_{r+1},1,l-1)$，其中第一项我们可以对于每个 r 都预处理出来，后一项我们可以把每个这样的项都离线存到对应的 $l-1$ 上，然后从小到大枚举并扫描线处理. 其它几个转移的方向也都可以类似地处理.

这一在莫队这个离线算法上，将转移再次离线处理的算法叫做莫队二次离线.

我们结合实际题目来理解.

例题

Luogu P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology II[1]

给你一个长为 n 的序列 a，m 次询问，每次查询一个区间的逆序对数.

数据范围：$1 \le n, m \le 10^5$，$0 \le a_i \le 10^9$.

直接莫队每次转移至少是 $O(\log n)$ 的，观察可以发现我们每次转移要求的信息是「一个数在某个区间内的排名」，而这一信息关于区间可以差分，因此考虑二次离线.

我们将 $[l, r]$ 拓展到 $[l, r+1]$，要求的是 在 $[l, r]$ 区间内有多少比 a_{r+1} 大的数. 我们用 $f(x, r)$ 表示 $[1, r]$ 区间内比 a_x 大的数，$g(x, r)$ 表示 $[1, r]$ 区间内比 a_x 小的数. 则答案的变化量可以写作 $f(r+1, r) - f(r+1, l-1)$.

同理 $[l, r]$ 缩小至 $[l, r-1]$ 的变化量可以写作 $-f(r, r-1) + f(r, l-1)$；$[l, r]$ 拓展到 $[l-1, r]$ 可以写作 $g(l-1, r) - g(l-1, l-2)$,$[l, r]$ 缩小至 $[l+1, r]$ 可以写作 $-g(l, r) + g(l, l-1)$.

对于这些式子中的 $f(x, x-1)$ 和 $g(x, x-1)$，我们可以使用树状数组提前 $O(n \log n)$ 预处理出来；对于其余的 $f(x, p)$ 和 $g(x, p)$ 项，我们将 x 离线存在 l 进行批量处理.

这里有一个优化空间的技巧：我们发现处理每个询问时，离线到 p 上的 x 都是连续的一段，因此我们不需要把每次移动都存下，只需要存下调整的一段即可. 这样我们的空间可以从总次数 $O(n\sqrt{m})$ 下降到询问数 $O(m)$.

现在我们来处理我们二次离线下来的问题：向一个集合中加入数，查询一个数在集合中的排名. 莫队总共移动端点的次数是 $O(n\sqrt{m})$ 的，而数组总共只有 $O(n)$ 的长度，因此我们考虑使用 $O(\sqrt{n})$ 插入、$O(1)$ 查询的值域分块解决这个问题.

至此，我们在 $O(n\sqrt{m} + n\sqrt{n})$ 的时间复杂度和 $O(n + m)$ 的空间复杂度下解决了此题.

最后值得注意的是，我们求得的是每次的答案变化量而非答案本身，需要对其进行前缀和才能得到最终的答案.

示例代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstring>
#include <iostream>
#include <vector>

constexpr int MAX = 1e5 + 5;
constexpr int MAXX = 4e2 + 5;
constexpr int MX = 1e5;
typedef long long ll;

int n, m, sz, _sz;
int a[MAX], b[MAX], bl[MAX], st[MAXX], ed[MAXX], b1[MAXX], b2[MAX], p1[MAX],
    p2[MAX];
ll ans[MAX];

struct N {
  int l, r, id, x, y;
};

```

```
std::vector<N> v[MAX];

struct Q {
  int l, r, id;
  ll ans = 0;
} q[MAX];

bool cmp(Q a, Q b) {
  if (a.l / _sz != b.l / _sz) {
    return a.l < b.l;
  }
  return ((a.l / _sz) & 1) ? a.r < b.r : a.r > b.r;
}

void init() {
  sz = sqrt(MX);
  for (int i = 1; i <= sz; ++i) {
    st[i] = (MX / sz) * (i - 1) + 1;
    ed[i] = (MX / sz) * i;
  }
  ed[sz] = MX;
  for (int i = 1; i <= sz; ++i) {
    for (int j = st[i]; j <= ed[i]; ++j) {
      bl[j] = i;
    }
  }
}

void clear() {
  memset(b1, 0, sizeof(b1));
  memset(b2, 0, sizeof(b2));
}

void mdf(int x) {
  for (int i = bl[x]; i <= sz; ++i) {
    ++b1[i];
  }
  for (int i = x; i <= ed[bl[x]]; ++i) {
    ++b2[i];
  }
}

int qry(int x) {
  if (!x) {
    return 0;
  }
  return b1[bl[x] - 1] + b2[x];
}

int main() {
  scanf("%d%d", &n, &m);
  init();
```

```
  _sz = sqrt(m);
  for (int i = 1; i <= n; ++i) {
    scanf("%d", &a[i]);
    b[i] = a[i];
  }
  std::sort(b + 1, b + n + 1);
  int len = std::unique(b + 1, b + n + 1) - b - 1;
  for (int i = 1; i <= n; ++i) {
    a[i] = std::lower_bound(b + 1, b + len + 1, a[i]) - b;
  }
  for (int i = 1; i <= n; ++i) {
    p1[i] = qry(a[i] - 1);
    p2[i] = qry(n) - qry(a[i]);
    mdf(a[i]);
  }
  for (int i = 1; i <= m; ++i) {
    scanf("%d%d", &q[i].l, &q[i].r);
    q[i].id = i;
  }
  std::sort(q + 1, q + m + 1, cmp);
  clear();
  for (int i = 1, L = 1, R = 0; i <= m; ++i) {
    int l = q[i].l, r = q[i].r;
    if (L > l) {
      v[R].push_back({l, L - 1, i, 1, 0});
    }
    while (L > l) {
      --L;
      q[i].ans -= p1[L];
    }
    if (R < r) {
      v[L - 1].push_back({R + 1, r, i, -1, 1});
    }
    while (R < r) {
      ++R;
      q[i].ans += p2[R];
    }
    if (L < l) {
      v[R].push_back({L, l - 1, i, -1, 0});
    }
    while (L < l) {
      q[i].ans += p1[L];
      ++L;
    }
    if (R > r) {
      v[L - 1].push_back({r + 1, R, i, 1, 1});
    }
    while (R > r) {
      q[i].ans -= p2[R];
      --R;
    }
  }
```

```
  for (int i = 1; i <= n; ++i) {
    mdf(a[i]);
    for (N j : v[i]) {
      for (int k = j.l; k <= j.r; ++k) {
        if (j.y == 0) {
          q[j.id].ans += j.x * qry(a[k] - 1);
        } else {
          q[j.id].ans += j.x * (i - qry(a[k]));
        }
      }
    }
  }
  for (int i = 2; i <= m; ++i) {
    q[i].ans += q[i - 1].ans;
  }
  for (int i = 1; i <= m; ++i) {
    ans[q[i].id] = q[i].ans;
  }
  for (int i = 1; i <= m; ++i) {
    printf("%lld\n", ans[i]);
  }
}

```

Luogu P5501 [LnOI2019] 来者不拒，去者不追[2]

多次询问区间中 $[l,r]$ 中所有数的「Abbi 值」之和.

Abbi 值定义为：若 a_i 在询问区间 $[l,r]$ 中是第 k 小，那么它的「Abbi 值」等于 ka_i.

我们不妨令 $f(x,r)$ 是 $[1,r]$ 中比 a_x 大的数之和，$g(x,r)$ 是 $[1,r]$ 中比 a_x 大的数的数量，那么我们向右移动右端点时，产生的贡献为 $f(r,r-1)-f(r,l-1)+a_r(r-l+1-(g(r,r-1)-g(r,l-1)))$，其它几个方向可同理写出，在此不加赘述.

上式中的 $f(r,r-1)$ 和 $g(r,r-1)$ 依旧可以进行预处理，其余的离线到另一端点上，进行扫描线处理. 不难发现我们要处理的依然是 $O(n)$ 次插入、$O(n\sqrt{m})$ 次询问排名的问题，因此同样使用值域分块解决.

示例代码

```
#include <algorithm>
#include <cmath>
#include <cstring>
#include <iostream>
#include <vector>

constexpr int MAX = 1e5 + 5;
constexpr int MAXX = 4e2 + 5;
constexpr int MX = 1e5;

typedef long long ll;

ll n, m, sz, _sz, sm;
```

```
ll st[MAXX], ed[MAXX], bl[MAX], a[MAX], ans[MAX], sm1[MAXX], sm3[MAXX],
    sm2[MAX], sm4[MAX], s[MAX], pre[MAX];

void init() {
  sz = sqrt(MX);
  for (int i = 1; i <= sz; ++i) {
    st[i] = (MX / sz) * (i - 1) + 1;
    ed[i] = (MX / sz) * i;
  }
  ed[sz] = MX;
  for (int i = 1; i <= sz; ++i) {
    for (int j = st[i]; j <= ed[i]; ++j) {
      bl[j] = i;
    }
  }
}

void mdf(ll x) {
  sm += x;
  for (int i = bl[x]; i <= sz; ++i) {
    ++sm1[i];
    sm3[i] += x;
  }
  for (int i = x; i <= ed[bl[x]]; ++i) {
    ++sm2[i];
    sm4[i] += x;
  }
}

ll qry(ll x) {
  if (!x) {
    return 0;
  }
  return sm1[bl[x] - 1] + sm2[x];
}

ll _qry(ll x) {
  if (!x) {
    return 0;
  }
  return sm3[bl[x] - 1] + sm4[x];
}

void clear() {
  sm = 0;
  memset(sm1, 0, sizeof(sm1));
  memset(sm2, 0, sizeof(sm2));
  memset(sm3, 0, sizeof(sm3));
  memset(sm4, 0, sizeof(sm4));
}

struct Q {
```

```
  ll l, r, id, ans;
} q[MAX];

struct N {
  ll l, r, id, x;
};

std::vector<N> v[MAX];

bool cmp(Q a, Q b) {
  if (a.l / _sz != b.l / _sz) {
    return a.l < b.l;
  } else {
    return ((a.l / _sz) & 1) ? a.r < b.r : a.r > b.r;
  }
}

int main() {
  scanf("%lld%lld", &n, &m);
  _sz = n / sqrt(m);
  init();
  for (int i = 1; i <= n; ++i) {
    scanf("%lld", &a[i]);
    s[i] = s[i - 1] + a[i];
  }
  for (int i = 1; i <= n; ++i) {
    pre[i] = qry(a[i] - 1) * a[i] + sm - _qry(a[i]);
    mdf(a[i]);
  }
  clear();
  for (int i = 1; i <= m; ++i) {
    scanf("%lld%lld", &q[i].l, &q[i].r);
    q[i].id = i;
  }
  std::sort(q + 1, q + m + 1, cmp);
  for (int i = 1, L = 1, R = 0; i <= m; ++i) {
    int l = q[i].l, r = q[i].r;
    if (L > l) {
      v[R].push_back({l, L - 1, i, 1});
    }
    while (L > l) {
      --L;
      q[i].ans -= pre[L];
    }
    if (R < r) {
      v[L - 1].push_back({R + 1, r, i, -1});
    }
    while (R < r) {
      ++R;
      q[i].ans += pre[R];
    }
    if (L < l) {
```

```
118       v[R].push_back({L, l - 1, i, -1});
119     }
120     while (L < l) {
121       q[i].ans += pre[L];
122       ++L;
123     }
124     if (R > r) {
125       v[L - 1].push_back({r + 1, R, i, 1});
126     }
127     while (R > r) {
128       q[i].ans -= pre[R];
129       --R;
130     }
131   }
132   for (int i = 1; i <= n; ++i) {
133     mdf(a[i]);
134     for (N j : v[i]) {
135       for (int k = j.l; k <= j.r; ++k) {
136         q[j.id].ans += 1ll * j.x * (qry(a[k] - 1) * a[k] + sm - _qry(a[k]));
137       }
138     }
139   }
140   for (int i = 2; i <= m; ++i) {
141     q[i].ans += q[i - 1].ans;
142   }
143   for (int i = 1; i <= m; ++i) {
144     ans[q[i].id] = q[i].ans + s[q[i].r] - s[q[i].l - 1];
145   }
146   for (int i = 1; i <= m; ++i) {
147     printf("%lld\n", ans[i]);
148   }
149   return 0;
150 }
151
```

外部参考资料

[1] Luogu P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology II

[2] Luogu P5501 [LnOI2019] 来者不拒，去者不追

13.4.4.8 莫队配合 bitset

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan

bitset 常用于常规数据结构难以维护的判定、统计问题，而莫队可以维护常规数据结构难以维护的区间信息．把两者结合起来使用可以同时利用两者的优势．

例题 「Ynoi2016」掉进兔子洞[1]

本题刚好符合上面提到的莫队配合 bitset 的特征．不难想到我们可以分别用 bitset 存储每一个区间内的出现过的所有权值，一组询问的答案即所有区间的长度和减去三者的并集元素个数 $\times$ 3.

但是在莫队中使用 bitset 也需要针对 bitset 的特性调整算法：

1. bitset 不能很好地处理同时出现多个权值的情况. 我们可以把当前元素离散化后的权值与当前区间的出现次数之和作为往 bitset 中插入的对象.
2. 我们平常使用莫队时, 可能会不注意 4 种移动指针的方法顺序, 所以指针移动的过程中可能会出现区间的左端点在右端点右边, 区间长度为负值的情况, 导致元素的个数为负数. 这在其他情况下并没有什么影响, 但是本题中在 bitset 中插入的元素与元素个数有关, 所以我们需要注意 4 种移动指针的方法顺序, 将左右指针分别往左边和右边移动的语句写在前面, 避免往 bitset 中插入负数.
3. 虽然 bitset 用空间小, 但是仍然难以承受 $10^5 \times 10^5$ 的数据规模. 所以我们需要将询问划分成常数块分别处理, 保证空间刚好足够的情况下时间复杂度不变.

参考代码

```cpp
#include <algorithm>
#include <bitset>
#include <cmath>
#include <cstring>
#include <iostream>
using namespace std;
constexpr int N = 100005, M = N / 3 + 10;
int n, m, maxn;
int a[N], ans[M], cnt[N];
bitset<N> sum[M], now;

struct query {
  int l, r, id;

  bool operator<(const query& x) const {
    if (l / maxn != x.l / maxn) return l < x.l;
    return (l / maxn) & 1 ? r < x.r : r > x.r;
  }
} q[M * 3];

void static_set() {
  static int tmp[N];
  memcpy(tmp, a, sizeof(a));
  sort(tmp + 1, tmp + n + 1);
  for (int i = 1; i <= n; i++)
    a[i] = lower_bound(tmp + 1, tmp + n + 1, a[i]) - tmp;
}

void add(int x) {
  now.set(x + cnt[x]);
  cnt[x]++;
}

void del(int x) {
  cnt[x]--;
  now.reset(x + cnt[x]);
}

void solve() {
  int cnt = 0, tot = 0;
  now.reset();
```

```cpp
  for (tot = 0; tot < M - 5 && m; tot++) {
    m--;
    ans[tot] = 0;
    sum[tot].set();
    for (int j = 0; j < 3; j++) {
      cin >> q[cnt].l >> q[cnt].r;
      q[cnt].id = tot;
      ans[tot] += q[cnt].r - q[cnt].l + 1;
      cnt++;
    }
  }
  sort(q, q + cnt);
  for (int i = 0, l = 1, r = 0; i < cnt; i++) {
    while (l > q[i].l) add(a[--l]);
    while (r < q[i].r) add(a[++r]);
    while (l < q[i].l) del(a[l++]);
    while (r > q[i].r) del(a[r--]);
    sum[q[i].id] &= now;
  }
  for (int i = 0; i < tot; i++)
    cout << ans[i] - (int)sum[i].count() * 3 << '\n';
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  static_set();
  maxn = sqrt(n);
  solve();
  memset(cnt, 0, sizeof(cnt));
  solve();
  memset(cnt, 0, sizeof(cnt));
  solve();
  return 0;
}
```

习题

- 小清新人渣的本愿[2]
- 「Ynoi2017」由乃的玉米田[3]
- 「Ynoi2011」WBLT[4]

外部参考资料

[1] 「Ynoi2016」掉进兔子洞 

[2] 小清新人渣的本愿 

[3] 「Ynoi2017」由乃的玉米田 

[4] 「Ynoi2011」WBLT 

13.5 分数规划

Authors: greyqz, Ir1d, hsfzLZH1, huaruoji, banglee13

分数规划用来求一个分式的极值．其形式化表述是，给出 a_i 和 b_i，求一组 $w_i \in \{0,1\}$，最小化或最大化

$$\frac{\sum_{i=1}^{n} a_i \times w_i}{\sum_{i=1}^{n} b_i \times w_i}$$

通俗来讲，这类问题类似于：每种物品有两个权值 a 和 b，选出若干个物品使得 $\frac{\sum a}{\sum b}$ 最小或最大．

一般分数规划问题还会有一些特殊的限制，比如「分母至少为 W」．

求解

二分法

分数规划问题的通用方法是二分答案法．假设当前二分到的答案为 mid，则一组满足条件的 $\{w_i\}$ 会让权值大于等于 mid．根据这一条件列不等式并变形

$$\begin{aligned} &\frac{\sum a_i \times w_i}{\sum b_i \times w_i} \geq mid \\ \Longrightarrow &\sum a_i \times w_i - mid \times \sum b_i \cdot w_i \geq 0 \\ \Longrightarrow &\sum w_i \times (a_i - mid \times b_i) \geq 0 \end{aligned}$$

那么只要求出不等号左边的式子的最大值就行了．如果最大值比 0 要大，说明 mid 是可行的，否则不可行．分数规划的主要难点就在于如何求 $\sum w_i \times (a_i - mid \times b_i)$ 的最大值或最小值．

Dinkelbach 算法

Dinkelbach 算法[1]的大概思想是每次用上一轮的答案当做新的 L 来输入，不断地迭代，直至答案收敛．

例题

LOJ 149 01 分数规划[2]

有 n 个物品，每个物品有两个权值 a 和 b．求一组 $w_i \in \{0,1\}$，满足 w_i 中恰好有 k 个 1，最大化 $\frac{\sum a_i \times w_i}{\sum b_i \times w_i}$ 的值．

解法

把 $a_i - mid \times b_i$ 作为第 i 个物品的权值，贪心地选权值前 k 大的物品．若权值和大于 0 则可行，否则不可行．

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <functional>
using namespace std;

constexpr int N = 100000 + 10;
constexpr double eps = 1e-6;

```

[1]Dinkelbach, Werner. "On nonlinear fractional programming." Management science 13.7 (1967): 492-498.[1]

```cpp
int n, k;
int a[N], b[N];
double c[N];

bool check(double mid) {
  double s = 0;
  for (int i = 1; i <= n; i++) c[i] = a[i] - b[i] * mid;
  // 将权值从大到小排序
  sort(c + 1, c + n + 1, greater<double>());
  for (int i = 1; i <= k; ++i)  // 选择前 k 个物品
    s += c[i];
  return s >= 0;
}

int main() {
  scanf("%d %d", &n, &k);
  for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
  for (int i = 1; i <= n; ++i) scanf("%d", &b[i]);
  double L = 0, R = 1;
  while (R - L > eps) {
    double mid = (L + R) / 2;
    if (check(mid))  // mid 可行，答案比 mid 大
      L = mid;
    else  // mid 不可行，答案比 mid 小
      R = mid;
  }
  printf("%.6lf\n", L);
  return 0;
}

```

洛谷 4377 Talent Show G[3]

有 n 个物品，每个物品有两个权值 a 和 b.

你需要确定一组 $w_i \in \{0,1\}$，使得 $\frac{\sum w_i \times a_i}{\sum w_i \times b_i}$ 最大.

要求 $\sum w_i \times b_i \geq W$.

解法

本题多了分母至少为 W 的限制，因此无法再使用上一题的贪心算法.

可以考虑 01 背包. 把 b_i 作为第 i 个物品的重量，$a_i - mid \times b_i$ 作为第 i 个物品的价值，然后问题就转化为背包了. 那么 $dp[n][W]$ 就是最大值.

在 DP 过程中，物品重量和可能超过 W，此时直接视为 W 即可.

参考代码

```cpp
#include <algorithm>
#include <cstdio>
```

```cpp
using namespace std;

constexpr int MAXN = 250 + 10;
constexpr int MAXW = 1000 + 10;
constexpr double eps = 1e-6;

int n, W;
int w[MAXN], t[MAXN];
double f[MAXW];

bool check(double mid) {
  double s = 0;
  for (int i = 1; i <= W; i++) f[i] = -1e9;
  for (int i = 1; i <= n; i++)
    for (int j = W; j >= 0; j--) {
      int k = min(W, j + w[i]);
      f[k] = max(f[k], f[j] + t[i] - mid * w[i]);
    }
  return f[W] >= 0;
}

int main() {
  scanf("%d %d", &n, &W);
  double L = 0, R = 0;
  for (int i = 1; i <= n; ++i) {
    scanf("%d %d", &w[i], &t[i]);
    R += t[i];
  }
  while (R - L > eps) {
    double mid = (L + R) / 2;
    if (check(mid))
      L = mid;
    else
      R = mid;
  }
  printf("%d\n", (int)(L * 1000));
  return 0;
}

```

POJ2728 Desert King[4]

每条边有两个权值 a_i 和 b_i，求一棵生成树 T 使得 $\frac{\sum_{e\in T} a_e}{\sum_{e\in T} b_e}$ 最小.

解法

把 $a_i - mid \times b_i$ 作为每条边的权值，那么最小生成树就是最小值. 本题中需要求解一个完全图中的最小生成树，应利用 Prim 算法求解.

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

const int N = 1000 + 10;
const double eps = 1e-5;

int n;
double d[N][N], c[N][N], dis[N];
int x[N], y[N], z[N];
bool vis[N];

bool ok(double m) {
  double ans = 0;
  for (int i = 1; i <= n; i++) vis[i] = false;
  dis[1] = 0;
  for (int i = 2; i <= n; i++) dis[i] = 1e18;
  for (int i = 1; i <= n; i++) {
    double mn = 1e18;
    int pt = -1;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && mn > dis[j]) {
        pt = j;
        mn = dis[j];
      }
    if (!~pt) break;
    vis[pt] = true;
    ans += mn;
    for (int j = 1; j <= n; j++)
      if (j != pt) dis[j] = min(dis[j], c[pt][j] - m * d[pt][j]);
  }
  return ans >= 0;
}

int main() {
  while (scanf("%d", &n) == 1) {
    if (n == 0) break;
    for (int i = 1; i <= n; i++) scanf("%d %d %d", &x[i], &y[i], &z[i]);
    for (int i = 1; i <= n; i++)
      for (int j = i + 1; j <= n; j++) {
        d[i][j] = d[j][i] =
            sqrt((x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]));
        c[i][j] = c[j][i] = abs(z[i] - z[j]);
      }
    double l = 0, r = 10000000;
    while (r - l > eps) {
      double m = (l + r) / 2;
      if (ok(m))
        l = m;
```

```
51        else
52          r = m;
53      }
54      printf("%.3f\n", l);
55    }
56    return 0;
57  }
58
```

[HNOI2009] 最小圈[5]

每条边的边权为 w，求一个环 C 使得 $\frac{\sum_{e\in C} w}{|C|}$ 最小.

解法

把 $a_i - mid$ 作为边权，那么权值最小的环就是最小值.

因为我们只需要判最小值是否小于 0，所以只需要判断图中是否存在负环即可.

另外本题存在一种复杂度 $O(nm)$ 的算法，如果有兴趣可以阅读 这篇文章[6].

参考代码

```
1  #include <algorithm>
2  #include <cstdio>
3  #include <tuple>
4  #include <vector>
5  using namespace std;
6
7  constexpr int N = 3000 + 10;
8  constexpr double eps = 1e-9;
9
10 int n, m;
11 double dis[N];
12 vector<pair<int, double>> g[N];
13
14 bool check(double mid) {  // 如果有负环返回 true
15   bool flag = false;
16   dis[0] = 0;
17   for (int i = 1; i <= n; ++i) dis[i] = 1e9;
18   for (int t = 1; t <= n; ++t) {
19     flag = false;
20     for (int u = 0; u <= n; ++u)
21       for (auto vw : g[u]) {
22         int v;
23         double w;
24         tie(v, w) = vw;
25         if (dis[v] > dis[u] + w - mid) {
26           dis[v] = dis[u] + w - mid;
27           flag = true;
28         }
```

```
      }
    if (!flag) break;
  }
  return flag;
}

int main() {
  scanf("%d %d", &n, &m);
  for (int i = 1; i <= m; ++i) {
    int u, v;
    double w;
    scanf("%d %d %lf", &u, &v, &w);
    g[u].push_back({v, w});
  }
  for (int i = 1; i <= n; i++) g[0].push_back({i, 0});
  double L = -1e7, R = 1e7;
  while (R - L > eps) {
    double mid = (L + R) / 2;
    if (check(mid))
      R = mid;
    else
      L = mid;
  }
  printf("%.8lf\n", L);
  return 0;
}

```

习题

- JSOI2016 最佳团体[7]
- SDOI2017 新生舞会[8]
- UVa1389 Hard Life[9]
- 洛谷 P2868 [USACO07DEC] Sightseeing Cows G[10]
- AtCoder Beginner Contest 324 F - Beautiful Path[11]

参考资料与注释

[1] Dinkelbach, Werner. "On nonlinear fractional programming." Management science 13.7 (1967): 492-498.

[2] LOJ 149 01 分数规划

[3] 洛谷 4377 Talent Show G

[4] POJ2728 Desert King

[5] [HNOI2009] 最小圈

[6] 这篇文章

[7] JSOI2016 最佳团体

[8] SDOI2017 新生舞会

[9] UVa1389 Hard Life

[10] 洛谷 P2868 [USACO07DEC] Sightseeing Cows G

[11] AtCoder Beginner Contest 324 F - Beautiful Path

13.6 随机化

13.6.1 随机函数

概述

要想使用随机化技巧，前提条件是能够快速生成随机数．本文将介绍生成随机数的常见方法．

随机数与伪随机数

说一个单独的数是「随机数」是无意义的，所以以下我们都默认讨论「随机数列」，即使提到「随机数」，指的也是「随机数列中的一个元素」．

现有的计算机的运算过程都是确定性的，因此，仅凭借算法来生成真正 **不可预测**、**不可重复** 的随机数列是不可能的．

然而在绝大部分情况下，我们都不需要如此强的随机性，而只需要所生成的数列在统计学上具有随机数列的种种特征（比如均匀分布、互相独立等等）．这样的数列即称为 **伪随机数** 序列．

随机数与伪随机数在实际生活和算法中的应用举例：

- 抽样调查时往往只需使用伪随机数．这是因为我们本就只关心统计特征．
- 网络安全中往往要用到（比刚刚提到的伪随机数）更强的随机数．这是因为攻击者可能会利用可预测性做文章．
- OI/ICPC 中用到的随机算法，基本都只需要伪随机数．这是因为，这些算法往往是通过引入随机数来把概率引入复杂度分析，从而降低复杂度．这本质上依然只利用了随机数的统计特征．
- 某些随机算法（例如 Moser 算法[1]）用到了随机数的熵相关的性质，因此必须使用真正的随机数．

随机数生成方法

`rand`

用于生成伪随机数，缺点是比较慢，使用时需要 `#include<cstdlib>`．

调用 `rand()` 函数会返回一个 `[0,RAND_MAX]` 中的随机非负整数，其中 `RAND_MAX` 是标准库中的一个宏，在 Linux 系统下 `RAND_MAX` 等于 $2^{31}-1$．可以用取模来限制所生成的数的大小．

使用 `rand()` 需要一个随机数种子，可以使用 `srand(seed)` 函数来将随机种子更改为 `seed`，当然不初始化也是可以的．

同一程序使用相同的 `seed` 两次运行，在同一机器、同一编译器下，随机出的结果将会是相同的．

有一个选择是使用当前系统时间来作为随机种子：`srand(time(nullptr))`．

> ⚠ **Warning**
>
> 在 `Windows` 系统下 `rand()` 返回值的取值范围为 $[0,2^{15})$（即 `RAND_MAX` 等于 $2^{15}-1$），当需要生成的数不小于 2^{15} 时建议使用 `(rand() << 15 | rand())` 来生成更大的随机数．

关于 `rand()` 和 `rand()%n` 的随机性：

- C/C++ 标准并未关于 `rand()` 所生成随机数的任何方面的质量做任何规定．
- GCC 对 `rand()` 所采用的实现方式，保证了分布的均匀性等基本性质，但具有低位周期长度短等明显缺陷．（例如在笔者的机器上，`rand()%2` 所生成的序列的周期长约 $2\cdot 10^6$）
- 即使假设 `rand()` 是均匀随机的，`rand()%n` 也不能保证均匀性，因为 `[0,n)` 中的每个数在 `0%n,1%n,...,RAND_MAX%n` 中的出现次数可能不相同．严格保证均匀性的做法可参考 Daniel Lemire, Fast Random Integer Generation in an Interval[2]．

预定义随机数生成器

定义了数个特别的流行算法. 如没有特别说明, 均定义于头文件 `<random>`.

> **Warning**
>
> 预定义随机数生成器于 C++11 标准[1]开始使用.

梅森缠绕器

梅森缠绕器（Mersenne Twister）由松本与西村于 1998 年提出[2], 因其中一种实现 MT19937 具有梅森素数 $M_{19937} = 2^{19937} - 1$ 这么长的周期而得名.

从 C++11 开始, 模板类 `std::mersenne_twister_engine` 实现基于如上方法的随机数生成器. 因其使用起来十分复杂, 通常实际使用的是该模板类的特化: `std::mt19937` 和 `std::mt19937_64`. 其优点是随机数质量高, 且速度比 `rand()` 快很多.

`mt19937` 基于 32 位梅森缠绕器, 由松本与西村于 1998 年提出[3], 是一个随机数生成器类, 效用同 `rand()`, 随机数的范围同 `unsigned int` 类型的取值范围. 使用时用其定义一个随机数生成器即可: `mt19937 myrand(seed)`, `seed` 可不填, 不填 `seed` 则会使用默认随机种子. 其重载了 `operator()`, 需要生成随机数时调用 `myrand()` 即可返回一个随机数.

另一个类似的生成器是 `mt19937_64`, 基于 64 位梅森缠绕器, 由松本与西村于 2000 年提出, 使用方式同 `mt19937`, 但随机数范围扩大到了 `unsigned long long` 类型的取值范围.

> **代码示例**

```cpp

#include <ctime>
#include <iostream>
#include <random>

using namespace std;

int main() {
  mt19937 myrand(SEED);  // 将 SEED 替换为任意整数或换为 time(nullptr)
  cout << myrand() << endl;
  return 0;
}

```

线性同余随机数生成器

线性同余随机数生成器（简称 LCG）由 Thomson 和 Rotenberg 于 1958 年提出. 其计算公式如下, 其中 A, C, M 为预定义常数:

$$s_i \equiv \begin{cases} \text{seed} & i = 0 \\ s_{i-1} \times A + C & i \geq 1 \end{cases} \pmod{M}$$

从 C++11 开始, 模板类 `std::linear_congruential_engine` 实现基于如上方法的随机数生成器, 模板参数如下:

```cpp
template <class UIntType, UIntType A, UIntType C, UIntType M>
class linear_congruential_engine;
```

[1]伪随机数生成 - cppreference.com[3]
[2]Mersenne Twister algorithm[4]
[3]Mersenne Twister algorithm[5]

这里 `UIntType` 表示 s 的类型，随后是三个常数.

随后 Lewis、Goodman 及 Miller 于 1969 提出，取 $A = 7^5 = 16807$、$C = 0$、$M = 2^{31} - 1 = 2147483647$ 时，此算法可以达到该模数下近乎最大的周期 $2^{31} - 2$（因为 16807 是该模数下的 原根→9.11.18@p1275）且具有相对不错的统计特征.

之后于 1988 年，该参数的 LCG 由 Park 与 Miller 采纳为「最小标准」，只因其实现极其简单、易懂、高效、且质量相对不错. 最后于 1993 年，Park、Miller 和 Stockmeyer 将参数 A 改为 48271，成为较新的「最小标准」. 两个版本的「最小标准」作为 `linear_congruential_engine` 的特化，也都于 C++11 中预定义，分别为 `minstd_rand0` 和 `minstd_rand`. 具体而言：

- 对于 `minstd_rand0`，s_i 的类型为 `std::uint_fast32_t`，A 取 16807，C 取 0，M 取 2147483647.
- 对于 `minstd_rand`，s_i 的类型为 `std::uint_fast32_t`，A 取 48271，C 取 0，M 取 2147483647.

使用方法也很简单. 首先定义 `std::minstd_rand myrand(seed);`，其中 `seed` 为种子，不填时默认为 1. 然后调用 `myrand()` 即可获得一个随机数.

如果需要自定义该形式的随机数生成器，且 A, C, M 不是常数，则还可以考虑手写实现. 该方法实现难度低，但生成的随机序列周期长度较短（周期最大为 M，但大多数情况下都会比 M 短）.

参考实现

```cpp
#include <iostream>
using namespace std;

struct myrand {
  int A, C, M, x;

  myrand(int A, int C, int M) {
    this->A = A;
    this->C = C;
    this->M = M;
    this->x = 0;
  }

  // 生成随机序列的下一个随机数
  int next() { return x = ((long long)A * x + C) % M; }
};

myrand rnd(3, 5, 97);  // 初始化一个随机数生成器

int main() {
  int x = rnd.next();
  cout << x << endl;
  return 0;
}
```

随机数引擎适配器

在预定义随机数生成器的基础上，可以对其进行二次封装得到新的随机数生成器. 具体类名和使用方法请参见 伪随机数生成——随机数引擎适配器[6] 的列表.

下面的代码使用 `std::independent_bits_engine` 包装 `std::minstd_rand` 的输出.

```cpp
#include <cstdint>
#include <iostream>
```

```cpp
#include <random>

using std::cout;

int main() {
  std::independent_bits_engine<std::minstd_rand, 32, std::uint_fast32_t> rng;
  for (int i = 0; i < 10; ++i) {
    cout << rng() << " ";
  }
  return 0;
}
```

非确定随机数的均匀分布整数随机数生成器

`random_device` 是一个基于硬件的均匀分布随机数生成器，**在熵池耗尽** 前可以高速生成随机数．该类在 C++11 定义，需要 `random` 头文件．由于熵池耗尽后性能急剧下降，所以建议用此方法生成 `mt19937` 等伪随机数的种子，而不是直接生成．

`random_device` 是非确定的均匀随机位生成器，尽管若不支持非确定随机数生成，则允许实现用伪随机数引擎实现．目前笔者尚未接到报告称 NOIP 评测机不支持基于硬件的均匀分布随机数生成．但出于保守考虑，建议使用该算法生成随机数种子．

参考代码如下．

```cpp
#include <iostream>
#include <map>
#include <random>
#include <string>

int main() {
  std::random_device rd;
  std::map<int, int> hist;
  std::uniform_int_distribution<int> dist(0, 9);
  for (int n = 0; n < 20000; ++n) {
    ++hist[dist(rd)];
  }
  for (auto p : hist) {
    std::cout << p.first << " : " << std::string(p.second / 100, '*') << '\n';
  }
  return 0;
}
```

可能的输出如下．

```
0 : ********************
1 : ******************
2 : ********************
3 : ********************
4 : ********************
5 : ******************
6 : ********************
7 : ********************
8 : ******************
9 : ********************
```

随机数分布

这里介绍的是要求生成的随机数遵从某一分布的随机数生成器, 如 离散均匀分布[7], 伯努利分布[8], 二项分布[9]，几何分布[10]，标准正态（高斯）分布[11].

具体类名请参见 伪随机数生成——随机数分布[12] 的列表.

下面的程序模拟了一个六面体骰子.

```cpp
#include <iostream>
#include <random>
using std::cout;
using std::mt19937;

using std::uniform_int_distribution;
int main() {
  mt19937 gen(SEED);  // 播种标准 mersenne_twister_engine
  uniform_int_distribution<> dis(1, 6);

  for (int n = 0; n < 10; ++n)
    // 用 dis 变换 gen 所生成的随机 unsigned int 到 [1, 6] 中的 int
    cout << dis(gen) << ' ';
  cout << '\n';
  return 0;
}

```

其他实现方法

有的时候我们需要实现自己的随机数生成器. 下面是一些常用的随机数生成方法.

Xorshift

Xorshift 系列随机数生成器由 George Marsaglia 于 2003 年提出，其主要基于异或一个数的移位这一种操作. 在算法竞赛中常用的版本有 `xorshift32` 和 `xorshift64` 两种:

```cpp

#include <cstdint>

using u32 = std::uint32_t;
using u64 = std::uint64_t;

class xorshift32 {
 private:
  u32 x;

 public:
  xorshift32() : x(2463534242) {}

  explicit xorshift32(u32 s) : x(s) {}

  u32 operator()() {
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return x;
  }
};
```

```
23
24 class xorshift64 {
25  private:
26   u64 x;
27
28  public:
29   xorshift64() : x(88172645463325252) {}
30
31   explicit xorshift64(u64 s) : x(s) {}
32
33   u64 operator()() {
34     x ^= x << 13;
35     x ^= x >> 7;
36     x ^= x << 17;
37     return x;
38   }
39 };
40
```

使用方法与前面介绍的大多数随机数生成器类似：首先定义随机数生成器 `xorshift32 rng32(seed)` 或 `xorshift64 rng64(seed)`，其中 `seed` 为种子，不填时使用默认种子；之后调用 `rng32()` 或 `rng64()` 即可获得随机数. 这两个随机数生成器在种子不为 0 时分别具有 $2^{32}-1$ 和 $2^{64}-1$ 的周期，因其实现极其简单，常将其和种子下发给选手生成大范围的数据以减少 I/O 开销.

SplitMix

SplitMix 系列随机数生成器由 Stelle，Lee 和 Flood 于 2014 年提出，并应用于 Java 8 的 `java.util.SplittableRandom` 类中. 在算法竞赛界最常用的版本为 `splitmix64`：

```
1
2 #include <cstdint>
3
4 using u64 = std::uint64_t;
5
6 class splitmix64 {
7  private:
8   u64 x;
9
10  public:
11   splitmix64() : x(0) {}
12
13   explicit splitmix64(u64 s) : x(s) {}
14
15   u64 operator()() {
16     u64 z = (x += 0x9e3779b97f4a7c15);
17     z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
18     z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
19     return z ^ (z >> 31);
20   }
21 };
22
```

其最主要的用途是对一个已经求出的哈希值进行二次哈希，以防止被人通过精心构造的数据卡哈希表卡到超时：

```
1 #include <cstdint>
```

```

using u64 = std::uint64_t;

struct MyHash {
  u64 mix64(u64 x) const {
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(size_t x) const {
    // 推荐使用 <chrono> 库的
    // std::chrono::steady_clock::now().time_since_epoch().count() 作为
    // FIXED_RANDOM
    static const u64 FIXED_RANDOM = 0x9e3779b97f4a7c15;
    return mix64(x + FIXED_RANDOM);
  }
};

```

其亦可用于生成随机数：首先定义 `splitmix64 rng64(seed)`，其中 `seed` 为种子，不填时使用默认种子；之后调用 `rng64()` 即可获得随机数．此时较为显然的，该函数具有 2^{64} 的周期（因为所有的大数都是奇数，所以 x 具有 2^{64} 的周期，且后面对 z 的所有操作都是完全可逆的）．这也意味着即使你记不住如上的大参数，你也可以直接随便写几个足够大的奇数结合异或右移来达到相近的效果．

时滞斐波那契随机数生成器

利用下式来生成随机数序列 $\{R_i\}$（其中 $0 < j < k$）：

$$R_i \equiv R_{i-j} \star R_{i-k} \bmod P$$

这里的 P 通常取 2 的幂（常用 2^{32} 或 2^{64}），$\star$ 表示二元运算符，可以使用加法，减法，乘法，异或．

该方法较传统的线性同余随机数生成器而言，拥有更长的周期，但随机性受初始条件影响较大．

参考实现

```

#include <iostream>
#include <random>
#include <vector>

using namespace std;

struct myrand {
  vector<unsigned int> vec;
  int l, j, k, cur;

  template <class URBG>
  myrand(int l, int j, int k, URBG &&rng) {
    this->l = l;
    this->j = j;
    this->k = k;
    cur = 0;
    for (int i = 0; i < l; i++) {
      vec.push_back(rng());  // 先用其他方法生成随机序列中的前几个元素
```

```
    }
  }

  unsigned int next() {
    vec[cur] = vec[(cur - j + l) % l] * vec[(cur - k + l) % l];
    // 这里用 unsigned 类型是为了实现自动对 2^32 取模
    return vec[cur++];
  }
};

// 最后一个可以换成其它随机数生成器，自行设置种子
myrand rnd(11, 4, 7, mt19937(SEED));

int main() {
  unsigned int x = rnd.next();
  cout << x << endl;
  return 0;
}

```

值得一提的是，在 C++11 中，`std::subtract_with_carry_engine` 是这一类随机数生成器的其中一种实现，而 `std::ranlux24_base` 和 `std::ranlux48_base` 是该模板类的特化，`std::ranlux24` 和 `std::ranlux48` 则是在前者的基础上再套上一个适配器 `std::discard_block_engine` 得到. 不过现在通常不再推荐使用它们.

随机算法

下面介绍在 C++ 标准中定义的一些依赖随机数生成的随机算法.

`random_shuffle`

`std::random_shuffle` 于 C++98 引入, 用于随机打乱指定序列. 使用时需要 `#include<algorithm>`, 常用实现方法为 Fisher-Yates 算法[13].

使用时传入指定区间的首尾指针或迭代器（左闭右开）即可: `std::random_shuffle(first, last)` 或 `std::random_shuffle(first, last, myrand)`.

内部使用的随机数生成器默认为 `rand()`. 当然也可以传入自定义的随机数生成器.

关于 `random_shuffle` 的随机性:

- C++ 标准中要求 `random_shuffle` 在所有可能的排列中 **等概率** 随机选取，但 GCC[4]的默认标准库 libstdc++**并未** 严格执行.
- GCC 中 `random_shuffle` 随机性上的缺陷的原因之一，是它使用了 `rand()%n` 这样的写法. 如先前所述，这样生成的不是均匀随机的整数.
- 原因之二，是 `rand()` 的值域有限. 如果所传入的区间长度超过 `RAND_MAX`，将存在某些排列 **不可能** 被产生 [5].

> **⚠ Warning**
>
> `random_shuffle` 已于 C++14 标准中被弃用，于 C++17 标准中被移除.

[4] 版本号为 GCC 9.2.0

[5] Don't use rand(): a guide to random number generators in C++[14]

shuffle

`std::shuffle` 于 C++11 引入，效用同 `random_shuffle`. 使用时需要 `#include<algorithm>`.

区别在于必须使用自定义的随机数生成器：`std::shuffle(first, last, myrand)`.

GCC[6] 实现的 `shuffle` 符合 C++ 标准的要求，即在所有可能的排列中等概率随机选取.

下面是用 `rand()` 及 `random_shuffle()` 编写的一个数据生成器. 生成数据为「ZJOI2012」灾难[15] 的随机小数据.

```cpp
#include <algorithm>
#include <cstdlib>
#include <iostream>
using std::cout;

using std::random_shuffle;

int a[100];

int main() {
  srand(SEED);
  int n = rand() % 99 + 1;
  for (int i = 1; i <= n; i++) a[i] = i;
  cout << n << '\n';
  for (int i = 1; i <= n; i++) {
    random_shuffle(a + 1, a + i);
    int cnt = rand() % i;
    for (int j = 1; j <= cnt; j++) cout << a[j] << ' ';
    cout << 0 << '\n';
  }
}

```

下面是用 `mt19937` 及 `shuffle()` 编写的同一个数据生成器.

```cpp
#include <algorithm>
#include <iostream>
#include <random>
using std::cout;
using std::mt19937;

using std::shuffle;

int a[100];

int main() {
  mt19937 rng(SEED);
  int n = rng() % 99 + 1;
  for (int i = 1; i <= n; i++) a[i] = i;
  cout << n << '\n';
  for (int i = 1; i <= n; i++) {
    shuffle(a + 1, a + i, rng);
    int cnt = rng() % i;
    for (int j = 1; j <= cnt; j++) cout << a[j] << ' ';
    cout << 0 << '\n';
```

[6] 版本号为 GCC 9.2.0

```
  }
}

```

下面是随机排列前十个正整数的一个实现.

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>
#include <vector>
using std::copy;
using std::cout;
using std::mt19937;
using std::ostream_iterator;
using std::vector;

using std::shuffle;

int main() {
  vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  mt19937 g(SEED);

  shuffle(v.begin(), v.end(), g);

  copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
  cout << "\n";
}

```

sample

`std::sample` 于 C++17 引入，用于从序列里随机选择其中 n 个元素，与 `shuffle` 后提取前 n 个元素的主要区别在于选出来的元素输出后仍保持相对顺序. 常用实现方法为 蓄水池抽样法[16].

使用方法为 `std::sample(first, last, dest, n, myrand)`，其中 `[first, last)` 表示要采样的范围，`[dest, dest + n)` 表示要输出到的地方，`myrand` 是指定的随机数生成器.

下面是从 11 个大写字母里随机抽取 4 个的实现.

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>
#include <string>
using std::back_inserter;
using std::copy;
using std::cout;
using std::mt19937;
using std::string;

using std::sample;

int main() {
  string in{"ABCDEFGHIJK"}, out;
```

```
  mt19937 rng{SEED};
  sample(in.begin(), in.end(), back_inserter(out), 4, rng);
  cout << "Four random letters out of " << in << ": " << out << '\n';
}

```

参考资料与注释

[1] Moser 算法

[2] Daniel Lemire, Fast Random Integer Generation in an Interval

[3] 伪随机数生成 - cppreference.com

[4] Mersenne Twister algorithm

[5] Mersenne Twister algorithm

[6] 伪随机数生成——随机数引擎适配器

[7] 离散均匀分布

[8] 伯努利分布

[9] 二项分布

[10] 几何分布

[11] 标准正态（高斯）分布

[12] 伪随机数生成——随机数分布

[13] Fisher-Yates 算法

[14] Don't use rand(): a guide to random number generators in C++

[15] 「ZJOI2012」灾难

[16] 蓄水池抽样法

13.6.2 随机化技巧

Authors: lr1d, partychicken, ouuan, Marcythm, TianyiQ

概述

前置知识：随机函数→13.6.1@p3252 和 概率初步→9.17.1@p1889

本文将对 OI/ICPC 中的随机化相关技巧做一个简单的分类，并对每个分类予以介绍. 本文也将介绍一些在 OI/ICPC 中很少使用，但与 OI/ICPC 在风格等方面较为贴近的方法，这些内容前将用 `(*)` 标注.

这一分类并不代表广泛共识，也必定不能囊括所有可能性，因此仅供参考.

记号和约定:

- $\Pr[A]$ 表示事件 A 发生的概率.
- $\mathrm{E}[X]$ 表示随机变量 X 的期望.
- 赋值号 $:=$ 表示引入新的量，例如 $Y := 1926$ 表示引入值为 1926 的量 Y.

用随机集合覆盖目标元素

庞大的解空间中有一个（或多个）解是我们想要的. 我们可以尝试进行多次撒网，只要有一次能够网住目标解就能成功.

例：三部图的判定

问题

给定一张 n 个结点、m 条边的简单无向图，用 RGB 三种颜色给每个结点染色 满足任意一对邻居都不同色，或者报告无解.

对每个点 v，从 $\{R, G, B\}$ 中等概率独立随机地选一种颜色 C_v，并钦定 v **不** 被染成 C_v. 最优解恰好符合这些限制的概率，显然是 $\left(\frac{2}{3}\right)^n$.

在这些限制下，对于一对邻居 (u,v)，「u,v 不同色」的要求等价于以下这条「推出」关系：

- 对于所有异于 C_u, C_v 的颜色 X，若 u 被染成 X，则 v 被染成 $\{R,G,B\}\setminus\{X,C_v\}$.

于是我们可以对每个 v 设置布尔变量 B_v，其取值表示 v 被染成两种剩余的颜色中的哪一种．借助 2-SAT 模型即可以 $O(n+m)$ 的复杂度解决这个问题．

这样做，单次的正确率是 $\left(\frac{2}{3}\right)^n$．将算法重复运行 $-\left(\frac{3}{2}\right)^n \log\epsilon$ 次，只要有一次得到解就输出，这样即可保证 $1-\epsilon$ 的正确率．（详见后文中「概率上界的分析」）

* * *

回顾：本题中「解空间」就是集合 $\{R,G,B\}^n$，我们每次通过随机施加限制来在一个缩小的范围内搜寻「目标解」——即合法的染色方案．

例：CodeChef SELEDGE[1]

简要题意

给定一张点、边都有非负权值的无向图，找到一个大小 $\le K$ 的边集合 S，以最大化与 S 相连的点的权值和减去 S 的边权和．一个点的权值只被计算一次．

观察：如果选出的边中有三条边构成一条链，则删掉中间的那条一定不劣；如果选出的边中有若干条构成环，则删掉任何一条一定不劣．

推论：最优解选出的边集，一定构成若干个不相交的菊花图（即直径不超过 2 的树）．

推论：最优解选出的边集，一定构成一张二分图．

我们对每个点等概率独立随机地染上黑白两种颜色之一，并要求这一染色方案，恰好也是最优解所对应的二分图的黑白染色方案．

尝试计算最优解符合这一要求的概率：

- 考虑一张 n 个点的菊花图，显然它有 2 种染色方案，所以它被染对颜色的概率是 $\frac{2}{2^n}=2^{1-n}$.
- 假设最优解中每个菊花的结点数分别为 $a_1,\cdots,a_l$，则一定有 $(a_1-1)+\cdots+(a_l-1)\le K$，其中 K 表示最多能够选出的边数.
- 从而所有菊花都被染对颜色的概率是 $2^{1-a_1}\cdots 2^{1-a_l}\ge 2^{-K}$.

在上述要求下，尝试建立费用流模型计算最优答案：

- 建立二分图，白点在左侧并与 S 相连，黑点在右侧并与 T 相连.
  - 对于白点 v，从 S 向它连一条容量为 1、费用为 $-A_v$ 的边，和一条容量为 ∞、费用为 0 的边.
  - 对于黑点 v，从它向 T 连一条容量为 1、费用为 $-A_v$ 的边，和一条容量为 ∞、费用为 0 的边.
- 对于原图中的边 (u,v,B) 满足 u 为白色、v 为黑色，连一条从 u 到 v 的边，容量为 1，费用为 B.
- 在该图中限制流量不超过 K，则最小费用的相反数就是答案.

用 SPFA 费用流求解的话，复杂度是 $O\big(K^2(n+m)\big)$，证明：

- 首先，显然 SPFA 的运行次数 $\le K$.
- 然后，在一次 SPFA 中，任何一个结点至多入队 $O(K)$ 次．这是因为：
  - 任意时刻有流量的边不会超过 $3K$ 条，否则就意味着在原图中选了超过 K 条边.
  - 对于任何一条长为 L 的增广路，其中至少有 $\frac{L}{2}-2$ 条边是某条有流量的边的反向边，因为正向边都是从图的左侧指向右侧，只有这些反向边才会从右侧指向左侧.
  - 综合以上两条，得到任意一条增广路的长度不超过 $6K+4$.
- 综上，复杂度是 $O\big(K^2(n+m)\big)$.

和上一题类似，我们需要把整个过程重复 $-2^K\log\epsilon$ 次以得到 $1-\epsilon$ 的正确率．总复杂度 $O\big(2^K K^2(n+m)\cdot-\log\epsilon\big)$.

用随机元素命中目标集合

我们需要确定一个集合中的任意一个元素，为此我们随机选取元素，以期能够恰好命中这一集合.

例：Gym 101550I[2]

简要题意

有一张图形如：两条平行的链，加上连接两链的两条平行边. 给定这张图上的若干条简单路径（每条路径表示一次通话），请你选择尽量少的边放置窃听器，以使得每条给定的路径上都有至少一个窃听器.

整张图可以拆分为一个环加上四条从环伸出去的链. 对于这四条链中的任何一条（记作 C），考虑在这条链上如何放置窃听器，容易通过贪心算法得到满足以下条件的方案：

- 在拦截所有 C 内部进行的通话的前提下，用的窃听器数量最少.
- 在上一条的前提下，使得 C 上的窃听器离环的最短距离尽可能小.
  - 作这一要求的目的是尽可能地拦截恰有一个端点在 C 内部的通话.

接着考虑链与环相接处的共计 4 条边，我们暴力枚举这些边上有没有放窃听器. 显然，如果想要拦截跨越链和环的通话，在这 4 条边上放窃听器一定是最优的. 现在，我们可以把通话线路分为以下几种：

1. 完全在链上的通话线路. 这些线路一定已经被拦截，故可以忽略.
2. 跨越链和环，且已经被拦截的通话线路. 它们可以忽略.
3. 跨越链和环，且未被拦截的通话线路. 我们可以直接截掉它在链上的部分（因为链上的窃听器放置方案已经固定了），只保留环上的部分.
4. 完全在环上的通话线路.

至此，问题转化成了环上的问题.

设最优解中在环上的边集 S 上放置了窃听器，如果我们已经确定了 S 中的任何一个元素 e，就可以：

- 先在 e 处断环为链.
- 然后从 e 开始贪心，不断找到下一个放置窃听器的边. 注意到如果经过合适的预处理，贪心的每一步可以做到 $O(1)$ 的复杂度.
- 从而以 $O(|S|)$ 的复杂度解决问题.

我们考虑随机选取环上的一条边 e'，并钦定 $e' \in S$ 再执行上述过程，重复多次取最优.

分析单次复杂度：

- 观察：记 S' 表示所有选取了 e' 的方案中的最优解，则 $|S'| \leq |S| + 1$.
- 从而单次复杂度 $O(|S'|) = O(|S|)$.

分析正确率：

- 显然单次正确率 $\dfrac{|S|}{n}$，其中 n 表示环长.
- 所以需要重复 $-\dfrac{n}{|S|}\log\epsilon$ 次以得到 $1-\epsilon$ 的正确率.

综上，该算法的复杂度 $O\left(|S| \cdot -\dfrac{n}{|S|}\log\epsilon\right) = O(-n\log\epsilon)$.

例：CSES 1685 New Flight Routes[3]

简要题意

给定一张有向图，请你加最少的边使得该图强连通，需 **输出方案**.

先对原图进行强连通缩点. 我们的目标显然是使每个汇点能到达每个源点.

不难证明，我们一定只会从汇点到源点连边，因为任何其他的连边，都能对应上一条不弱于它的、从汇点到源点的连边.

我们的一个核心操作是，取汇点 t 和源点 s（它们不必在同一个弱连通分量里），连边 $t \to s$ 以 **使得 s 和 t 都不再是汇点或源点**（记作目标 I）．理想情况下这种操作每次能减少一个汇点和一个源点，那我们不断操作直到只剩一个汇点或只剩一个源点，而这样的情形就很平凡了．由此，我们猜测答案是源点个数与汇点个数的较大值．

不难发现，上述操作能够达到目标 I 的充要条件是：t 拥有 s 以外的前驱、且 s 拥有 t 以外的后继．可以证明（等会会给出证明），对于任意一张有着至少两个源点和至少两个汇点的 DAG，都存在这样的 (s,t)；但存在性的结论无法帮助我们构造方案，还需做其他分析．

- 有了这个充要条件还难以直接得到算法，主要的原因是连边 $t \to s$ 后可能影响其他 (s',t') 二元组的合法性，这个比较难处理．

注意到我们关于源汇点间的关系知之甚少（甚至连快速查询一对 $s-t$ 间是否可达都需要 dfs + bitset 预处理，而时限并不允许这么做），这提示我们需要某种非常一般和强大的性质．

观察：不满足目标 I 的 (s,t) 至多有 $n+m-1$ 对，其中 n 表示源点个数，m 表示汇点个数．

- 理由：对于每一对这样的 (s,t)，若把它看成 s,t 间的一条边，则所有这些边构成的图形如若干条不相交的链，于是边数不超过点数减一．
- 作出这一观察的动机是，要想将存在性结论应用于算法，前置步骤往往是把定性的结果加强为定量的结果．

推论：等概率随机选取 (s,t)，满足前述要求的概率 $\geq \dfrac{(n-1)(m-1)}{nm}$．

- 注意到这个结论严格强于先前给出的存在性结论．

推论：等概率独立随机地连续选取 $\dfrac{\min(n,m)}{2}$ 对不含公共元素的 (s,t)，并对它们 **依次** 操作（即连边 $t \to s$），则这些操作全部满足目标 I 的概率 $\geq \dfrac{1}{4}$．

- 理由：

$$\begin{aligned}
&\frac{(n-1)(m-1)}{nm} \cdot \frac{(n-2)(m-2)}{(n-1)(m-1)} \cdots \frac{(n-k)(m-k)}{(n-k+1)(m-k+1)} \\
=\ &\frac{(n-k)(m-k)}{nm} \\
\geq\ &\frac{1}{4}
\end{aligned}$$

而连续选完 k 对 (s,t) 后判断它们是否全部满足目标 I 很简单，只要再跑一遍强连通缩点，判断一下 n,m 是否都减小了 k 即可．注意到若每次减少 $k=\dfrac{\min(n,m)}{2}$，则 $\min(n,m)$ 必在 $O\big(\log(n+m)\big)$ 轮内变成 1，也就转化到了平凡的情况．

算法伪代码

```
while(n>1 and m>1):
    randomly choose k=min(n,m)/2 pairs (s,t)
    add edge t->s for all these pairs
    if new_n>n-k or new_m>m-k:
        roll_back()
solve_trivial()
```

复杂度 $O\big((\mid V \mid + \mid E \mid)\log \mid V \mid\big)$．

* * *

回顾：我们需要确定任意一对能够实现目标 I 的二元组 (s,t)，为此我们随机选择 (s,t)．

用随机化获得随机数据的性质

如果一道题的数据随机生成，我们可能可以利用随机数据的性质解决它．而在有些情况下，即使数据并非随机生成，我们也可以通过随机化来给其赋予随机数据的某些特性，从而帮助解决问题．

例：随机增量法

随机生成的元素序列可能具有「前缀最优解变化次数期望下很小」等性质，而随机增量法就通过随机打乱输入的序列来获得这些性质.

详见 随机增量法→12.12@p3152.

例：TopCoder MagicMolecule[4] 随机化解法

简要题意

给定一张 n 个点、带点权的无向图，在其中所有大小不小于 $\frac{2n}{3}$ 的团中，找到点权和最大的那个.

$n \leq 50$

不难想到折半搜索. 把点集均匀分成左右两半 V_L, V_R（大小都为 $\frac{n}{2}$），计算数组 $f_{L,k}$ 表示点集 $L \subseteq V_L$ 中的所有 $\geq k$ 元团的最大权值和. 接着我们枚举右半边的每个团 C_R，算出左半边有哪些点与 C_R 中的所有点相连（这个点集记作 N_L），并用 $f_{N_L,\frac{2}{3}n-|C_R|} + value(C_R)$ 更新答案.

- 注意到可以 $O(1)$ 转移每一个 $f_{L,k}$. 具体地说，取 d 为 L 中的任意一个元素，然后分类讨论：
  - 假设最优解中 d 不在团中，则从 $f_{L\setminus\{d\},k}$ 转移而来.
  - 假设最优解中 d 在团中，则从 $f_{L\cap N(d),k} + value(d)$ 转移而来，其中 $N(d)$ 表示 d 的邻居集合.
  - 别忘了还要用 $f_{L,k+1}$ 来更新 $f_{L,k}$.

这个解法会超时. 尝试优化：

- 平分点集时均匀随机地划分. 这样的话，最优解的点集 C_{res} 以可观的概率也被恰好平分（即 $|C_{res} \cap V_L| = |C_{res} \cap V_R|$）.
  - 当然，$|C_{res}|$ 可能是奇数. 简单起见，这里假设它是偶数；奇数的情况对解法没有本质改变.
  - 实验发现，随机尝试约 20 次就能以很大概率有至少一次满足该性质. 也就是说，如果我们的算法依赖于「C_{res} 被平分」这一性质，则将算法重复执行 20 次取最优，同样也能保证以很大概率得到正确答案.
- 有了这一性质，我们就可以直接钦定左侧团 L、右侧团 C_R 的大小都 $\geq \frac{n}{3}$. 这会对复杂度带来两处改进：
  - f 可以省掉记录大小的维度.
  - 因为只需考虑大小 $\geq \frac{n}{3}$ 的团，所以需要考虑的左侧团 L 和 右侧团 C_R 的数量也大大减少至约 $1.8 \cdot 10^6$.
- 现在的瓶颈变成了求单侧的某一子集的权值和，因为这需要 $O\left(2^{|V_L|} + 2^{|V_R|}\right)$ 的预处理.
  - 解决方案：在 V_L, V_R 内部再次折半；当查询一个子集的权值和时，将这个子集分成左右两半查询，再把答案相加.
- 这样即可通过本题.

* * *

回顾：一个随机的集合有着「在划分出的两半的数量差距不会太悬殊」这一性质，而我们通过随机划分获取了这个性质.

随机化用于哈希

例：UOJ #207 共价大爷游长沙[5]

简要题意

维护一棵动态变化的树，和一个动态变化的结点二元组集合. 你需要支持：

- 删边、加边. 保证得到的还是一棵树.
- 加入/删除某个结点二元组.

- 给定一条边 e，判断是否对于集合中的每个结点二元组 (s,t)，e 都在 s,t 间的简单路径上.

对图中的每条边 e，我们定义集合 S_e 表示经过该边的关键路径（即题中的 (a,b)）集合. 考虑对每条边动态维护集合 S_e 的哈希值，这样就能轻松判定 S_e 是否等于全集（即 e 是否是「必经之路」）.

哈希的方式是，对每个 (a,b) 赋予 2^{64} 以内的随机非负整数 $H_{(a,b)}$，然后一个集合的哈希值就是其中元素的 H 值的异或和.

这样的话，任何一个固定的集合的哈希值一定服从 $R:=\{0,1,\cdots,2^{64}-1\}$ 上的均匀分布（换句话说，哈希值的取值范围为 R，且取每一个值的概率相等）. 这是因为:

1. 单个 $H_{(a,b)}$ 显然服从均匀分布.
2. 两个独立且服从 R 上的均匀分布的随机变量的异或和，一定也服从 R 上的均匀分布. 自证不难.

从而该算法的正确率是有保障的.

至于如何维护这个哈希值，使用 LCT 即可.

例：CodeChef PANIC[6] 及其错误率分析

本题的大致解法:

1. 可以证明[7] $S(N)$ 服从一个关于 N 的 $O(K)$ 阶线性递推式.
2. 用 BM 算法求出该递推式.
3. 借助递推式，用凯莱哈密顿定理计算出 $S(N)$.

这里仅关注第二部分，即如何求一个矩阵序列的递推式. 所以我们只需考虑下述问题:

问题

给定一个矩阵序列，该序列在模 $P:=998244353$ 意义下服从一个齐次线性递推式（递推式中的数乘和加法运算定义为矩阵的数乘和加法），求出最短递推式.

如果一系列矩阵服从一个递推式 F，那么它的每一位也一定服从 F. 然而，如果对某一位求出最短递推式 F'，则 F' 可能会比 F 更短，从而产生问题.

解决方案: 给矩阵的每一位 (i,j) 赋予一个 $<P$ 的随机权值 $x_{i,j}$，然后对于序列中每个矩阵计算其所有位的加权和模 P 的结果，再把每个矩阵算出的这个数连成一个数列，最后我们对所得数列运行 BM 算法.

错误率分析:

- 假设上述做法求得了不同于 F（且显然也不长于 F）的 l 阶递推式 F'.
- 因为矩阵序列不服从 F'，所以一定存在矩阵中的某个位置 (i,j)，满足该位置对应的数列 $S_{i,j}$ 在某个 N 处不服从 F'. 也就是说:

$$S(N)_{i,j}-{F'}_1S(N-1)_{i,j}-\cdots-{F'}_lS(N-l)_{i,j}\not\equiv 0 \pmod P$$

- 假设 (i,j) 是唯一的不服从的位置，则一定有:

$$T_{i,j}:=\left(x_{i,j}\cdot\left(S(N)_{i,j}-{F'}_1S(N-1)_{i,j}-\cdots-{F'}_lS(N-l)_{i,j}\right)\bmod P\right)=0$$

- 显然这仅当 $x_{i,j}=0$ 时才成立，概率 P^{-1}.
- 如果有多个不服从的位置呢?
  - 对每个这样的位置 (i,j)，易证 $T_{i,j}$ 服从 $R:=\{0,1,\cdots,P-1\}$ 上的均匀分布.
  - 若干个互相独立的、服从 R 上的均匀分布的随机变量，它们在模意义下的和，依然服从 R 上的均匀分布. 自证不难.
  - 从而这种情况下的错误率也是 P^{-1}.

[7] PANIC - Editorial[7]

例：UOJ #552 同构判定鸭[8] 及其错误率分析

简要题意

给定两张边权为小写字母的有向图 G_0, G_1，你要对这两张图分别算出「所有路径对应的字符串构成的多重集」（可能是无穷集），并判断这两个多重集是否相等. 如果不相等，你要给出一个最短的串，满足它在两个多重集中的出现次数不相等.

令 $f_{K,i,j}$ 表示图 G_K 中从点 i 开始的所有长为 j 的路径，这些路径对应的所有字符串构成的多重集的哈希值. 按照 j 升序考虑每个状态，转移时枚举 i 的出边并钦定该边为路径上的第一条边.

要判断是否存在长度 $= L$ 的坏串，只需把 $\{f_{0,*,L}\}$ 和 $\{f_{1,*,L}\}$ 各自「整合」起来再比较即可（通配符 `*` 这里表示每一个结点，例如 $\{f_{0,*,L}\}$ 表示全体 $f_{0,i,L}$ 构成的集合，其中 i 取遍所有结点）. 官方题解[8] 中证明了最短坏串（如果存在的话）长度一定不超过 $n_1 + n_2$，所以这个解法的复杂度是可靠的.

接下来考虑具体的哈希方式. 注意到常规的哈希方法——即把串 $a_1a_2\cdots a_k$ 映射到 $\left(a_1 + Pa_2 + P^2a_3 + \cdots + P^{k-1}a_k\right) \bmod Q$ 上、再把多重集的哈希值定为其中元素的哈希值之和模 Q——在这里是行不通的. 一个反例是，集合 `{"ab","cd"}` 与集合 `{"cb","ad"}` 的哈希值是一样的，不论 P, Q 如何取值.

上述做法的问题在于，一个串的哈希值是一个和式，从而其中的每一项可以拆出来并重组. 为避免这一问题，我们考虑把哈希值改为一个连乘式. 此外，乘法交换律会使得不同的位不可区分，为避免这一点我们要为不同的位赋予不同的权值.

对每一个二元组 (c, j)（其中 c 为字符，j 为整数表示 c 在某个串中的第几位）我们都预先生成一个随机数 $x_{c,j}$. 然后我们把串 $a_1a_2\cdots a_k$ 映射到 $x_{a_1,1}x_{a_2,2}\cdots x_{a_k,k} \bmod Q$ 上（其中 Q 为 **随机选取** 的质数）、再把多重集的哈希值定为其中元素的哈希值之和模 Q. 接下来分析它的错误率.

(*)Schwartz–Zippel 引理

令 $f \in F[z_1, \cdots, z_k]$ 为域 F 上的 k 元 d 次非零多项式，令 S 为 F 的有限子集，则至多有 $d \cdot |S|^{k-1}$ 组 $(z_1, \cdots, z_k) \in S^k$ 满足 $f(z_1, \cdots, z_k) = 0$.

如果你不知道域是什么

你只需记得这两样东西都是域：

1. 模质数的剩余系，以及其上的各种运算.
2. 实数集，以及其上的各种运算.

推论：若 $z_1, \cdots, z_k$ 都在 S 中等概率独立随机选取，则 $\Pr[f(z_1, \cdots, z_k) = 0] \le \dfrac{d}{|S|}$.

记 F 为模 Q 的剩余系所对应的域，则对于一个 $L \le n_1 + n_2$，$\sum\limits_i f_{0,i,L}$ 和 $\sum\limits_i f_{1,i,L}$ 就分别对应着一个 F 上关于变元集合 $\{x_{*,*}\}$ 的 L 次多元多项式，不妨将这两个多项式记为 P_0, P_1.

假如两个不同的字符串多重集的哈希值相同，则有两种可能：

1. $P_0 \equiv P_1 \pmod Q$，即 P_0, P_1 的每一项系数在模 Q 意义下都对应相等.
2. $P_0 \not\equiv P_1 \pmod Q, P_0(x_{*,*}) \equiv P_1(x_{*,*}) \pmod Q$，即 P_0, P_1 虽然不恒等，但我们选取的这一组 $\{x_{*,*}\}$ 恰好使得它们在此处的点值相等.

分析前者发生的概率：

- 观察：对于任意的 $A \neq B; A, B \le N$ 和随机选取的质数 $Q \le Q_{\max}$，一定有：

$$\Pr[A \equiv B \pmod Q] = O\left(\frac{\log N \log Q_{max}}{Q_{max}}\right)$$

[8] UOJ NOI Round #4 Day2 题解[9]

- 这是因为：使 $A \equiv B$ 成立的 Q 一定满足 $Q \mid (A-B)$，这样的 Q 有 $\omega(A-B) \le \log_2 N$ 个；而由质数定理，$Q_{\max}$ 以内不同的质数又有 $\Theta\left(\frac{Q_{\max}}{\log Q_{\max}}\right)$ 个．将两者相除即可得到上式．
- 在上述观察中取 A, B（满足 $A \neq B$）为某一特定项在 P_0, P_1 中的系数（也就等于该项对应的串在 G_0, G_1 中的出现次数），则易见 $A, B \le (m_1+m_2)^L$，得到：

$$\Pr[A \equiv B \pmod Q] = O\left(\frac{L \log(m_1+m_2) \log Q_{max}}{Q_{max}}\right)$$

- 所以取 $Q_{\max} \approx 10^{12}$ 就绰绰有余．如果机器无法支持这么大的整数运算，可以用双哈希代替．

分析后者发生的概率：

- 在 Schwartz–Zippel 引理中：
  - 取域 F 为模 Q 的剩余系对应的域
  - 取 $f(x_{*,*}) = P_0(x_{*,*}) - P_1(x_{*,*})$ 为 L 次非零多项式
  - 取 $S = F$
- 得到：所求概率 $\le \frac{L}{Q}$.

注意到我们需要对每个 L 都能保证正确性，所以要想保证严谨的话还需用 Union Bound（见后文）说明一下．

实践上我们不必随机选取模数，因为——比如说——用自己的生日做模数的话，实际上已经相当于随机数了．

例：(*) 子矩阵不同元素个数

问题

给定 $n \times m$ 的矩阵，q 次询问一个连续子矩阵中不同元素的个数，要求在线算法．

允许 ϵ 的相对误差和 δ 的错误率，换句话说，你要对至少 $(1-\delta)q$ 个询问给出离正确答案相对误差不超过 ϵ 的回答．

$n \cdot m \le 2 \cdot 10^5; q \le 10^6; \epsilon = 0.5, \delta = 0.2$

引理：令 $X_{1 \cdots k}$ 为互相独立的随机变量，且取值在 $[0,1]$ 中均匀分布，则 $\mathrm{E}[\min_i X_i] = \frac{1}{k+1}$.

- 证明：考虑一个单位圆，其上分布着 **相对位置** 均匀随机的 $k+1$ 个点，分别在位置 $0, X_1, X_2, \cdots, X_k$ 处．那么 $\min_i X_i$ 就等于 $k+1$ 段空隙中特定的一段的长度．而因为这些空隙之间是「对称」的，所以其中任何一段特定空隙的期望长度都是 $\frac{1}{k+1}$.

我们取 k 为不同元素的个数，并借助上述引理来从 $\min_i X_i$ 反推得到 k.

考虑采用某个哈希函数，将矩阵中每个元素都均匀、独立地随机映射到 $[0,1]$ 中的实数上去，且相等的元素会映射到相等的实数．这样的话，一个子矩阵中的所有元素对应的那些实数，在去重后就恰好是先前的集合 $\{X_1, \cdots, X_k\}$ 的一个实例，其中 k 等于子矩阵中不同元素的个数．

于是我们得到了算法：

1. 给矩阵中元素赋 $[0,1]$ 中的哈希值．为保证随机性，哈希函数可以直接用 `map` 和随机数生成器实现，即每遇到一个新的未出现过的值就给它随机一个哈希值．
2. 回答询问时设法求出子矩阵中哈希值的最小值 M，并输出 $\frac{1}{M} - 1$.

然而，这个算法并不能令人满意．它的输出值的期望是 $\mathrm{E}\left[\frac{1}{\min_i X_i} - 1\right]$，但事实上这个值并不等于 $\frac{1}{\mathrm{E}[\min_i X_i]} - 1 = k$，而（可以证明）等于 ∞.

也就是说，我们不能直接把 $\min_i X_i$ 的单次取值放在分母上，而要先算得它的期望，再把期望值放在分母上．

怎么算期望值？多次随机取平均.

我们用 C 组不同的哈希函数分别执行前述过程，回答询问时计算出 C 个不同的 M 值，并算出其平均数 $\overline{M}$，然后输出 $\left(\overline{M}\right)^{-1}-1$.

实验发现取 $C\approx 80$ 即可满足要求．严格证明十分繁琐，在此略去.

最后，怎么求子矩阵最小值？用二维 S-T 表即可，预处理 $O(nm\log n\log m)$，回答询问 $O(1)$.

随机化在算法中的其他应用

随机化的其他作用还包括：

- 防止被造数据者用针对性数据卡掉．例如在搜索时随机打乱邻居的顺序.
- 保证算法过程中进行的「操作」具有（某种意义上的）均匀性．例如 模拟退火→13.6.4@p3278 算法.

在这些场景下，随机化常常（但并不总是）与乱搞、骗分等做法挂钩.

例：「TJOI2015」线性代数[10]

本题的标准算法是网络流，但这里我们采取这样的乱搞做法：

- 每次随机一个位置，把这个位置取反，判断大小并更新答案.

代码

```cpp
#include <algorithm>
#include <cstdlib>
#include <iostream>

int n;

int a[510], b[510], c[510][510], d[510];
int p[510], q[510];

int maxans = 0;

void check() {
  memset(d, 0, sizeof d);
  int nowans = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) d[i] += a[j] * c[i][j];
  for (int i = 1; i <= n; i++) nowans += (d[i] - b[i]) * a[i];
  maxans = std::max(maxans, nowans);
}

int main() {
  srand(19260817);
  std::cin >> n;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) std::cin >> c[i][j];
  for (int i = 1; i <= n; i++) std::cin >> b[i];
  for (int i = 1; i <= n; i++) a[i] = 1;
  check();
  for (int T = 1000; T; T--) {
    int tmp = rand() % n + 1;
    a[tmp] ^= 1;
    check();
  }
```

```
34   std::cout << maxans << '\n';
35 }
```

例：(*) 随机堆 [9]

可并堆最常用的写法应该是左偏树了，通过维护树高让树左偏来保证合并的复杂度. 然而维护树高有点麻烦，我们希望尽量避开.

那么可以考虑使用随机堆，即不按照树高来交换儿子，而是随机交换.

代码

```
1  struct Node {
2    int child[2];
3    long long val;
4  } nd[100010];
5
6  int root[100010];
7
8  int merge(int u, int v) {
9    if (!(u && v)) return u | v;
10   int x = rand() & 1, p = nd[u].val > nd[v].val ? u : v;
11   nd[p].child[x] = merge(nd[p].child[x], u + v - p);
12   return p;
13 }
14
15 void pop(int &now) { now = merge(nd[now].child[0], nd[now].child[1]); }
```

随机堆对堆的形态没有任何硬性或软性的要求，合并操作的期望复杂度对任何两个堆（作为`merge`函数的参数）都成立. 下证.

期望复杂度的证明

将证，对于任意的堆 A，从根节点开始每次随机选左或者右走下去（直到无路可走），路径长度（即路径上的结点数）的期望值 $h(A) \le \log_2(|A|+1)$.

- 注意到在前述过程中合并堆 A, B 的期望复杂度是 $O\big(h(A)+h(B)\big)$ 的，所以上述结论可以保证随机堆的期望复杂度.

证明采用数学归纳. 边界情况是 A 为空图，此时显然. 下设 A 非空.

假设 A 的两个子树分别为 L, R，则：

$$
\begin{aligned}
h(A) &= 1 + \frac{h(L)+h(R)}{2} \\
&\le 1 + \frac{\log_2(|L|+1) + \log_2(|R|+1)}{2} \\
&= \log_2 2\sqrt{(|L|+1)(|R|+1)} \\
&\le \log_2 \frac{2\big((|L|+1)+(|R|+1)\big)}{2} \\
&= \log_2(|A|+1)
\end{aligned}
$$

[9] Anna Gambin and Adam Malinowski, Randomized Meldable Priority Queues[11]

证毕.

与随机性有关的证明技巧

以下列举几个比较有用的技巧.

自然，这寥寥几项不可能就是全部；如果你了解某种没有列出的技巧，那么欢迎补充.

概率上界的分析

详见 概率不等式→9.17.5@p1898 页面.

除了上述页面中提到的各种不等式外，推导过程中还经常会用到以下结论：

自然常数的使用：$\left(1-\dfrac{1}{n}\right)^n \le \dfrac{1}{\mathrm{e}}, \forall n \ge 1$

- 左式关于 $n \ge 1$ 单调递增且在 $+\infty$ 处的极限是 $\dfrac{1}{\mathrm{e}}$，因此有这个结论.
- 这告诉我们，如果 n 个互相独立的事件，每个的发生概率为 $1-\dfrac{1}{n}$，则它们全部发生的概率至多为 $\dfrac{1}{\mathrm{e}}$.

「耦合」思想

「耦合」思想常用于同时处理超过一个有随机性的对象，或者同时处理随机的对象和确定性的对象.

引子：随机图的连通性

问题

对于 $n \in \mathbf{N}^*; p, q \in [0,1]$ 且 $q \le p$，求证：随机图 $G_1(n,p)$ 的连通分量个数的期望值不超过随机图 $G_2(n,q)$ 的连通分量个数的期望值. 这里 $G(n,\alpha)$ 表示一张 n 个结点的简单无向图 G，其中 $\dfrac{n(n-1)}{2}$ 条可能的边中的每一条都有 α 的概率出现，且这些概率互相独立.

这个结论看起来再自然不过，但严格证明却并不那么容易.

证明思路

我们假想这两张图分别使用了一个 01 随机数生成器来获知每条边存在与否，其中 G_1 的生成器 T_1 每次以 p 的概率输出 1，G_2 的生成器 T_2 每次以 q 的概率输出 1. 这样，要构造一张图，就只需把对应的生成器运行 $\dfrac{n(n-1)}{2}$ 遍即可.

现在我们把两个生成器合二为一. 考虑随机数生成器 T，每次以 q 的概率输出 0，以 $p-q$ 的概率输出 1，以 $1-p$ 的概率输出 2. 如果我们将这个 T 运行 $\dfrac{n(n-1)}{2}$ 遍，就能同时构造出 G_1 和 G_2. 具体地说，如果输出是 0，则认为 G_1 和 G_2 中都没有当前考虑的边；如果输出是 1，则认为只有 G_1 中有当前考虑的边；如果输出是 2，则认为 G_1 和 G_2 中都有当前考虑的边.

容易验证，这样生成的 G_1 和 G_2 符合其定义，而且在每个实例中，G_2 的边集都是 G_1 边集的子集. 因此在每个实例中，G_2 的连通分量个数都不小于 G_1 的连通分量个数；那么期望值自然也满足同样的大小关系.

这一段证明中用到的思想被称为「耦合」，可以从字面意思来理解这种思想. 本例中它体现为把两个本来独立的随机过程合二为一.

应用：NERC 2019 Problem G: Game Relics[12]

简要题意

有若干个物品，每个物品有一个价格 c_i. 你想要获得所有物品，为此你可以任意地进行两种操作：

1. 选择一个未拥有的物品 i，花 c_i 块钱买下来.
2. 花 x 块钱从所有物品（包括已经拥有的）中等概率随机抽取一个. 如果尚未拥有该物品，则直接获得它；否则一无所获，但是会返还 $\frac{x}{2}$ 块钱. x 为输入的常数.

问最优策略下的期望花费.

观察：如果选择抽物品，就一定会一直抽直到获得新物品为止.

- 理由：如果抽一次没有获得新物品，则新的局面和抽物品之前的局面一模一样，所以如果旧局面的最优行动是「抽一发」，则新局面的最优行动一定也是「再抽一发」.

我们可以计算出 f_k 表示：如果当前已经拥有 k 个不同物品，则期望要花多少钱才能抽到新物品. 根据刚才的观察，我们可以直接把 f_k 当作一个固定的代价，即转化为「每次花 f_k 块钱随机获得一个新物品」.

期望代价的计算

显然 $f_k = \frac{x}{2} \cdot (R-1) + x$，其中 R 表示要得到新物品期望的抽取次数.

引理：如果一枚硬币有 p 的概率掷出正面，则首次掷出正面所需的期望次数为 $\frac{1}{p}$.

- 感性理解：$\frac{1}{p} \cdot p = 1$，所以扔这么多次期望得到 1 次正面，看起来就比较对.
- 这种感性理解可以通过 大数定律[13] 严谨化，即考虑 $n \to \infty$ 次「不断抛硬币直到得到正面」的实验. 推导细节略.
- 另一种可行的证法是，直接把期望的定义带进去暴算. 推导细节略.

显然抽一次得到新物品的概率是 $\frac{n-k}{n}$，那么 $R = \frac{n}{n-k}$.

结论：最优策略一定是先抽若干次，再买掉所有没抽到的物品.

这个结论符合直觉，因为 f_k 是关于 k 递增的，早抽似乎确实比晚抽看起来好一点.

证明

先考虑证明一个特殊情况. 将证：

- 随机过程 A：先买物品 x，然后不断抽直到得到所有物品
- ……一定不优于……
- 随机过程 B：不断抽直到得到 x 以外的所有物品，然后如果还没有 x 则买下来

考虑让随机过程 A 和随机过程 B 使用同一个随机数生成器. 即，A 的第一次抽取和 B 的第一次抽取会抽到同一个元素，第二次、第三次……也是一样.

显然，此时 A 和 B 抽取的次数必定相等. 对于一个被 A 抽到的物品 $y \neq x$，观察到：

- A 中抽到 y 时已经持有的物品数，一定大于等于 B 中抽到 y 时已经持有的物品数.

因此 B 的单次抽取代价不高于 A 的单次抽取代价，进而抽取的总代价也不高于 A.

显然 B 的购买代价同样不高于 A. 综上，B 一定不劣于 A.

然后可以通过数学归纳把这一结论推广到一般情况. 具体地说，每次我们找到当前策略中的最后一次购买，然后根据上述结论，把这一次购买移到最后一定不劣. 细节略.

基于这个结论，我们再次等价地转化问题：把「选一个物品并支付对应价格购买」的操作，改成「随机选一个未拥有的物品并支付对应价格购买」. 等价性的理由是，既然购买只是用来扫尾的，那选到哪个都无所谓.

现在我们发现，「抽取」和「购买」，实质上已经变成了相同的操作，区别仅在于付出的价格不同. 选择购买还是抽取，对于获得物品的顺序毫无影响，而且每种获得物品的顺序都是等可能的.

观察：在某一时刻，我们应当选择买，当且仅当下一次抽取的代价（由已经抽到的物品数确定）大于剩余物品的平均价格（等于的话则任意）.

- 可以证明，随着时间的推移，抽取代价的增速一定不低于剩余物品均价的增速．这说明从抽到买的「临界点」只有一个，进一步验证了先前结论．

最后，我们枚举所有可能的局面（即已经拥有的元素集合），算出这种局面出现的概率（已有元素的排列方案数除以总方案数），乘上当前局面最优决策的代价（由拥有元素个数和剩余物品总价确定），再加起来即可．这个过程可以用背包式的 DP 优化，即可通过本题．

* * *

回顾：可以看到，耦合的技巧在本题中使用了两次．第一次是在证明过程中，令两个随机过程使用同一个随机源；第二次是把购买转化成随机购买（即引入随机源），从而使得购买和抽取这两种操作实质上「耦合」为同一种操作（即令抽取和购买操作共享一个随机源）．

参考资料

[1] CodeChef SELEDGE

[2] Gym 101550I

[3] CSES 1685 New Flight Routes

[4] TopCoder MagicMolecule

[5] UOJ #207 共价大爷游长沙

[6] CodeChef PANIC

[7] PANIC - Editorial

[8] UOJ #552 同构判定鸭

[9] UOJ NOI Round #4 Day2 题解

[10] 「TJOI2015」线性代数

[11] Anna Gambin and Adam Malinowski, Randomized Meldable Priority Queues

[12] NERC 2019 Problem G: Game Relics

[13] 大数定律

13.6.3 爬山算法

简介

爬山算法是一种局部择优的方法，采用启发式方法，是对深度优先搜索的一种改进，它利用反馈信息帮助生成解的决策．

直白地讲，就是当目前无法直接到达最优解，但是可以判断两个解哪个更优的时候，根据一些反馈信息生成一个新的可能解．

因此，爬山算法每次在当前找到的最优方案 x 附近寻找一个新方案．如果这个新的解 x' 更优，那么转移到 x'，否则不变．

这种算法对于单峰函数显然可行．

Q：都知道是单峰函数了为什么不三分呢?

A：爬山算法的优势在于当正解的写法你并不了解（常见于毒瘤计算几何和毒瘤数学题），或者本身状态维度很多，无法容易地写分治（例 2 就可以用二分完成合法正解）时，可以通过非常暴力的计算得到最优解．

但是对于多数需要求解的函数，爬山算法很容易进入一个局部最优解，如下图（最优解为 ⇑，而爬山算法可能找到的最优解为 ⇓）．



具体实现

爬山算法一般会引入温度参数（类似模拟退火）．类比地说，爬山算法就像是一只兔子喝醉了在山上跳，它每次都会朝着它所认为的更高的地方（这往往只是个不准确的趋势）跳，显然它有可能一次跳到山顶，也可能跳过头翻到对面去．不过没关系，兔子翻过去之后还会跳回来．显然这个过程很没有用，兔子永远都找不到出路，所以在这个过程中兔子冷静下来并在每次跳的时候更加谨慎，少跳一点，以到达合适的最优点．

兔子逐渐变得清醒的过程就是降温过程，即温度参数在爬山的时候会不断减小．

关于降温：降温参数是略小于 1 的常数，一般在 $[0.985, 0.999]$ 中选取．

例题

「JSOI2008」球形空间产生器[1]

给出 n 维空间中的 $n+1$ 个点，已知它们在同一个 n 维球面上，求出球心．$n \leq 10$，坐标绝对值不超过 20000．

解答

很明显的单峰函数，可以使用爬山解决．本题算法流程：

1. 初始化球心为各个给定点的重心（即其各维坐标均为所有给定点对应维度坐标的平均值），以减少枚举量．
2. 对于当前的球心，求出每个已知点到这个球心欧氏距离的平均值．
3. 遍历所有已知点．记录一个改变值 $cans$（分开每一维度记录）对于每一个点的欧氏距离，如果大于平均值，就把改变值加上差值，否则减去．实际上并不用判断这个大小问题，只要不考虑绝对值，直接用坐标计算即可．这个过程可以形象地转化成一个新的球心，在空间里推来推去，碰到太远的点就往点的方向拉一点，碰到太近的点就往点的反方向推一点．
4. 将我们记录的 $cans$ 乘上温度，更新球心，回到步骤 2
5. 在温度小于某个给定阈值的时候结束．

因此，我们在更新球心的时候，不能直接加上改变值，而是要加上改变值与温度的乘积．

并不是每一道爬山题都可以具体地用温度解决，这只是一个例子．

参考代码

```cpp
#include <cmath>
```

```cpp
#include <iomanip>
#include <iostream>
using namespace std;
double ans[10001], cans[100001], dis[10001], tot, f[1001][1001];
int n;

void check() {
  tot = 0;
  for (int i = 1; i <= n + 1; i++) {
    dis[i] = 0;
    cans[i] = 0;
    for (int j = 1; j <= n; j++)
      dis[i] += (f[i][j] - ans[j]) * (f[i][j] - ans[j]);
    dis[i] = sqrt(dis[i]);  // 欧氏距离
    tot += dis[i];
  }
  tot /= (n + 1);  // 平均
  for (int i = 1; i <= n + 1; i++)
    for (int j = 1; j <= n; j++)
      cans[j] += (dis[i] - tot) * (f[i][j] - ans[j]) /
                 tot;  // 对于每个维度把修改值更新掉，欧氏距离差*差值贡献
}

int main() {
  cin >> n;
  for (int i = 1; i <= n + 1; i++)
    for (int j = 1; j <= n; j++) {
      cin >> f[i][j];
      ans[j] += f[i][j];
    }
  for (int i = 1; i <= n; i++) ans[i] /= (n + 1);       // 初始化
  for (double t = 10001; t >= 0.0001; t *= 0.99995) {  // 不断降温
    check();
    for (int i = 1; i <= n; i++) ans[i] += cans[i] * t;  // 修改
  }
  cout << fixed << setprecision(3);
  for (int i = 1; i <= n; i++) cout << ans[i] << ' ';
}

```

「BZOJ 3680」吊打 XXX[2]

求 n 个点的带权类费马点.

解答

框架类似，用了点物理知识.

参考代码

```cpp
#include <cmath>
#include <iomanip>
#include <iostream>
constexpr int N = 10005;
int n, x[N], y[N], w[N];
double ansx, ansy;

void hillclimb() {
  double t = 1000;
  while (t > 1e-8) {
    double nowx = 0, nowy = 0;
    for (int i = 1; i <= n; ++i) {
      double dx = x[i] - ansx, dy = y[i] - ansy;
      double dis = sqrt(dx * dx + dy * dy);
      nowx += (x[i] - ansx) * w[i] / dis;
      nowy += (y[i] - ansy) * w[i] / dis;
    }
    ansx += nowx * t, ansy += nowy * t;
    if (t > 0.5)
      t *= 0.5;
    else
      t *= 0.97;
  }
}

int main() {
  std::cin.tie(nullptr)->sync_with_stdio(false);
  std::cin >> n;
  for (int i = 1; i <= n; ++i) {
    std::cin >> x[i] >> y[i] >> w[i];
    ansx += x[i], ansy += y[i];
  }
  ansx /= n, ansy /= n;
  hillclimb();
  std::cout << std::fixed << std::setprecision(3) << ansx << ' ' << ansy
            << '\n';
  return 0;
}
```

优化

很容易想到的是，为了尽可能获取优秀的答案，我们可以多次爬山．方法有修改初始状态/修改降温参数/修改初始温度等，然后开一个全局最优解记录答案．每次爬山结束之后，更新全局最优解．

这样处理可能会存在的问题是超时，在正式考试时请手造大数据测试调参．

劣势

其实爬山算法的劣势上文已经提及：它容易陷入一个局部最优解．当目标函数不是单峰函数时，这个劣势是致命的．因此我们要引进 **模拟退火**→13.6.4@p3278．

外部参考资料

[1]「JSOI2008」球形空间产生器  [2]「BZOJ 3680」吊打 XXX 

13.6.4 模拟退火

引入

模拟退火是一种随机化算法. 当一个问题的方案数量极大(甚至是无穷的)而且不是一个单峰函数时, 我们常使用模拟退火求解.

解释

根据 爬山算法→13.6.3@p3274 的过程, 我们发现: 对于一个当前最优解附近的非最优解, 爬山算法直接舍去了这个解. 而很多情况下, 我们需要去接受这个非最优解从而跳出这个局部最优解, 即为模拟退火算法.

> **什么是退火?(选自 百度百科[1])**
>
> 退火是一种金属热处理工艺, 指的是将金属缓慢加热到一定温度, 保持足够时间, 然后以适宜速度冷却. 目的是降低硬度, 改善切削加工性; 消除残余应力, 稳定尺寸, 减少变形与裂纹倾向; 细化晶粒, 调整组织, 消除组织缺陷. 准确的说, 退火是一种对材料的热处理工艺, 包括金属材料、非金属材料. 而且新材料的退火目的也与传统金属退火存在异同.

由于退火的规律引入了更多随机因素, 那么我们得到最优解的概率会大大增加. 于是我们可以去模拟这个过程, 将目标函数作为能量函数.

过程

先用一句话概括: 如果新状态的解更优则修改答案, 否则以一定概率接受新状态.

我们定义当前温度为 T, 新状态 S' 与已知状态 S (新状态由已知状态通过随机的方式得到) 之间的能量(值)差为 ΔE ($\Delta E \geqslant 0$), 则发生状态转移(修改最优解)的概率为

$$P(\Delta E) = \begin{cases} 1, & S' \text{ is better than } S, \\ \mathrm{e}^{\frac{-\Delta E}{T}}, & \text{otherwise.} \end{cases}$$

注意: 我们有时为了使得到的解更有质量, 会在模拟退火结束后, 以当前温度在得到的解附近多次随机状态, 尝试得到更优的解(其过程与模拟退火相似).

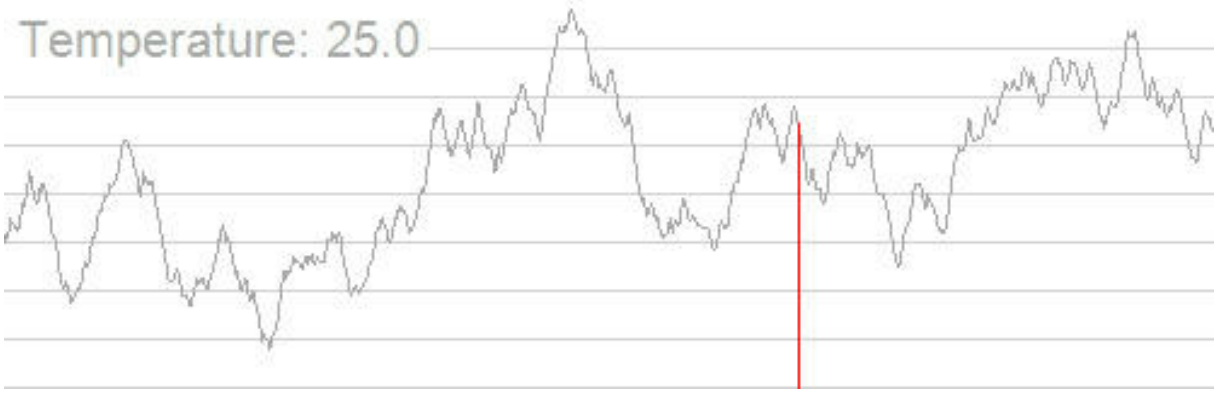
如何退火(降温)

模拟退火时我们有三个参数: 初始温度 T_0, 降温系数 d, 终止温度 T_k. 其中 T_0 是一个比较大的数, d 是一个非常接近 1 但是小于 1 的数, T_k 是一个接近 0 的正数.

首先让温度 $T = T_0$, 然后按照上述步骤进行一次转移尝试, 再让 $T = d \cdot T$. 当 $T < T_k$ 时模拟退火过程结束, 当前最优解即为最终的最优解.

注意为了使得解更为精确, 我们通常不直接取当前解作为答案, 而是在退火过程中维护遇到的所有解的最优值.

引用一张 Simulated annealing - Wikipedia[2] 的图片(随着温度的降低, 跳跃越来越不随机, 最优解也越来越稳定).



实现

此处代码以「BZOJ 3680」吊打 XXX[3]（求 n 个点的带权类费马点）为例.

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>

constexpr int N = 10005;
int n, x[N], y[N], w[N];
double ansx, ansy, dis;

double Rand() { return (double)rand() / RAND_MAX; }

double calc(double xx, double yy) {
  double res = 0;
  for (int i = 1; i <= n; ++i) {
    double dx = x[i] - xx, dy = y[i] - yy;
    res += sqrt(dx * dx + dy * dy) * w[i];
  }
  if (res < dis) dis = res, ansx = xx, ansy = yy;
  return res;
}

void simulateAnneal() {
  double t = 100000;
  double nowx = ansx, nowy = ansy;
  while (t > 0.001) {
    double nxtx = nowx + t * (Rand() * 2 - 1);
    double nxty = nowy + t * (Rand() * 2 - 1);
    double delta = calc(nxtx, nxty) - calc(nowx, nowy);
    if (exp(-delta / t) > Rand()) nowx = nxtx, nowy = nxty;
    t *= 0.97;
  }
  for (int i = 1; i <= 1000; ++i) {
    double nxtx = ansx + t * (Rand() * 2 - 1);
    double nxty = ansy + t * (Rand() * 2 - 1);
    calc(nxtx, nxty);
  }
}

```

```cpp
int main() {
  std::cin.tie(nullptr)->sync_with_stdio(false);
  srand(0);  // 注意，在实际使用中，不应使用固定的随机种子。
  std::cin >> n;
  for (int i = 1; i <= n; ++i) {
    std::cin >> x[i] >> y[i] >> w[i];
    ansx += x[i], ansy += y[i];
  }
  ansx /= n, ansy /= n, dis = calc(ansx, ansy);
  simulateAnneal();
  std::cout << std::fixed << std::setprecision(3) << ansx << ' ' << ansy
            << '\n';
  return 0;
}

```

一些技巧

分块模拟退火

有时函数的峰很多，模拟退火难以跑出最优解.

此时可以把整个值域分成几段，每段跑一遍模拟退火，然后再取最优解.

卡时

有一个 `clock()` 函数，返回程序运行时间.

可以把主程序中的 `simulateAnneal();` 换成 `while ((double)clock()/CLOCKS_PER_SEC < MAX_TIME) simulateAnneal();`. 这样子就会一直跑模拟退火，直到用时即将超过时间限制.

这里的 `MAX_TIME` 是一个自定义的略小于时限的数（单位：秒）.

习题

- 「BZOJ 3680」吊打 XXX[4]
- 「JSOI 2016」炸弹攻击[5]
- 「HAOI 2006」均分数据[6]

外部参考资料

[1] 百度百科 [2] Simulated annealing - Wikipedia [3] 「BZOJ 3680」吊打 XXX

[4] 「BZOJ 3680」吊打 XXX [5] 「JSOI 2016」炸弹攻击 [6] 「HAOI 2006」均分数据

13.7 悬线法

Authors: mwsht, sshwy, ouuan, Ir1d, Henry-ZHR, hsfzLZH1

引入

悬线法的适用范围是单调栈的子集. 具体来说，悬线法可以应用于满足以下条件的题目：

- 需要在扫描序列时维护单调的信息；
- 可以使用单调栈解决；
- 不需要在单调栈上二分.

看起来悬线法可以被替代，用处不大，但是悬线法概念比单调栈简单，更适合初学 OI 的选手理解并解决最大子矩阵等问题.

例题

SPOJ HISTOGRA - Largest Rectangle in a Histogram[1]

大意：在一条水平线上有 n 个宽为 1 的矩形，求包含于这些矩形的最大子矩形面积.

悬线，就是一条竖线，这条竖线有初始位置和高度两个性质，可以在其上端点不超过当前位置的矩形高度的情况下左右移动.

对于一条悬线，我们在这条上端点不超过当前位置的矩形高度且不移出边界的前提下，将这条悬线左右移动，求出其最多能向左和向右扩展到何处，此时这条悬线扫过的面积就是包含这条悬线的尽可能大的矩形. 容易发现，最大子矩形必定是包含一条初始位置为 i，高度为 h_i 的悬线. 枚举实现这个过程的时间复杂度为 $O(n^2)$，但是我们可以用悬线法将其优化到 $O(n)$.

我们考虑如何快速找到悬线可以到达的最左边的位置.

过程

定义 l_i 为当前找到的 i 位置的悬线能扩展到的最左边的位置，容易得到 l_i 初始为 i，我们需要进一步判断还能不能进一步往左扩展.

- 如果当前 $l_i = 1$，则已经扩展到了边界，不可以.
- 如果当前 $a_i > a_{l_i-1}$，则从当前悬线扩展到的位置不能再往左扩展了.
- 如果当前 $a_i \le a_{l_i-1}$，则从当前悬线还可以往左扩展，并且 $l_i - 1$ 位置的悬线能向左扩展到的位置，i 位置的悬线一定也可以扩展到，于是我们将 l_i 更新为 l_{l_i-1}，并继续执行判断.

通过摊还分析，可以证明每个 l_i 最多会被其他的 l_j 遍历到一次，因此时间复杂度为 $O(n)$.

实现

参考代码

```cpp
#include <algorithm>
#include <iostream>
using std::max;
constexpr int N = 100010;
int n, a[N];
int l[N], r[N];
long long ans;
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> n, n) {
    ans = 0;
    for (int i = 1; i <= n; i++) cin >> a[i], l[i] = r[i] = i;
    for (int i = 1; i <= n; i++)
      while (l[i] > 1 && a[i] <= a[l[i] - 1]) l[i] = l[l[i] - 1];
    for (int i = n; i >= 1; i--)
      while (r[i] < n && a[i] <= a[r[i] + 1]) r[i] = r[r[i] + 1];
    for (int i = 1; i <= n; i++)
```

```
      ans = max(ans, (long long)(r[i] - l[i] + 1) * a[i]);
    cout << ans << '\n';
  }
  return 0;
}
```

UVa1619 感觉不错 Feel Good[2]

对于一个长度为 n 的数列，找出一个子区间，使子区间内的最小值与子区间内元素和的乘积最大，要求在满足舒适值最大的情况下最小化长度，最小化长度的情况下最小化左端点序号.

本题中我们可以考虑枚举最小值，将每个位置的数 a_i 当作最小值，并考虑从 i 向左右扩展，找到满足 $\min\limits_{j=l}^{r} a_j = a_i$ 的尽可能向左右扩展的区间 $[l, r]$. 这样本题就被转化成了悬线法模型.

参考代码

```
#include <cstring>
#include <iostream>
constexpr int N = 100010;
int n, a[N], l[N], r[N];
long long sum[N];
long long ans;
int ansl, ansr;
bool fir = true;
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> n) {
    memset(a, -1, sizeof(a));
    if (!fir)
      cout << '\n';
    else
      fir = false;
    ans = 0;
    ansl = ansr = 1;
    for (int i = 1; i <= n; i++) {
      cin >> a[i];
      sum[i] = sum[i - 1] + a[i];
      l[i] = r[i] = i;
    }
    for (int i = 1; i <= n; i++)
      while (a[l[i] - 1] >= a[i]) l[i] = l[l[i] - 1];
    for (int i = n; i >= 1; i--)
      while (a[r[i] + 1] >= a[i]) r[i] = r[r[i] + 1];
    for (int i = 1; i <= n; i++) {
      long long x = a[i] * (sum[r[i]] - sum[l[i] - 1]);
      if (ans < x || (ans == x && ansr - ansl > r[i] - l[i]))
        ans = x, ansl = l[i], ansr = r[i];
    }
```

```
    cout << ans << '\n' << ansl << ' ' << ansr << '\n';
  }
  return 0;
}
```

最大子矩形

P4147 玉蟾宫[3]

给定一个 $n \times m$ 的包含 `'F'` 和 `'R'` 的矩阵，求其面积最大的子矩阵的面积 $\times 3$，使得这个子矩阵中的每一位的值都为 `'F'`.

我们会发现本题的模型和第一题的模型很像．仔细分析，发现如果我们每次只考虑某一行的所有元素，将位置 (x, y) 的元素尽可能向上扩展的距离作为该位置的悬线长度，那最大子矩阵一定是这些悬线向左右扩展得到的尽可能大的矩形中的一个.

参考代码

```cpp
#include <algorithm>
#include <iostream>
int m, n, a[1010], l[1010], r[1010], ans;
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      l[j] = r[j] = j;
    }
    char s[3];
    for (int j = 1; j <= m; j++) {
      cin >> s;
      if (s[0] == 'F')
        a[j]++;
      else if (s[0] == 'R')
        a[j] = 0;
    }
    for (int j = 1; j <= m; j++)
      while (l[j] != 1 && a[l[j] - 1] >= a[j]) l[j] = l[l[j] - 1];
    for (int j = m; j >= 1; j--)
      while (r[j] != m && a[r[j] + 1] >= a[j]) r[j] = r[r[j] + 1];
    for (int j = 1; j <= m; j++) ans = std::max(ans, (r[j] - l[j] + 1)
* a[j]);
  }
  cout << ans * 3;
  return 0;
}

```

习题

- P1169「ZJOI2007」棋盘制作[4]

外部参考资料

[1] SPOJ HISTOGRA - Largest Rectangle in a Histogram

[2] UVa1619 感觉不错 Feel Good

[3] P4147 玉蟾宫

[4] P1169「ZJOI2007」棋盘制作

13.8 有限状态自动机

Authors: CCXXXI, countercurrent-time, Enter-tainer, FFjet, H-J-Granger, Ir1d, mgt, NachtgeistW, orzAtalod, ouuan, SukkaW

前置知识：语言和判定问题→13.9@p3306

有限状态自动机（Finite State Machine，FSM，以下也简称自动机）是最简单的一类计算模型，体现在它的描述能力与资源都极其有限．自动机广泛应用在 OI、计算机科学中，其思想在许多字符串算法中都有涉及，因此推荐在学习一些字符串算法（KMP→8.7@p917、AC 自动机→8.10@p951、SAM→8.12@p999）前先完成自动机的学习．

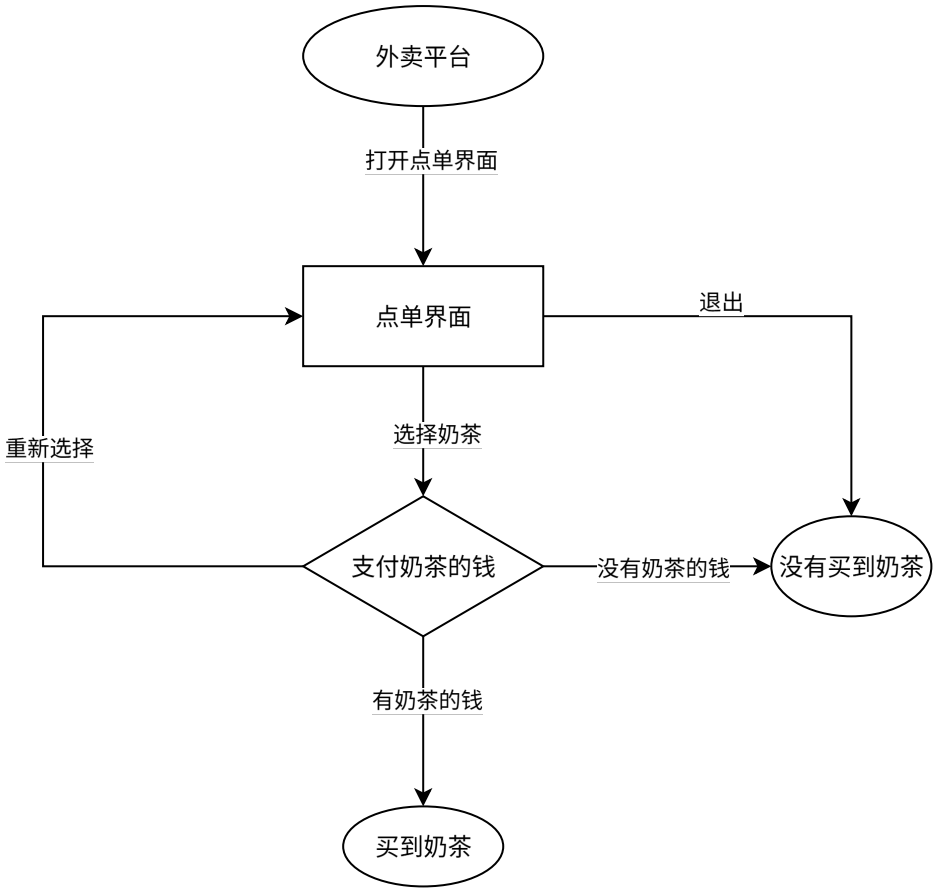
自动机入门

首先，我们来理解自动机是用来做什么的：自动机是一种判断一个信号序列是否满足某种特定模式或规则的数学模型．

这句话中的一些术语可以具体解释一下．「信号序列」指的是一串按顺序排列的信号，例如字符串从前到后的每一个字符、数组从 1 到 n 的每一个数、数从高到低的每一位等．「判断是否满足某种规则」，可以理解为：我们关心这个序列是否属于某个特定的集合．这个集合由我们事先设定好的规则来定义，例如「所有长度为偶数的二进制串」或「所有回文串」．

有时我们需要回答这类问题：一个给定的序列，是否满足某种特性？例如，一个二进制数是否是奇数，一个字符串是否是回文，或是否是另一个字符串的子序列等等．自动机就是用来解决这类问题的数学工具．

自动机的工作原理和流程图很类似．假设你想要在外卖平台点购一杯奶茶，你的所有选择就构成了一个序列．以下这个流程图是一个例子：



例如，你的选择序列是「打开点单界面 -> 选择奶茶 -> 有奶茶的钱」，那你按顺序经过的状态可能是「外卖平台 -> 点单界面 -> 支付奶茶的钱 -> 买到奶茶」. 就这样，我们的这个「奶茶自动机」根据我们的选择，帮我们判定了我们是否买到了奶茶. 我们还可以发现，到达一个状态的方法可能不止一条. 同样没有买到奶茶，你可能是在点单界面直接退出，或者没有奶茶的钱以至于没有买到奶茶.

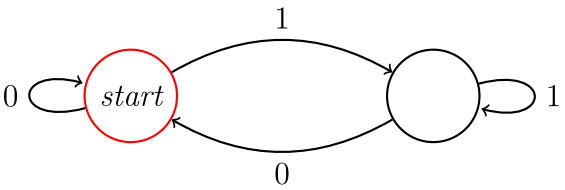
我们通过这个自动机，将信号序列分成了两类：一类是买到了奶茶的信号序列，一类是没有买到奶茶的信号序列. 根据最后位于的状态的不同，我们就完成了一个判定问题.

虽然我们刚才用流程图来类比自动机的工作过程，但流程图本身只是一个直观的可视化工具，并不构成对自动机的数学定义. 为了更准确地刻画自动机的结构，我们需要对流程图中的元素进行抽象. 抽象之后，我们发现流程图的结构其实可以简化为一个有向图，其中每个结点表示一种状态，每条有向边表示状态之间的转换.

因此，自动机的核心结构可以形式化地看作是一张有向图，我们称之为 **状态图**.

自动机的工作方式和流程图类似，不同的是：自动机的每一个结点都是一个判定结点；自动机的结点只是一个单纯的状态而非任务；自动机的边可以接受多种字符（不局限于 `T` 或 `F`）.

举个例子，完成「判断一个二进制数是不是偶数」的自动机如下：



从起始结点开始，从高到低接受这个数的二进制序列，然后看最终停在哪里. 如果最终停在红圈结点，就是偶数；否则不是.

在这里，我们需要强调，下文中我们会多次提到「字符」、「字符集」之类的名词，这不代表自动机只能应用于字符串领域，字符不一定是 abc…z 之类的字母，也可以是一种选择.

如果需要判定一个有限的信号序列和另外一个信号序列的关系（例如这个有限的信号序列是不是另一个信号序列的子串），那么常用的方法是针对这个有限的信号序列构建一个自动机. 这个在学习 KMP 的时候会讲到.

需要注意的是，自动机只是一个 **数学模型**，而 **不是算法**，也 **不是数据结构**. 实现同一个自动机的方法有很多种，可能会有不一样的时空复杂度.

接下来你可以选择在本页面继续进一步研究自动机，也可以去学习 KMP→8.7@p917、AC 自动机→8.10@p951 或 SAM→8.12@p999 等具体的例子.

FSM 分为两类：确定性有限状态自动机、非确定性有限状态自动机.

确定性有限状态自动机

确定性有限状态自动机（Deterministic Finite Automaton，DFA）体现在它的判定过程是确定性的. 以「奶茶自动机」为例子，你只要打开点单界面，就会进入点单界面，不会出现网络崩溃打不开、手机没电黑屏了之类的意外情况.

DFA

DFA 是一个五元组 $(Q, \Sigma, \delta, q_0, F)$，包括：

1. **有限状态集合** Q. 如果把一个 DFA 看成一张有向图，那么 DFA 中的状态就相当于图上的顶点.
2. **字符集** Σ. 该自动机只能输入这些字符.
3. **转移函数** $\delta : Q \times \Sigma \to Q$ 是一个接受两个参数返回一个值的函数，其中第一个参数和返回值都是一个状态，第二个参数是字符集中的一个字符. 如果把一个 DFA 看成一张有向图，那么 DFA 中的转移函数就相当于顶点间的边，而每条边上都有一个字符.
4. **起始状态** $q_0 \in Q$ 是一个特殊的状态. 在不同文章中，起始状态一般用 s、$start$、q_0 表示，本文中选择使用 q_0 表示.
5. **接受状态集合** $F \subseteq Q$ 是一组特殊的状态.

DFA 可以简单地用以下结构体表示：

参考实现

```cpp
// Deterministic Finite Automaton (DFA)
struct DFA {
  int m;                                  // Alphabet size.
  int n;                                  // Number of states.
  int q0;                                 // Initial state.
  std::vector<std::vector<int>> trans;    // Transitions: trans[c][q].
  std::vector<int> acc;                   // Acceptance labels per state:
                                          // - 0 = non-accepting

  DFA(int m, int n = 0, int q0 = 0)
      : m(m), n(n), q0(q0), trans(m, std::vector<int>(n)), acc(n) {}

  // Returns minimized DFA via Hopcroft's algorithm.
  DFA hopcroft_minimize() const;
};

```

求出输入串 w 在 DFA 中的状态序列，并判断它是否被接受的过程称为 **计算**.

自 DFA 的计算流程

设 $M=(Q,\Sigma,\delta,q_0,F)$ 是一个 DFA，$w=w_1w_2\cdots w_n\in\Sigma^*$ 是一个串. 若存在 Q 中的状态序列 $r_0,r_1,\cdots,r_n$ 满足

- $r_0=q_0$,
- $\delta(r_i,w_{i+1})=r_{i+1}$ 对于任何 $i=0,1,\cdots,n-1$ 都成立,
- $r_n\in F$,

则称 M **接受**（accepts）w. 反之，则称 M **不接受** w.

当一个 DFA 读入一个字符串时，从初始状态起按照转移函数一个一个字符地转移. 如果读入完一个字符串的所有字符后位于一个接受状态，那么我们称这个 DFA **接受** 这个字符串，反之我们称这个 DFA **不接受** 这个字符串.

自 形式语言

字符集合 Σ 上的一个 **形式语言**（language），或简称 **语言**，是 Σ 上字符串的一个集合 L.

自 自动机识别的语言

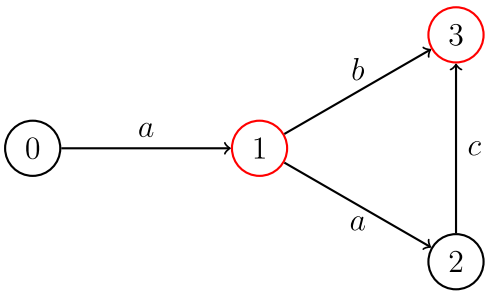
对于一个自动机 M，它识别的语言 $L(M)$ 就定义为它接受的全部字符串的集合 $\{w\mid M \text{ accepts } w\}$.

并非所有的语言都可以通过 DFA 识别.

自 正则语言

如果一个语言能由某个 DFA 识别，则称它为 **正则语言**（regular language），也称为正规语言.

上文提到过，一个自动机可以由状态图表示出来. 如下是一个接受且仅接受字符串 `a`、`ab`、`aac` 的 DFA：



（图中省略了失配状态，所有未画出的转移均指向该失配状态）

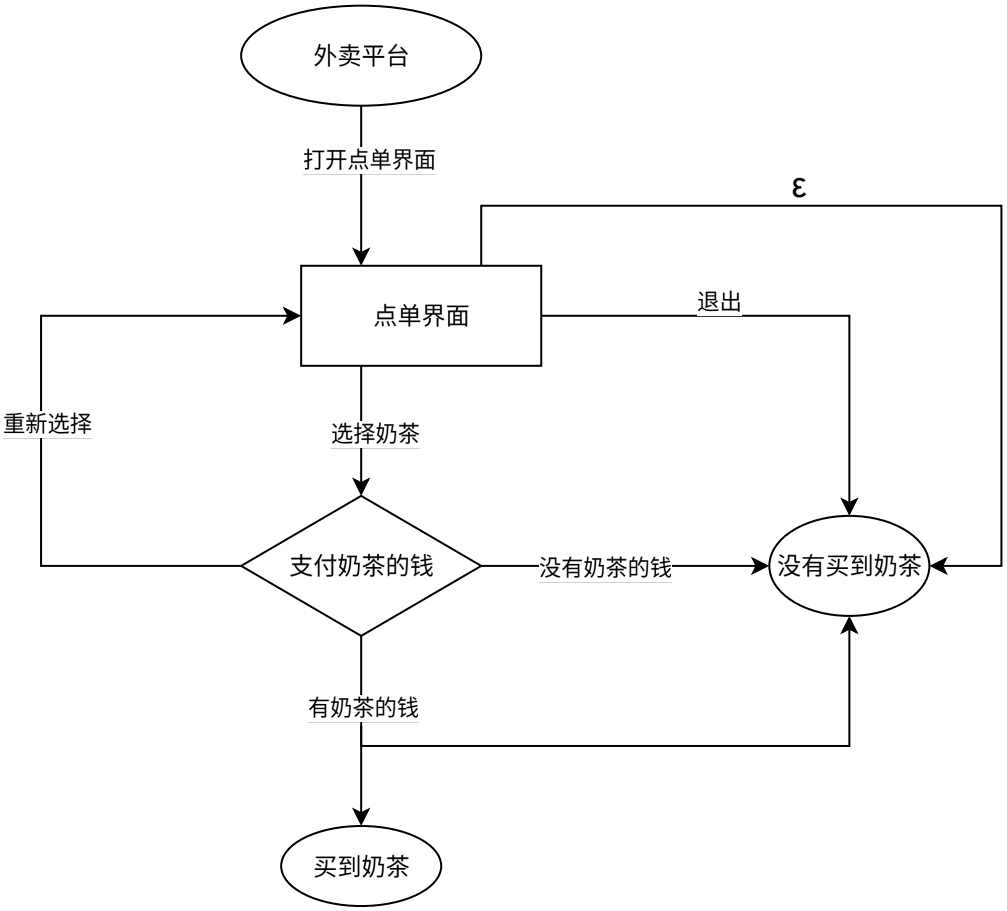
非确定性有限状态自动机

非确定性有限状态自动机[1]（Nondeterministic Finite Automaton，NFA）是 DFA 的自然推广. 在 NFA 中，对于任意状态和任意字符，都可能存在零个、一个或多个后继状态. 同时，本节讨论的 NFA 允许接受空字符，也就是说，可以在不消耗任何字符的情况下，由一个状态转移到它的某个后继状态.

举个例子，还是「奶茶自动机」. 下单后，尽管有奶茶的钱，却有可能因为网络不佳从而没有买到奶茶，这是存在多个后继；也有可能因为手速慢了，尽管输入的串（即操作序列）是一样的，却因为奶茶售

[1]这个定义中我们允许状态之间通过空字符（ε）转移，因此更准确地说，这是一个带 ε 转移的非确定有限自动机（NFA-ε）. 有些教材中将它直接称为 NFA，为简洁起见，本文采用这一用法. 在理论上 NFA 与 NFA-ε 是有所区分的，但是实际上它们的计算能力是一致的.

完从而没有买到奶茶，这就是空字符的存在，空字符可走可不走．对前文的自动机稍加修改即可实现上述功能：



显然，所有的 DFA 都是一个 NFA，所以 NFA 至少可以识别所有正则语言．但是，作为 DFA 的一个扩展，NFA 是否能够识别更多的语言呢？其实不然，我们之后将探讨 DFA 与 NFA 的等价性．

自 NFA

令 $\mathcal{P}(Q)$ 表示 Q 的幂集．令 $\varepsilon \notin \Sigma$ 表示空串，并记 $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$．NFA 是一个五元组 $(Q, \Sigma, \delta, q_0, F)$，包括：

1. **有限状态集合** Q，
2. **字符集** Σ，
3. **转移函数** $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$，一个接受两个参数返回一个 **状态集合** 的函数，其中第一个参数是一个状态，第二个参数是字符集中的一个字符或空串 ε，而返回值则是所有可能的后继状态形成的集合（可能为空），
4. **起始状态** $q_0 \in Q$，
5. **接受状态集合** $F \subseteq Q$．

NFA 的计算过程，相当于同时运行多个 DFA．每一步操作都穷举所有的可能性，最后，只要有一条分支到达了接受状态，NFA 就接受整个串．

自 NFA 的计算流程

设 $N = (Q, \Sigma, \delta, q_0, F)$ 是一个 NFA，若串 w 可以表示为某个 $y_1 y_2 \cdots y_m$（其中 $y_i \in \Sigma_\varepsilon$），且存在 Q 中的状态序列 $r_0, r_1, \cdots, r_m$ 满足

- $r_0 = q_0$,
- $r_{i+1} \in \delta(r_i, y_{i+1})$ 对于任何 $i = 0, 1, \cdots, m-1$ 都成立,
- $r_m \in F$,

则称 N **接受** w. 反之，则称 N **不接受** w.

由于允许空字符，将串 w 表示为 $y_1 y_2 \cdots y_m \in \Sigma_\varepsilon^*$ 时，可以插入任意多的空字符. 例如，字符串 `abc` 可以表示为 $\texttt{a}\varepsilon\ \texttt{bc}\varepsilon\varepsilon \in \Sigma_\varepsilon^*$. 相较于 DFA 的每一次输入只对应一个结果，NFA 的每次输入可能对应多个结果，形成一个结果集.

DFA 与 NFA 的等价性

我们称两个自动机等价，当且仅当它们能识别的语言相同. DFA 与 NFA 是等价的，即每一个 NFA 都等价于某一个 DFA；因此，NFA 识别的语言类也是全体正则语言. 每个 DFA 都可以直接看作一个 NFA；反过来，可以通过 **幂集构造**（powerset construction）的方法将一个 NFA 转换为 DFA.

自 幂集构造

假设 NFA 为 $N = (Q, \Sigma, \delta, q_0, F)$. 定义 $E(q)$ 表示从状态 q 出发，只沿 ε 转移能到达的状态集合（包括 q 本身），即 q 的 ε 闭包.

构造 DFA 为 $M = (Q', \Sigma, \delta', E(q_0), F')$，其中:

- **有限状态集合** $Q' = \mathcal{P}(Q)$,
- **转移函数** $\delta' : Q' \times \Sigma \to Q'$ 满足 $\delta'(S, c) = \bigcup_{q \in S,\ q' \in \delta(q,c)} E(q')$,
- **接受状态集合** $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$.

显然，计算的每一步中，M 所在的状态都对应 N 可能处于的状态集合.

虽然 NFA 与 DFA 识别语言的能力相同，但 NFA 仍然是有用的. 这是因为对于某些正则语言，用 NFA 表示所需的状态数远小于 DFA 所需的状态数. 例如，可以构造出一个状态数为 n 的 NFA 使得它对应的最小 DFA 状态数是 $\Theta(2^n)$ 的. 此时直接计算 NFA 的时间复杂度是更优的.

计算 DFA 与 NFA 的时间复杂度

设给定的串长为 n，自动机状态数为 s，字符集大小为常数. 那么显然地，DFA 计算的时间复杂度为 $O(n)$，只需要模拟上述的过程即可.

朴素计算 NFA 的时间复杂度为 $O(ns^2)$，这是因为需要考虑到每一种后继，以及状态的合并所需的复杂度. 当然，可以使用 bitset 或者 Method of Four Russians 将计算的复杂度优化到 $O\left(\dfrac{ns^2}{w}\right)$ 或 $O\left(\dfrac{ns^2}{w \cdot \log n}\right)$.

正则表达式与正则语言

本节将讨论正则表达式和正则语言的定义、性质，并研究正则表达式与 FSM 的关系.

正则表达式

正则表达式（regular expression）是另一种常用的正则语言的描述方法. 尽管我们可以在许多现代语言（例如 Python）中看到这个名字，但实际上这些语言实现的是正则表达式的一个超集.

自 正则表达式

给定一个字符集 Σ，正则表达式是由以下规则归纳定义的符号串:

1. 任意字符 $c \in \Sigma$ 是一个正则表达式；
2. 空串符号 ε 是正则表达式；
3. 空语言符号 $\emptyset$ 是正则表达式；
4. 如果 R_1 和 R_2 是正则表达式，那么 $(R_1 + R_2)$、(R_1R_2)（也记作 $(R_1 \cdot R_2)$）、(R_1^*) 都是正则表达式.

正则表达式的目标是通过这些符号描述一个语言．每个正则表达式都有一个对应的形式语言.

正则表达式所表示的语言

设每个正则表达式 R 对应的形式语言为 $L(R)$，则有：

1. 若 $R = c$，其中 $c \in \Sigma$，则 $L(R) = \{c\}$；
2. 若 $R = \varepsilon$，则 $L(R) = \{\varepsilon\}$；
3. 若 $R = \emptyset$，则 $L(R) = \emptyset$；
4. 若 $R = (R_1 + R_2)$，则 $L(R) = L(R_1) \cup L(R_2)$；
5. 若 $R = (R_1R_2) = (R_1 \cdot R_2)$，则 $L(R) = \{uv \mid u \in L(R_1),\ v \in L(R_2)\}$，其中，$uv$ 指将两个串前后拼接在一起；
6. 若 $R = (R_1^*)$，则 $L(R) = \{u_1u_2\cdots u_n \mid u_i \in L(R_1), n \in \mathbf{N}_+\} \cup \{\varepsilon\}$，也称为 **Kleene 星号**（Kleene star）或 **Kleene 闭包**（Kleene closure），简称闭包.

当然，规定了运算的优先级后，这些小括号在不引起混淆时可以省略.

例子

设 $L(R_1) = \{0, 01\}$，$L(R_2) = \{\varepsilon, 1, 11, 111, ...\}$，则有：

- $L(R_1R_2) = \{0, 01, 011, 0111, ...\}$，
- $L(R_2^*) = L(R_2)$，
- $L(R_1 + R_2) = \{0, 01, \varepsilon, 1, 11, 111, ...\}$.

每个正则表达式都可以通过 Thompson 构造法[1]（Thompson's construction）转换为一个 NFA，每个 DFA 也都可以通过状态消除法[2]（State Elimination Method）转换为一个正则表达式．所以，正则表达式与 FSM 是等价的.

正则语言

在本小节中，我们不考虑具体的正则表达式，转而考虑以变量为参数的正则表达式（变量可以为任意正则语言）．运用正则表达式的代数定律有助于化简正则表达式.

正则语言的代数性质

1. 并的交换律：$L + M = M + L$
2. 并的结合律：$(L + M) + N = L + (M + N)$
3. 连接的结合律：$(LM)N = L(MN)$
4. $\emptyset$ 是并运算的单位元：$\emptyset + L = L + \emptyset = L$
5. ε 是连接运算的单位元：$\varepsilon L = L\varepsilon = L$
6. $\emptyset$ 是连接运算的零因子：$\emptyset L = L\emptyset = \emptyset$
7. 分配律：$L(M + N) = LM + LN$，$(M + N)L = ML + NL$
8. 并的幂等律：$L + L = L$

[2]详见 国家集训队 2021 论文 徐哲安 浅谈有限状态自动机及其应用[2] 中 3.2 节.

9. 闭包相关的定律：$(L^*)^* = L^*$，$\emptyset^* = \varepsilon$，$\varepsilon^* = \varepsilon$

正则语言的 **封闭性** 也是重要的性质. 这些性质允许我们从一些简单的自动机出发, 通过一定的运算, 构造能够识别另一些语言的有限状态机（FSM）. 简而言之，封闭性可以作为构造复杂 FSM 的工具.

关于正则语言的封闭性，我们有：

正则语言的封闭性

设 L, M 为字符集 Σ 上的两个正则语言，且映射 $h : \Sigma \to \Sigma^*$. 定义字符串 $s = s_1 s_2 \cdots s_n$ 的同态为 $h(s) = h(s_1)h(s_2)\cdots h(s_n)$. 那么，

1. 两个正则语言的并 $L + M$ 是正则的,
2. 两个正则语言的连接 LM 是正则的,
3. 正则语言的闭包 L^* 是正则的,
4. 正则语言的补 $\Sigma^* \setminus L$ 是正则的,
5. 两个正则语言的交 $L \cap M$ 是正则的,
6. 两个正则语言的差 $L \setminus M$ 是正则的,
7. 正则语言的反转 $L^R = \{s_n \cdots s_2 s_1 \mid s = s_1 s_2 \cdots s_n \in L\}$ 是正则的,
8. 正则语言的同态 $h(L) = \{h(s) \mid s \in L\}$ 是正则的,
9. 正则语言的逆同态 $h^{-1}(L) = \{s \in \Sigma^* \mid h(s) \in L\}$ 是正则的.

一个简单的推论是，所有的有限语言都是正则语言. 实际上，字典树 Trie→8.6@p903 就是一个识别它们的自动机.

Myhill–Nerode 定理

Myhill–Nerode 定理给出了一个语言是否是正则语言的判定标准. 该定理通过等价类的概念描述了正则语言的结构特征.

自 Nerode 等价关系

对于一个语言 L 和任意串 $x, y \in \Sigma^*$，如果对于任意 $z \in \Sigma^*$，都有 $xz \in L \iff yz \in L$，那么，称字符串 x 和 y 关于 L 是等价的，记作 $x \underset{L}{\equiv} y$.

也就是说，如果对于两个串 x 与 y，在 x 和 y 后面拼上相同的任意串 z（包括空串），它们总是要么同时属于 L，要么同时不属于 L，则我们说 x 与 y 关于 L 等价.

根据上述定义, 我们把所有有限字符串的集合划分成一个或多个等价类. 当且仅当这些等价类的数目只有有限多个时，可以利用这些等价类构造一个识别该语言的 DFA. 这个 DFA 的状态数目就等于等价类的数目. 而且，这个状态数目是所有能够识别该语言的 DFA 中最小的. 这就是 Myhill–Nerode 定理.

Myhill–Nerode 定理

一个语言 L 是正则的，当且仅当 Σ^* 通过等价关系 $\underset{L}{\equiv}$ 划分成的等价类数量是有限的.

对于任何能识别语言 L 的 DFA，任意两个能驱使它走到同一个状态的串 x 和 y 必在同一个等价类中.

进而，等价类的数量就是可以识别 L 的最小 DFA 的状态数量. 每个等价类都恰好对应最小 DFA 里的一个状态. 这个最小 DFA 在同构意义下是唯一的.

这个定理提供了一种方法，能够利用等价关系构造 DFA：

- 状态集合，就是根据等价关系划分得到的所有等价类. 每个等价类都随意选定一个代表字符串（例如某个串长最小的串）.

- 要构造转移函数，只需要将选定的代表字符串后面添加转移中的字符，并找到得到的字符串所在等价类对应的状态，就是相应的转移的后继状态．因为同一等价类中，所有字符串都是等价的，所以任意选定的代表字符串并不会影响转移的结果．
- 初始状态，就是空字符串 ε 对应的等价类．
- 接受状态集合，就是代表字符串属于所给语言的等价类的集合．

作为一个经典的例子，后缀自动机→8.12@p999 就是利用 Myhill–Nerode 定理构造出的最小 DFA．

Myhill–Nerode 定理通常应用于一些无限大的正则语言对应的 DFA 的构造．很多时候，问题的条件比较简单，只需要考察长度不太长的字符串的集合，就可以构造出识别整个语言的自动机．

例题

本节通过一道例题介绍如何实际应用 Myhill–Nerode 定理．

P12294 [THUPC 2025 决赛] 一个 01 串，n 次三目运算符，最后值为 1（加强版） [3]

关于 a,b,c 的三目运算表 $s_0s_1\cdots s_7$（s 仅由 $0,1$ 组成）的含义是，如果 s 的第 $a+2b+4c$ 位为 1，那么返回 1，否则返回 0．

给定运算表 s 以及 q 个长为 $2n+1$ 的 01 串，你需要对每个 01 串分别回答：

能否操作 n 次，每次将三位连续的数字替换为所对应的运算值，使得运算的结果为 1，给出方案，或判断无解．

$1\le 2n+1\le 10^5$，$\sum(2n+1)\le 3\times 10^5$．

题解

能够合成出 1 的 01 串集合是一个正则语言（也就是存在一个 DFA 能够判定一个 01 串能否合成出 1）[3]．故考虑使用 Myhill–Nerode 定理．因为条件比较简单，经过实验，我们只需要对于长度 ≤ 9 的 01 串进行等价类划分；判定两个串等价时，只需要往后枚举长度 ≤ 6 的后缀进行判定．只要两个串，接上长度 ≤ 6 的任意后缀，它们要么能够同时合成出我们想要的串，要么两个都不能合成我们想要的串，那么这两个串就是等价的．

每次转移都相当于在当前串后面添加一个新的 01 字符，然后将这个新串变为这个新串所在等价类中串长最小的串．根据上述转移设计一个自动机．该自动机能够在 $O(n)$ 复杂度内判定一个长度为 n 的串是否存在一种运算方式使得结果为 1．同时自动机的状态数非常少．

为了方便，我们会建 6 个自动机，这 6 个自动机分别表示能否通过一种运算方式使得结果为 $0,1,00,01,10,11$．对于所有可能的运算表，自动机的最大状态数目为 47．

利用自动机，通过适当的预处理，可以考虑使用倍增或者猫树实现静态区间查询：判断区间是否存在一种运算方式使得结果为 1，前者查询一次是 $O(\log n)$，后者查询一次是 $O(1)$ 的．

考虑使用分治解决构造问题．设 $f(l,r,t)$ 表示区间 $[l,r]$ 合并出 $t\in\{0,1,00,01,10,11\}$ 的方案．此时使用启发式分裂，维护两个指针 i,j 一个从左到右扫，一个从右往左扫，以枚举断点 mid 为 i 或 j．对于 $t\in\{0,1\}$，则枚举 t 是怎么分为左右两个部分的，其中一个部分的 t 长度为 2，另一个部分 t 长度为 1．（例如对于中位数的运算表 $s=00010111$，1 可以分为 01 和 1．）对于 $t\in\{00,01,10,11\}$，则 t 直接分为左右两个部分．

如果此时分为的左右两个部分分别为 t_1 与 t_2，则进一步判断 $[l,mid]$ 能否生成 t_1 和 $[mid+1,r]$ 能否生成 t_2，如果能则直接分治下去．如果使用 $O(1)$ 猫树判定，这么启发式分裂构造的复杂度是 $O(n\log n)$；否则，利用倍增判定，构造的复杂度就是 $O(n\log^2 n)$ 的．

[3] 详见 官方题解[4]．

如果使用了猫树，总复杂度是 $O(n \mid Q \mid \log n + n \log n)$，其中，$\mid Q \mid \leq 47$. 参考代码为了方便，使用了倍增，并且通过底层分块减小常数，对应的总复杂度为 $O(n \mid Q \mid \log n + n \log^2 n)$.

参考代码

```cpp
#include <array>
#include <iostream>
#include <unordered_map>
#include <vector>

constexpr int L = 3;
std::array<int, 1 << L> op;
constexpr int X = (1 << L) - 2;

// Build a DFA that accepts binary strings which evaluate to a given string.
//   0b000 → "00"
//   0b001 → "01"
//   0b010 → "10"
//   0b011 → "11"
//   0b100 → "0"
//   0b101 → "1"
DFA build(int x) {
  constexpr int L1 = 9, L2 = 6;
  // The result for string `st` of length `len` is stored at (1 << len) | st.
  const auto idx = [](int len, int st) -> int { return (1 << len) | st; };

  // Compute which bit strings are accepted using dynamic programming.
  std::vector<bool> acc(1 << (L1 + L2 + 1));
  acc[(x >> 2) ? idx(1, x & 1) : idx(2, x)] = true;

  for (int len = (x >> 2) ? 3 : 4; len <= L1 + L2; len += 2) {
    for (int st = 0; st < (1 << len); ++st) {
      for (int i = 0; i + L <= len; ++i) {
        // Replace st[i..i+2] using the op[] rule, and check if the result is
        // accepted. Result = st[0..i-1] + op[st[i..i+2]] + st[i+3..end]
        int pr =
            ((((st >> (i + L)) << 1) | op[(st >> i) & ((1 << L) - 1)]) << i) |
            (st & ((1 << i) - 1));
        if (acc[idx(len - 2, pr)]) {
          acc[idx(len, st)] = true;
          break;
        }
      }
    }
  }

  // Construct a DFA using the Myhill-Nerode theorem.
  DFA dfa(2);
  std::unordered_map<std::vector<bool>, int>
```

```
      mp;                // Maps characteristic vectors to state IDs.
  std::vector<int> sts;  // Representative 01 strings for each state.
  std::vector<int> ids(1 << (L1 + 1));  // Maps 01 strings to DFA state IDs.

  for (int len = 0; len <= L1; ++len) {
    for (int st = (1 << len); st < (2 << len); ++st) {
      // Construct characteristic vector for this prefix.
      std::vector<bool> key(1 << (L2 + 1));
      for (int nxt = 0; nxt <= L2; ++nxt)
        std::copy(acc.begin() + (st << nxt), acc.begin() + ((st + 1) << nxt),
                  key.begin() + (1 << nxt));
      if (!mp.count(key)) {
        mp[key] = dfa.n++;
        sts.push_back(st);
        dfa.acc.push_back(key[idx(0, 0)]);
      }
      ids[st] = mp[key];
    }
  }

  // Build transitions.
  for (int c = 0; c < 2; ++c) {
    dfa.trans[c].resize(dfa.n);
    for (int i = 0; i < dfa.n; ++i) dfa.trans[c][i] = ids[(sts[i] << 1) | c];
  }

  return dfa;
}

std::vector<DFA> dfa;

// Initialize the DFA's.
void init() {
  std::string s;
  std::cin >> s;
  for (int i = 0; i < (1 << L); ++i) op[i] = s[i] - '0';
  std::swap(op[1], op[4]);
  std::swap(op[3], op[6]);
  for (int x = 0; x < X; ++x) dfa.push_back(build(x));
}

std::string s;
constexpr int B = 5;  // Block size is 2^B.
std::vector<std::vector<std::array<std::vector<int>, X>>> pre;

void precompute() {
  int n = s.size();
  for (int i = 0; i < n; ++i) s[i] -= '0';
  int nn = ((n - 1) >> B) + 1;
  pre.clear();
```

```
95
96    // Compute transitions over single blocks (2^B characters) for each DFA.
97    pre.emplace_back(nn);
98    for (int i = 0; i < nn; ++i) {
99      for (int x = 0; x < X; ++x) {
100       pre[0][i][x].resize(dfa[x].n);
101       for (int j = 0; j < dfa[x].n; ++j) {
102         int cr = j;
103         for (int ii = (i << B); ii < ((i + 1) << B) && ii < n; ++ii)
104           cr = dfa[x].trans[s[ii]][cr];
105         pre[0][i][x][j] = cr;
106       }
107     }
108   }
109
110   // Build higher-level transitions using binary lifting.
111   for (int d = 1; (1 << d) <= nn; ++d) {
112     pre.emplace_back(nn - (1 << d) + 1);
113     for (int i = 0; i <= nn - (1 << d); ++i) {
114       for (int x = 0; x < X; ++x) {
115         pre[d][i][x].resize(dfa[x].n);
116         for (int j = 0; j < dfa[x].n; ++j)
117           pre[d][i][x][j] =
118               pre[d - 1][i + (1 << (d - 1))][x][pre[d - 1][i][x][j]];
119       }
120     }
121   }
122 }
123
124 // Check if the x-th DFA accepts s[l...r].
125 bool check(int l, int r, int x) {
126   if (l > r) return false;
127   int cr = dfa[x].q0;
128   if ((l >> B) == (r >> B)) {
129     for (; l <= r; ++l) cr = dfa[x].trans[s[l]][cr];
130   } else {
131     for (; (l & ((1 << B) - 1)) && (l <= r); ++l) cr = dfa[x].trans[s[l]]
    [cr];
132     int d = (r >> B) - (l >> B);
133     for (int z = 0; z < (int)pre.size(); ++z)
134       if ((d >> z) & 1) {
135         cr = pre[z][l >> B][x][cr];
136         l += 1 << (z + B);
137       }
138     for (; l <= r; ++l) cr = dfa[x].trans[s[l]][cr];
139   }
140   return dfa[x].acc[cr];
141 }
142
143 // Construct expression for s[l...r].
144 void calc(int l, int r, int x) {
145   if (l == r) return (void)(std::cout << (x & 1));
```

```
146   if (x >> 2) {
147     for (int i = l, j = r; i <= j; i += 2, j -= 2) {
148       for (int k = 0; k < (1 << L); ++k) {
149         if (op[k] != (x & 1)) continue;
150         if (check(l, i, (k >> 2) | 4) && check(i + 1, r, k & 3)) {
151           std::cout << '(';
152           calc(l, i, (k >> 2) | 4);
153           calc(i + 1, r, k & 3);
154           std::cout << ')';
155           return;
156         }
157         if (check(l, j - 1, k >> 1) && check(j, r, (k & 1) | 4)) {
158           std::cout << '(';
159           calc(l, j - 1, k >> 1);
160           calc(j, r, (k & 1) | 4);
161           std::cout << ')';
162           return;
163         }
164       }
165     }
166   } else {
167     for (int i = l, j = r; i <= j; i += 2, j -= 2) {
168       if (check(l, i, (x >> 1) | 4) && (check(i + 1, r, (x & 1) | 4))) {
169         calc(l, i, (x >> 1) | 4);
170         calc(i + 1, r, (x & 1) | 4);
171         return;
172       }
173       if (check(l, j - 1, (x >> 1) | 4) && check(j, r, (x & 1) | 4)) {
174         calc(l, j - 1, (x >> 1) | 4);
175         calc(j, r, (x & 1) | 4);
176         return;
177       }
178     }
179   }
180 }
181
182 void solve() {
183   std::cin >> s;
184   int n = s.size();
185   precompute();
186   if (!check(0, n - 1, 0b101)) {
187     std::cout << -1 << '\n';
188     return;
189   }
190   calc(0, n - 1, 0b101);
191   std::cout << '\n';
192 }
193
194 int main() {
195   init();
196   int q;
197   std::cin >> q;
```

```
198   for (; q; --q) {
199     solve();
200   }
201   return 0;
202 }
203
```

习题

- Median Replace Hard[5]
- JOISC 2024 卡牌收集[6]（通过 Myhill–Nerode 定理建立自动机，本题可以做到多次区间查询）

DFA 最小化

前文提到，两个 DFA 等价当且仅当它们识别相同的正则语言．根据识别的语言不同，全体 DFA 划分为无穷多个等价类．在进行 DP 套 DP 之类的算法时，建立出来的 DFA 的 $|Q|$ 可能过大，使得外层 DP 转移复杂度过大．因此，往往需要找到 DFA 所属等价类中的最小 DFA，以减少外层 DP 转移复杂度．

上一节的 Myhill–Nerode 定理就提供了一种构造方法．但是，对于一些比较复杂的问题，直接通过 Myhill–Nerode 定理构造需要遍历相当长的字符串的集合，花费大量时间．因此，我们需要一种方法，可以从已经构造出来的 DFA（通常较为容易）出发，直接构造一个最小 DFA．这就称为 **DFA 最小化**（DFA minimization）问题．

DFA 最小化常用的算法是 **Hopcroft 算法**．Myhill–Nerode 定理指出，对于任意一个可以识别某语言的 DFA，能够驱使它到达同一状态的字符串都必然是 Nerode 等价的．所有 Nerode 等价的字符串对应着最小 DFA 中的同一个状态，所以，最小 DFA 的状态一定是当前 DFA 中若干个状态的集合（这里假设当前 DFA 的所有状态都可以从 q_0 到达，否则需要先删去不可达的状态）．我们可以从已有的 DFA 的状态集合出发，将它们划分为若干个等价类，而无需考察具体的字符串．Hopcroft 算法从最粗糙的划分 $\{F, Q \setminus F\}$ 开始，利用一系列证据 A，改进这个划分，直到无法进一步改进为止．这就是 Hopcroft 算法的核心想法．

所谓 **证据** A，就是一个状态集合，而且它和它的补集 $Q \setminus A$ 一定对应着不同的 Nerode 等价类．也就是说，对于任意 $u \in A$ 和 $v \in Q \setminus A$，都存在某个字符串 $s \in \Sigma^*$，使得分别从 u 和 v 出发，读入字符串 s 后，恰有一个处于接受状态．因此，如果有两个状态 $u, v \in Q$，它们在某个字符 c 下恰好一个转移到证据 A 中，一个转移到证据 A 外，即 $\delta(u, c) \in A$ 和 $\delta(v, c) \in A$ 成立且仅成立一个，那么，u, v 同样不等价——分别从 u 和 v 出发读入字符串 cs 后，恰有一个处于接受状态．因此，利用是否成立 $\delta(u, c) \in A$ 这一点，就可以改进划分．具体地，设

$$S_x = \{u \mid u \in P_x,\ \delta(u, c) \in A\}.$$

如果 S_x 和 $P_x \setminus S_x$ 均不是空集，那么，当前的划分中状态集合 P_x 就可以改进为 S_x 和 $P_x \setminus S_x$．

最开始时，将接受状态集合 F 作为一个证据塞入证据集合 W，即 $W \leftarrow \{F\}$，并维护当前的划分为 $P \leftarrow \{F,\ Q \setminus F\}$．初始证据是显然成立的：$F$ 和 $Q \setminus F$ 中的状态绝不可能等价．每次都从证据集合 W 中随意取出一个集合 A 用于改进当前的划分．枚举所有的字符 $c \in \Sigma$．对当前划分 P 中的每个状态集合 P_x 都求出前文描述的 S_x．如果 $S_x \neq \emptyset$ 且 $|S_x| \neq |P_x|$，就意味着 P_x 可以进一步分为两个集合 S_x 和 $P_x \setminus S_x$，直接用它们替换掉 P 中的 P_x．

每当获得更细致的划分时，就意味着获得了新的证据．原则上，可以将新得到的 S_x 和 $P_x \setminus S_x$ 都塞进证据集合 W，等待后续进一步验证．但是，这样做并不是必要的．容易理解，对于三个证据 $P_x, S_x, P_x \setminus S_x$，只需要验证其中任意两个，就可以保证结果的正确性：因为结果只有 $\delta(u, c) \in S_x$、$\delta(u, c) \in P_x \setminus S_x$ 和 $\delta(u, c) \notin P_x$ 三种，而将集合划分成三部分只需要两次判断．因此，将 P_x 划分为 S_x 和 $P_x \setminus S_x$ 时，如果 P_x 仍处于证据集合 W 中，这说明还没有检验过证据 P_x，就需要将证据集合 W 中的 P_x 替换为 S_x 和 $P_x \setminus$

S_x 两个；否则，当前的划分一定相当于[4]已经检验过 P_x 的结果，所以，只需要将 S_x 和 $P_x \setminus S_x$ 中较小的那个塞入证据集合 W 中．类似于启发式分裂，这样做可以得到优秀的复杂度．

将上述过程写成伪代码就是：

Algorithm Hopcroft's Algorithm$(Q, \Sigma, \delta, q_0, F)$:
Input. DFA $M = (Q, \Sigma, \delta, q_0, F)$.
Output. A partition of Q into equivalence classes of the minimal DFA.
Method.

$P \leftarrow \{F, Q \setminus F\}$
$W \leftarrow \{F\}$
while $W \neq \emptyset$
  choose and remove any $A \in W$
  for each $c \in \Sigma$
    $S \leftarrow \{q \in Q \mid \delta(q, c) \in A\}$
    for each $Y \in P$ such that $S \cap Y \neq \emptyset$ and $Y \setminus S \neq \emptyset$
      $Y_1 \leftarrow S \cap Y,\ Y_2 \leftarrow Y \setminus S$
      $P \leftarrow (P \setminus \{Y\}) \cup \{Y_1, Y_2\}$
      if $Y \in W$
        $W \leftarrow (W \setminus \{Y\}) \cup \{Y_1, Y_2\}$
      else
        add the smaller of Y_1 and Y_2 to W
return P

算法实现时，复杂度的瓶颈在于 S 的计算．直接遍历所有 $q \in Q$ 进而判断 $\delta(q, c) \in A$ 是否成立是不可行的．因此，需要在算法运行前，预处理反向转移边 $\{q \in Q \mid \delta(q, c) = a\}$，从而，利用这些反向转移，遍历 $a \in A$，就可以得到集合 S．这样做可以保证每条转移 $\delta(q, c) = a$ 只会在 a 属于某个证据时才会遍历到；而前文的证据筛选方法保证了，算法中实际用到的包含 a 的证据序列 $A_1 \supset A_2 \supset \cdots \supset A_k$ 中，前一个至少是后一个的两倍大小，因此，$k \in O(\log |\ Q\ |)$．也就是说，每条转移边至多只会遍历 $O(\log |\ Q\ |)$ 次，而总的转移数目是 $|\ Q\ | \cdot |\ \Sigma\ |$ 的，因此，总的复杂度就是 $O(|\ Q\ | \cdot |\ \Sigma\ | \log |\ Q\ |)$ 的．

参考实现如下：[5]

参考实现

```cpp
// DFA minimization via Hopcroft's algorithm.
// Complexity: O(n * m * log(n)).
DFA DFA::hopcroft_minimize() const {
  // Construct inverse transition maps:
  // - pre[c] stores states sorted by the target of transition c.
  // - pos[c][s] is the start index in pre[c] of transitions going to states.
  std::vector<std::vector<int>> pre(m), pos(m);
  for (int c = 0; c < m; ++c) {
    pre[c].assign(n, 0);
    pos[c].assign(n + 1, 0);
    // Counting sort.
    for (int i = 0; i < n; ++i) ++pos[c][trans[c][i]];
    for (int i = 0; i < n; ++i) pos[c][i + 1] += pos[c][i];
```

[4]此处的「相当于」指的是，尽管 P_x 可能并没有实际检验过，但是，即使对当前划分进行 P_x 的检验，也不会有任何改进．简单理解，就是在集合分裂得到的证据集合的树上，它的某个祖先和路径上的所有旁支都已经得到了检验，因此，可以归纳地说明，就相当于它也已经检验过了．

[5]算法实现中有一处细节：对于一个证据 A，有可能检验完一部分字符后，这个证据集合就已经分裂为 B 和 C 了．不妨设 $|\ B\ | \geq |\ C\ |$．参考实现中，较小的集合 C 插入到了证据队列的末尾，而较大的证据集合 B 替换到了集合 A 原来的位置．算法继续运行时，实际只是利用证据 B 检验剩余的字符．这样做是正确的．这是因为对于已经检验完的字符，至少验证了 A 和 C 两个集合；而对于尚未检验的字符，至少验证了 B 和 C 两个集合．

```
    for (int i = 0; i < n; ++i) pre[c][--pos[c][trans[c][i]]] = i;
  }

  // Partition element structure:
  // - os: starting index in the state list.
  // - sz: number of states in this class.
  // - cnt: temporary count of marked states during refinement.
  // - mv: temporary count of marked states moved to the front.
  struct EquivClasses {
    int os, sz, cnt, mv;

    EquivClasses(int os, int sz, int cnt) : os(os), sz(sz), cnt(cnt), mv(0)
{}
  };

  // Partition and helper data structures.
  std::vector<EquivClasses> ec;  // Current list of equivalence classes.
  std::vector<int> ids(n);       // Permutation of states, grouped by ECs.
  std::vector<int> par(n);       // Maps state to its EC index.
  std::vector<int> loc(n);       // Position of each state in ids.
  std::vector<bool> tag(n);      // Temporary marking for splitting.
  std::queue<int> evidences;     // Worklist of ECs to check.

  // Initial partition by acceptance label.
  std::iota(ids.begin(), ids.end(), 0);
  std::sort(ids.begin(), ids.end(),
            [&](int l, int r) { return acc[l] < acc[r]; });
  for (int l = 0, r; l < n; l = r) {
    for (r = l; r < n && acc[ids[r]] == acc[ids[l]]; ++r)
      par[ids[r]] = ec.size();
    if (l) evidences.push(ec.size());  // Add all but first class to
worklist.
    ec.emplace_back(l, r - l, 0);
  }
  for (int i = 0; i < n; ++i) loc[ids[i]] = i;

  // Refinement loop.
  while (!evidences.empty()) {
    int cr = evidences.front();
    evidences.pop();
    for (int c = 0; c < m; ++c) {
      std::vector<int> todo, marked;
      for (int i = ec[cr].os; i < ec[cr].os + ec[cr].sz; ++i) {
        for (int k = pos[c][ids[i]]; k < pos[c][ids[i] + 1]; ++k) {
          int j = pre[c][k];
          if (!tag[j]) {
            if (!ec[par[j]].cnt) todo.push_back(par[j]);
            ++ec[par[j]].cnt;
            tag[j] = true;
            marked.push_back(j);
          }
        }
```

```
      }
      // Move marked states to the front of their ECs, in O(#marked).
      for (int j : marked) {
        int e = par[j], d = ec[e].os + ec[e].mv++;
        int k = ids[d];
        std::swap(ids[loc[j]], ids[d]);
        loc[k] = loc[j];
        loc[j] = d;
      }
      // Perform splits: the smaller segment becomes a new EC.
      for (int i : todo) {
        int cnt = ec[i].cnt;
        if (cnt != ec[i].sz) {
          bool small_tagged = cnt * 2 <= ec[i].sz;
          int l = small_tagged ? ec[i].os : ec[i].os + cnt;
          int len = small_tagged ? cnt : ec[i].sz - cnt;
          for (int j = l; j < l + len; ++j) par[ids[j]] = ec.size();
          evidences.push(ec.size());
          if (small_tagged) ec[i].os += cnt;
          ec[i].sz -= len;
          ec.emplace_back(l, len, 0);
        }
        ec[i].cnt = ec[i].mv = 0;
      }
      // Clear temporary tags.
      for (int j : marked) tag[j] = false;
    }
  }

  // Build minimized DFA.
  DFA res(m, ec.size(), par[q0]);
  for (const auto& e : ec) {
    int i = ids[e.os];  // Representative state.
    res.acc[par[i]] = acc[i];
    for (int c = 0; c < m; ++c) res.trans[c][par[i]] = par[trans[c][i]];
  }
  return res;
}

```

这一参考实现允许自动机的状态带有任何整数取值的标签，而并非简单的「接受」与「不接受」的二元标签. 从参考实现可以看出，与基础 Hopcroft 算法的唯一不同就在于初始划分和证据集合的构造. 这种拓展的自动机也称为 Moore 机[7]. 它的一个应用可以看本节的第二个例题.

例题

本节通过两道例题介绍如何实际应用 DFA 最小化的技巧.

例题

给定一个长度为 n 的 01? 串 a，初始变量 $x = 0$，我们按顺序遍历每一位 a_i 并执行如下操作：

1. 若 $a_i = 0$，令 $x \leftarrow x - \operatorname{lowbit}(x)$；
2. 若 $a_i = 1$，令 $x \leftarrow x + \operatorname{lowbit}(2^k - 1 - x)$；

3. 若 a_i =?，可任选 0 或 1，对应上述两种操作之一.

最终若 $x \in [0, r]$，则称该操作序列是好的.

现在需要对每个 $i = 1...n$，求在强制 $a_i = 0$ 的前提下，有多少「好的」完整序列. 特别地，$a_i = 1$ 时，答案为 0.

$1 \le n \le 10^5$，$1 \le k \le 20$，$0 \le r < 2^k$. 输出对 998244353 取模.

题解

考虑朴素 DP. 设 $f_{i,j}$ 表示从 $x = 0$ 开始, 经过 $[1, i]$ 的操作, 当前数为 j 的方案数. 设 $g_{i,j}$ 表示从 $x = j$ 开始, 经过 $[i, n]$ 的操作, 最终 $x \in [0, r]$ 的方案数. 强制 $a_i = 0$ 的答案, 就是 $\sum_j f_{i-1,j} g_{i+1,j-\text{ lowbit}(j)}$. 复杂度是 $O(n2^k)$.

考虑直接将 j 的转移建成 DFA，然后跑 DFA 最小化，再 DP 就可以了.

参考代码

```cpp
#include <iostream>
#include <string>
#include <vector>

int main() {
  constexpr int M = 998244353;
  // Initialize the DFA.
  const auto lowbit = [](int x) { return x & -x; };
  int n, k, r;
  std::cin >> n >> k >> r;
  DFA raw_dfa(2, 1 << k);
  for (int x = 0; x < (1 << k); ++x) {
    raw_dfa.trans[0][x] = x - lowbit(x);
    raw_dfa.trans[1][x] = x + lowbit((1 << k) - 1 - x);
    if (x <= r) raw_dfa.acc[x] = 1;
  }
  // DFA minimization.
  auto dfa = raw_dfa.hopcroft_minimize();
  // DP.
  std::string s;
  std::cin >> s;
  for (int i = 0; i < n; ++i) s[i] -= '0';
  std::vector<std::vector<int>> f(n + 1, std::vector<int>(dfa.n, 0));
  f[0][dfa.q0] = 1;
  for (int i = 0; i < n; ++i) {
    for (int c = 0; c < 2; ++c) {
      if (s[i] != 1 - c) {
        for (int j = 0; j < dfa.n; ++j) {
          (f[i + 1][dfa.trans[c][j]] += f[i][j]) %= M;
        }
      }
    }
  }
  std::vector<std::vector<int>> g(n + 1, std::vector<int>(dfa.n, 0));
```

```
  for (int j = 0; j < dfa.n; ++j) {
    if (dfa.acc[j]) g[n][j] = 1;
  }
  for (int i = n - 1; i >= 0; --i) {
    for (int c = 0; c < 2; ++c) {
      if (s[i] != 1 - c) {
        for (int j = 0; j < dfa.n; ++j) {
          (g[i][j] += g[i + 1][dfa.trans[c][j]]) %= M;
        }
      }
    }
  }
  for (int i = 0; i < n; ++i) {
    long long res = 0;
    if (s[i] != 1) {
      for (int j = 0; j < dfa.n; ++j) {
        (res += (long long)f[i][j] * g[i + 1][dfa.trans[0][j]]) %= M;
      }
    }
    std::cout << res << '\n';
  }
  return 0;
}

```

Minimal Subset Difference[8]

定义 $f(n)$ 表示将十进制数 n 所有数码之间填入加号或者减号，最终得到的值的绝对值最小值.

T 组询问. 每组询问给定 l, r, k，求满足 $l \le m \le r$ 且 $f(m) \le k$ 的 m 的个数.

$1 \le T \le 5 \times 10^4$，$1 \le l \le r \le 10^{18}$，$0 \le k \le 9$.

题解

先给出一种贪心地计算 $f(n)$ 的方法. 从高位向低位考虑一个数，最开始，设得到的数的和是 0. 计算到某一数位, 如果当前合成出的数小于等于 0 就加上当前数位, 否则就减去当前数位. 这么处理, 贪心计算出的结果的绝对值一定小于等于 9. 所以真实的 $f(n)$ 的绝对值一定小于等于 9.

我们进一步思考, 要合成出最终的答案, 中间过程中能够合成出来的数最大能是多少. 因为答案一定是小于等于 9 的，而且数位至多只有 19 位，每次最多只能加减 9. 过程中能够合成出来的数肯定是小于等于 90 的，否则最后减不回来. 实际上，这个上限还能够更低[6].

考虑朴素 DP 套 DP. 首先，思考内层 DP 怎么判定一个数的答案：定义 $g_{i,c}$ 表示这个数只根据前 i 位，能否合成出 c. 根据前文，c 只用保留小于等于 90 的数. 如果当前这一位填的是 v，那么，有转移:

$$g_{i+1,c+v} \leftarrow g_{i+1,c+v} \vee g_{i,c},\ g_{i+1,|\,c-v\,|} \leftarrow g_{i+1,|\,c-v\,|} \vee g_{i,c}.$$

外层 DP 考虑数位 DP. 将询问差分. 设状态为 $f_{len,lim,sta}$，它的下标分别表示已经考虑到第 len 位，是否有上界限制，当前自动机的状态位于 sta 等.

[6]详见 国家集训队 2021 论文 徐哲安 浅谈有限状态自动机及其应用[9] 中的例题 5.2.

与普通的 DFA 不同，我们需要对自动机的每个状态记录对应的答案．跑一次暴力搜索，会发现内层 DP 的状态数只有 19564．然后接下来直接跑 DFA 最小化，可以将状态数优化到 715.

此时我们将 $lim = 0$ 的数位 DP 答案都预处理出来，在多测时就只需要跑 $lim = 1$ 的情况，可以很快地求出答案.

时间复杂度 $O(|S||\Sigma|\log|S| + (|Q||\Sigma| + T)|\Sigma|\log_{10} V)$（$|S| = 19564$，$|Q| = 715$）.

参考代码

```cpp
#include <bitset>
#include <cmath>
#include <iostream>
#include <unordered_map>
#include <vector>

constexpr int B = 10, L = 91;

using state = std::bitset<L>;
DFA raw_dfa(B), dfa(B);
std::unordered_map<state, int> ids;

// Construct a DFA by DFS.
int dfs(const state& cr) {
  if (ids.count(cr)) return ids[cr];
  int id = ids[cr] = raw_dfa.n++;
  // Check if accepted.
  for (int i = 0; i < B; ++i) {
    if (cr[i]) {
      raw_dfa.acc.push_back(i);
      break;
    }
  }
  // Construct transitions recursively.
  for (int c = 0; c < B; ++c) {
    raw_dfa.trans[c].push_back(0);
  }
  for (int c = 0; c < B; ++c) {
    state nt;
    for (int i = 0; i < L; ++i) {
      if (cr[i]) {
        if (i + c < L) nt[i + c] = true;
        nt[std::abs(i - c)] = true;
      }
    }
    raw_dfa.trans[c][id] = dfs(nt);
  }
  return id;
};

// DP.
std::vector<long long> memo;
std::vector<int> nums;
```

```cpp

long long sol(int x, int len, bool lim, int k) {
  if (!len) return dfa.acc[x] <= k;
  auto key = (x * 20 + len) * B + k;
  if (!lim && memo[key] != -1) return memo[key];
  long long res = 0;
  for (int c = 0; c <= (lim ? nums[len - 1] : B - 1); ++c)
    res += sol(dfa.trans[c][x], len - 1, lim && c == nums[len - 1], k);
  return lim ? res : memo[key] = res;
};

long long calc(long long n, int k) {
  nums.clear();
  for (; n; n /= B) nums.push_back(n % B);
  return sol(dfa.q0, nums.size(), true, k);
};

int main() {
  // Construct a DFA.
  for (int c = 0; c < B; ++c) raw_dfa.trans[c].reserve(20000);
  dfs(1);
  // DFA minimization.
  dfa = raw_dfa.hopcroft_minimize();
  memo.assign(dfa.n * 20 * B, -1);
  // Queries.
  int t;
  std::cin >> t;
  for (; t; --t) {
    long long l, r;
    int k;
    std::cin >> l >> r >> k;
    std::cout << (calc(r, k) - calc(l - 1, k)) << '\n';
  }
  return 0;
}

```

习题

- Language Recognition[10]
- Equanimous[11]

自动机常见应用

本节列举了一些算法竞赛中常见的自动机的应用[7].

字典树

字典树→8.6@p903 是大部分 OIer 接触到的第一个自动机，接受且仅接受指定的字符串集合中的元素. 转移函数就是 Trie 上的边，接受状态是将每个字符串插入到 Trie 时到达的那个状态.

[7]本文对自动机的定义要求它是完备的, 即任一状态在任一字符下都必须有转移. 对这些字符串相关的自动机的描述中，通常会忽略失配状态. Trie、SAM 等都是这样的例子. 为了与本文提供的定义相适应，需要在这些自动机的描述中显式地添加失配状态.

KMP 自动机

KMP 算法→8.7@p917 可以视作自动机，基于字符串 s 的 KMP 自动机接受且仅接受以 s 为后缀的字符串，其接受状态为 $|s|$.

转移函数如下，其中 s 的下标从 1 开始，$\pi(i)$ 表示前缀 $s[1..i]$ 的最长公共真前后缀的长度（即 前缀函数→8.7@p917 中的 $\pi[i-1]$）：

$$\delta(i,c)=\begin{cases}i+1 & i<|s|\wedge s[i+1]=c\\ 0 & s[1]\neq c\wedge i=0\\ \delta(\pi(i),c) & (i=|s|\vee s[i+1]\neq c)\wedge i>0\end{cases}$$

AC 自动机

AC 自动机→8.10@p951 接受且仅接受以指定的字符串集合中的某个元素为后缀的字符串. 也就是 Trie + KMP.

后缀自动机

后缀自动机→8.12@p999 接受且仅接受指定字符串的后缀.

广义后缀自动机

广义后缀自动机→8.14@p1029 接受且仅接受指定的字符串集合中的某个元素的后缀. 也就是 Trie + SAM.

广义 SAM 与 SAM 的关系就是 AC 自动机与 KMP 自动机的关系.

回文自动机

回文自动机→8.17@p1053 比较特殊，它不能非常方便地定义为自动机.

如果需要定义的话，它接受且仅接受某个字符串的所有回文子串的 **中心及右半部分**.

「中心及右半部分」在奇回文串中就是字面意思，在偶回文串中定义为一个特殊字符加上右半部分. 这个定义看起来很奇怪，但它能让 PAM 真正成为一个自动机，而不仅是两棵树.

序列自动机

序列自动机→8.18@p1064 接受且仅接受指定字符串的子序列.

DP 套 DP

DP 套 DP→7.14@p801 是自动机的一个应用，可以看作是先通过内层 DP 建出自动机，再在外层通过自动机上的 DP 实现计数、最优化任务的技巧.

后缀链接

由于自动机和匹配有着密不可分的关系，而匹配的一个基本思想是「这个串不行，就试试它的后缀可不可以」，所以在很多自动机（KMP、AC 自动机、SAM、PAM）中，都有后缀链接的概念.

一个状态会对应若干字符串. 它的后缀链接，就指向自动机上该状态对应的字符串的公共真后缀中，最长的那个对应的状态. 一般地，后缀链接会形成一棵树，并且不同自动机的后缀链接树有着一些相同的性质，学习时可以加以注意.

拓展阅读

- 计算复杂性（1）Warming Up: 自动机模型[12]
- 国家集训队 2021 论文 徐哲安 浅谈有限状态自动机及其应用[13]
- Myhill–Nerode theorem - Wikipedia[14]

- Knuutila, Timo. "Re-describing an algorithm by Hopcroft." Theoretical Computer Science 250, no. 1-2 (2001): 333-363.
- Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. "Introduction to automata theory, languages, and computation." 2nd ed. Addison-Wesley, 2001.

[1] Thompson 构造法 [2] 国家集训队 2021 论文 徐哲安 浅谈有限状态自动机及其应用 [3] P12294 [THUPC 2025 决赛] 一个 01 串，n 次三目运算符，最后值为 1（加强版）

[4] 官方题解 [5] Median Replace Hard [6] JOISC 2024 卡牌收集

[7] Moore 机 [8] Minimal Subset Difference [9] 国家集训队 2021 论文 徐哲安 浅谈有限状态自动机及其应用

[10] Language Recognition [11] Equanimous [12] 计算复杂性（1）Warming Up: 自动机模型

[13] 国家集训队 2021 论文 徐哲安 浅谈有限状态自动机及其应用 [14] Myhill–Nerode theorem - Wikipedia

13.9 计算理论基础

本部分将介绍基础的计算理论的知识．这部分内容在 OI 中作用不大（但还是略有作用：如果你遇到了一个 NP-hard 问题，你可以认为它是不存在多项式复杂度的解法的），可以作为兴趣了解，或者为以后的学习做准备．

本文中许多结论都是不加证明的，如果有兴趣的话可以自行查阅相关证明．

前置知识：时间复杂度→5.2.1@p544．

问题

语言

一个 **字母表（alphabet）** 是一个非空有限集合，该集合中的元素称为 **符号/字符（symbol）**．

令 Σ^* 表示非负整数个 Σ 中的字符连接而成的串，字母表 Σ 上的一个 **语言（language）** 是 Σ^* 的一个子集．

需要注意的是，这里的「语言」是一个抽象的概念，通常意义上的字符串是语言，所有的有向无环图也可以是一个语言（01 串与有向图之间可以建立双射，具体方式无需了解）．

由于任何语言都可以转化成 01 串的形式，所以在下文中不加说明时 $\Sigma = \{0, 1\}$．

判定问题

判定问题就是只能用 YES/NO 回答的问题，本质上是判定一个串是否属于一个语言，即：$f : \Sigma^* \to \{0, 1\}, f(x) = 1 \iff x \in L$ 是一个关于字母表 Σ 和语言 L 的判定问题．如，「判定一张图是不是一个有向无环图」就是一个判定问题．

判定问题由于其简洁性而常常被作为计算理论研究的对象．本文中不加说明时，「问题」都指「判定问题」，当然，有时一些命题也能简单地推广到其它问题上．

一个语言也可以代指「判定一个串是否属于这个语言」这个判定问题，因此，「语言」和「问题」可以视作同义词．

功能性问题

功能性问题的回答不止 YES/NO，可以是一个数或是其它．如，「求两个数的和」就是一个功能性问题．

任何功能性问题都可以转化为一个判定问题，如，「求两个数的和」可以转化为「判定两个数的和是否等于第三个数」.

判定问题也可以转化为一个功能性问题: 求这个判定问题的指示函数, 即上文中判定问题定义里的 f.

图灵机

确定性图灵机

不加说明时，「图灵机」往往指「确定性图灵机」，本文中也是如此.

图灵机有很多不同的定义, 这里选取其中一种, 其它定义下的图灵机往往与下面这种定义的图灵机计算能力等价.

图灵机是一个在一条可双向无限延伸且被划分为若干格子的纸带上进行操作的机器，其有内部状态，还有一个可以在纸带上进行修改与移动的磁针.

正式地说，图灵机是一个七元组 $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F\rangle$，其中:

- Q 是一个有限非空的 **状态集合**;
- Γ 是一个有限非空的 **磁带字母表**;
- $b \in \Gamma$ 是 **空字符**，它是唯一一个在计算过程中可以在磁带上无限频繁地出现的字符;
- $\Sigma \subseteq (\Gamma \setminus \{b\})$ 是 **输入符号集**，是可以出现在初始磁带（即输入）上的字符;
- $q_0 \in Q$ 是 **初始状态**;
- $F \subseteq Q$ 是 **接受状态**，如果一个图灵机在某个接受状态停机，则称初始磁带上的内容被这个图灵机 **接受**.
- $\delta : (Q \setminus F) \times \Gamma \not\to Q \times \Gamma \times \{L, R\}$ 是一个被称作 **转移函数** 的 partial function（即只对定义域的一个子集有定义的函数）. 如果 δ 在当前状态下没有定义，则图灵机停机.

图灵机从初始状态与纸带起点起，每次根据当前的内部状态 x 和当前磁针指向的纸带上的单元格中的字符 y 进行操作：若 $\delta(x, y)$ 没有定义则停机，否则若 $\delta(x, y) = (a, b, c)$，则将内部状态修改为 a，将磁针指向的格子中的字符修改为 b，若 c 为 L 则向左移动一格，为 R 则向右移动一格.

其实，知道图灵机的工作细节是不必要的，只需建立直观理解即可.

图灵机 M 在输入 x 下的输出记作 $M(x)$（$M(x) = 1$ 当且仅当 M 接受 x，$M(x) = 0$ 当且仅当 M 在输入 x 下在有限步骤内停机且 M 不接受 x），也可以在括号内包含多个参数，用逗号隔开，具体实现时可以向字母表中添加一个元素表示逗号来隔开各个参数.

图灵机与冯·诺依曼计算机解决问题的时间复杂度差别在多项式级别内，所以研究复杂度类时可以使用图灵机作为计算模型.

非确定性图灵机

非确定型图灵机是图灵机的一种, 它与确定型图灵机的不同在于: 确定型图灵机的每一步只能转移到一个状态, 而非确定型图灵机可以「同时」转移到多个状态, 从而在多个「分支」并行计算，一旦这些「分支」中有一个在接受状态停机，则此非确定性图灵机接受这个输入.

事实上，任何确定型图灵机都可以用类似于迭代加深搜索的方式在指数级时间内模拟一台非确定型图灵机多项式时间内的行为.

在现实生活中, 确定型图灵机相当于单核处理器, 只支持串行处理; 而非确定型图灵机相当于理想的多核处理器，支持无限大小的并行处理.

多带图灵机

标准的图灵机只能在一条纸带上进行操作，但为了方便，本文中研究多带图灵机. 对于一个 k 带图灵机，其中一条纸带是只读的输入带，而剩下的 $k-1$ 条纸带可以进行读写，并且这 $k-1$ 条纸带中还有一条纸带用作输出.

多带图灵机的纸带数必须是有限的.

对于一个多带图灵机，它使用的空间是磁头在除输入带外的其它纸带上所访问过的单元格数目.

图灵机的编码

图灵机可以被自然数编码，即存在满射函数 $f:\mathbb{N}\to\mathbb{M}$，使得每个自然数都对应一个图灵机，而每个图灵机都有无数个编码. 因此，由若干图灵机构成的集合可以是一个语言.

记由自然数 α 编码的图灵机为 M_α.

通用图灵机

存在一台图灵机 $\mathcal{U}$ 满足:

1. 若 M_α 在输入 x 下在有限时间内停机，则 $\mathcal{U}(x,\alpha)=M_\alpha(x)$，否则 $\mathcal{U}(x,\alpha)$ 不会在有限时间内停机;
2. 如果对于任意 $x\in\{0,1\}^*$，M_α 在输入 x 下在 $T(|\ x\ |)$ 时间内停机，则对于任意 $x\in\{0,1\}^*$，$\mathcal{U}(x,\alpha)$ 在 $O(T(|\ x\ |)\log T(|\ x\ |))$ 时间内停机.

即: 存在一台通用图灵机，它能模拟任何一台图灵机，且花费的时间只会比这台被模拟的图灵机慢其运行时间的对数.

可计算性

不可计算问题

对于一个判定问题, 若存在一个总是在有限步内停机且能够正确进行判定的图灵机, 则这个问题是一个 **图灵可计算** 的问题，否则这个问题是一个 **图灵不可计算** 的问题.

由于图灵机可以被自然数编码，所以图灵机的个数是可数无穷，而语言（即二进制串的集合）的个数是不可数无穷，而每个图灵机最多判定一个语言，所以一定存在图灵不可计算的问题.

停机问题

停机问题是一个经典的图灵不可计算问题：给定 α 和 x，判定 M_α 在输入为 x 时是否会在有限步内停机.

停机问题是图灵不可计算的证明

定义函数 $UC:\{0,1\}^*\to\{0,1\}$ 为:

$$UC(\alpha)=\begin{cases}0 & M_\alpha(\alpha)=1\\ 1 & \text{otherwise}\end{cases}$$

我们先证明 UC 函数是图灵不可计算的:

假设存在一台图灵机 M_β 能够计算 UC，那么根据 UC 的定义可以得到 $UC(\beta)=1\iff M_\beta(\beta)\neq 1$，而根据 M_β 能够计算 UC 可以得到 $M_\beta(\beta)=UC(\beta)$，产生了矛盾，所以假设不成立，不存在可以计算 UC 的图灵机.

令 M_{HALT} 是一个可以解决停机问题的图灵机，$M_{HALT}(x,\alpha)$ 的值是判定问题 M_α 在输入为 x 时是否会在有限步内停机的解，那么我们可以构造出一台能够计算 UC 函数的图灵机 M_{UC}:

M_{UC} 首先调用 $M_{HALT}(\alpha,\alpha)$, 如果它输出 0, 则 $M_{UC}(\alpha)=1$；否则，M_{UC} 使用通用图灵机模拟计算得到答案.

由于 UC 函数是图灵不可计算的，所以 M_{HALT} 不存在，也就是说停机问题是图灵不可计算的.

丘奇 - 图灵论题

丘奇 - 图灵论题称，若一类问题有一个有效的方法解决，则这类问题可以被某个图灵机解决.

其中,「有效的方法」需要满足:

1. 包含有限条清晰的指令；
2. 当用其解决这类问题的其中一个时，这个方法需要在有限步骤内结束，且得到正确的答案.

这个论题没有被证明，但其是计算理论的一条基本公理.

复杂度类

复杂度类有很多，本文只会介绍其中较为常见的一小部分.

R 和 RE

对于语言 L 和图灵机 M，若 M 在任何输入下都能在有限步骤内停机，且 $M(x)=1 \iff x \in L$，则称 M 能够 **判定** L.

对于语言 L 和图灵机 M，若对于任何属于 L 的输入，M 都在有限步骤内停机，且 $M(x)=1 \iff x \in L$，则称 M 能够 **识别** L.

复杂度类 R 表示那些可以被某台图灵机判定的语言的集合，即所有图灵可计算的语言.

复杂度类 RE 表示那些可以被某台图灵机识别的语言的集合. RE 也被称作递归可枚举语言.

由定义可以得到 $R \subseteq RE$.

DTIME

如果存在一台确定性图灵机能够判定一个语言，且对于任何输入 x，这台图灵机可以在 $O(f(|\ x\ |))$ 的时间内停机，那么这个语言属于 $DTIME(f(n))$ 类.

P

复杂度类 P 表示可以由确定性图灵机在多项式时间内解决的判定问题，即：

$$P=\bigcup_{k\in\mathbb{N}} DTIME(n^k)$$

线性规划、计算最大公约数、求图的最大匹配的判定版本都是 P 类问题.

EXPTIME

复杂度类 $EXPTIME$ 表示可以由确定性图灵机在指数级时间内解决的判定问题，即：

$$EXPTIME=\bigcup_{k\in\mathbb{N}} DTIME(2^{n^k})$$

停机问题的弱化版——给定一个图灵机的编码以及一个正整数 k，判定这个图灵机是否在 k 步内停机，是一个 $EXPTIME$ 类的问题. 因为这个问题的解法需要 $O(k)$ 的时间，而数字 k 可以被编码为长度为 $O(\log k)$ 的二进制串.

NTIME

如果存在一台非确定性图灵机能够判定一个语言，且对于任何输入 x，这台图灵机可以在 $O(f(|\ x\ |))$ 的时间内停机，那么这个语言属于 $NTIME(f(n))$ 类.

NP

复杂度类 NP 表示可以由非确定性图灵机在多项式时间内解决的判定问题，即：

$$NP=\bigcup_{k\in\mathbb{N}} NTIME(n^k)$$

所有 P 类问题都是 NP 类问题. 更多 NP 类问题请参见下文中的 NPC 问题以及 NP-intermediate 问题.

NP-hard

如果所有 NP 类问题都可以在多项式时间内规约到问题 H，那么问题 H 是 NP-hard 的.

换句话说，如果可以在一单位的时间内解决 NP-hard 的问题 H，那么所有 NP 类问题都可以在多项式单位的时间内解决.

NP-complete

如果一个问题既是 NP 类问题又是 NP-hard 的，那么这个问题是 NP 完全 (NP-complete) 的，或者说这是一个 NPC 问题.

一些经典的 NPC 问题：旅行商问题的判定版本、最大独立集问题的判定版本、最小点覆盖问题的判定版本、最长路问题的判定版本、0-1 整数规划问题的判定版本、集合覆盖问题、图着色问题、背包问题、三维匹配问题、最大割问题的判定版本.

NPC 问题的功能性版本往往是 NP-hard 的，例如：「判定一张图中是否存在大小为 k 的团」既是一个 NP 类问题又是 NP-hard 的，从而它是一个 NPC 问题，而它的功能性版本「求一张图的最大团」不是 NPC 问题，但这个功能性版本依然是 NP-hard 的.

类似地，其它复杂度类也会有「XX-complete」，如所有 $EXPTIME$ 类的问题都能在多项式时间内规约到 EXPTIME-complete 的问题.

co-NP

一个问题是 $co-NP$ 类问题，当且仅当它的补集是 NP 类问题. 如果将「问题」理解为「语言」，而「语言」是 Σ^* 的子集，就能理解「补集」了.

例如：「给定 n 个子集，判断是否能够从中选取 k 个，覆盖整个集合」是一个 NPC 问题，而其补集「给定 n 个子集，判断是否从中任取 k 个都不能覆盖整个集合」是一个 $co-NP$ 类问题. 如果第一个问题的答案是「是」，那么相当于找到了第二个问题的一组反例，从而第二个问题的答案是「否」.

NP-intermediate

如果一个问题是 NP 类问题，但它既不是 P 类问题也不是 NPC 问题，则称其为 NP-intermediate 问题.

就人们目前的了解，图同构问题、离散对数问题和因数分解问题可能是 NP-intermediate 的.

Ladner 定理指出，如果 $P \neq NP$，则一定存在问题是 NP-intermediate 的.

NEXPTIME

复杂度类 $NEXPTIME$ 表示可以由非确定性图灵机在指数级时间内解决的判定问题，即：

$$NEXPTIME = \bigcup_{k\in\mathbb{N}} NTIME(2^{n^k})$$

#P

$\#P$ 类问题不是判定问题，而是关于 NP 类问题的计数问题：数一个 NP 类问题的解的个数是一个 $\#P$ 类的问题. 换句话说，数一个串在一个总是在多项式时间内停机的非确定性图灵机的多少个分支处被接受是一个 $\#P$ 类的问题.

求一张普通图或二分图的匹配或完美匹配个数都是 #P 完全的，对应的判定问题为「判定一张图是否存在（完美）匹配」.

DSPACE

如果存在一台确定性图灵机能够在输入为 x 时在 $O(f(|\ x\ |))$ 的空间内判定一个语言，那么这个语言属于 $DSPACE(f(n))$ 类.

- $REG = DSPACE(O(1))$，即正则语言，也就是自动机能够判定的语言.
- $L = DSPACE(O(\log n))$，需要注意的是图灵机使用的空间不包括输入占用的空间.
- $PSPACE = \bigcup_{k\in\mathbb{N}} DSPACE(n^k)$
- $EXPSPACE = \bigcup_{k\in\mathbb{N}} DSPACE(2^{n^k})$

NSPACE

如果存在一台非确定性图灵机能够在输入为 x 时在 $O(f(|\ x\ |))$ 的空间内判定一个语言，那么这个语言属于 $NSPACE(f(n))$ 类.

- $REG = DSPACE(O(1)) = NSPACE(O(1))$
- $NL = NSPACE(O(\log n))$
- $CSL = NSPACE(O(n))$，即上下文相关语言.
- $PSPACE = NPSPACE = \bigcup\limits_{k\in\mathbb{N}} NSPACE(n^k)$
- $EXPSPACE = NEXPSPACE = \bigcup\limits_{k\in\mathbb{N}} NSPACE(2^{n^k})$

多项式时间

简单来说，如果存在正数 k 使得一个算法的时间复杂度为 $O(n^k)$（注意，不是 $\Theta(n^k)$），其中 n 为问题规模（输入的长度），则称这个算法是 **多项式时间** 的. 如果一个问题有（确定性图灵机上的）多项式时间的算法来解决，则这个问题属于复杂度类 P.

多项式时间可分为强多项式时间和弱多项式时间，除此之外还有伪多项式时间.

Strongly polynomial time 强多项式时间

我们先定义一个计算模型，称作算术模型. 在算术模型中，数字之间的算术运算（加减乘除、比较大小）可以在单位时间内完成（即 $O(1)$ 时间内完成，与数字大小无关）.

如果一个算法在算术模型下的操作数是输入中的数字个数的多项式，并且空间复杂度是输入规模（而非数字个数）的多项式，则这个算法是 **强多项式时间** 的. 由于算术操作在一般的计算模型下可以在输入规模（即数字大小的对数）的多项式时间内完成，强多项式时间的算法一定是多项式时间的.

一般来说，强多项式时间的算法的时间复杂度与值域无关.

Weakly polynomial time 弱多项式时间

如果一个算法是多项式时间的但不是强多项式时间的，则它是 **弱多项式时间** 的.

例如，计算最大公约数的欧几里得算法，时间复杂度为 $O(\log a + \log b)$（a 和 b 为输入的数的大小），是弱多项式时间的.

Pseudo-polynomial time 伪多项式时间

如果一个算法的用时是值域的多项式，则称它是 **伪多项式时间** 的. 伪多项式时间的算法可能是多项式时间的也可能不是，可能不是多项式时间是因为表示一个大小为 n 的正整数一般只需要 $O(\log n)$ 个二进制位，所以关于值域多项式时间的算法往往关于输入长度是指数级时间的. 虽然从定义上来说伪多项式时间也可能是多项式时间，但当我们说一个算法是伪多项式时间的，一般都是说这个算法不是多项式时间的.

例如，背包问题是 NP-hard 问题，但它有基于动态规划的伪多项式时间的解法.

如果一个 NPC/NP-hard 问题有伪多项式时间的解法，则称这个问题是 **弱 NPC/弱 NP-hard** 问题. 如果一个 NPC/NP-hard 问题在 $P \neq NP$ 的前提下没有伪多项式时间的解法，则称这个问题是 **强 NPC/强 NP-hard** 问题.

可构造函数

时间可构造函数

有时，我们想让图灵机知道自己用了多长的时间，例如，强制图灵机在进行 $T(n)$ 步计算后停机. 但如果计算 $T(n)$ 的用时就超过了 $T(n)$，这便是不可做到的. 为此，定义了时间可构造函数，来避免这样的麻烦.

如果存在图灵机 M，使得输入为 1^n(n 个 1) 时 M 能在 $O(f(n))$ 的时间内停机并且输出 $f(n)$ 的二进制表示（注意，这里的图灵机的输出不是接受/不接受，而是一个串，输出可以在纸带上进行），则 $f(n)$ 是一个 **时间可构造函数**.

由于读入需要 $O(n)$ 的时间，$o(n)$ 的非常值函数都不是时间可构造函数.

空间可构造函数

类似地可以定义空间可构造函数.

如果存在图灵机 M，使得输入为 1^n(n 个 1) 时 M 能在 $O(f(n))$ 的空间内停机并且输出 $f(n)$ 的二进制表示，则 $f(n)$ 是一个 **空间可构造函数**.

复杂度类之间的关系

时间谱系定理

确定性时间谱系定理

若 $f(n)$ 是一个时间可构造函数，则:

$$DTIME\left(o\left(\frac{f(n)}{\log f(n)}\right)\right) \subsetneq DTIME(f(n))$$

由确定性时间谱系定理可以得到 $P \subsetneq EXPTIME$.

确定性时间谱系定理的证明

定义语言 $L = \{(x,y)|\ \mathcal{U}((x,y),x)$在$f(|\ x\ |+|\ y\ |)$ 时间内停机并拒绝$\}$, 由于 $f(n)$ 是一个时间可构造函数，可以根据定义进行计算来判定 L，用时为 $O(f(|\ x\ |+|\ y\ |))$，所以 $L \in DTIME(f(n))$.

现在假设 $L \in DTIME(o\left(\frac{f(n)}{\log f(n)}\right))$，设 M_z 就是那台在 $o\left(\frac{f(n)}{\log f(n)}\right)$ 的时间内判定 L 的图灵机.

令通用图灵机 $\mathcal{U}(x,z)$ 关于 x 的用时为 $g(|\ x\ |)$，由上文关于通用图灵机的介绍可以得到 $g(n) = o(f(n))$，所以，当 y 足够大时，$g(|\ z\ |+|\ y\ |) < f(|\ z\ |+|\ y\ |)$.

令 y' 是一个足够大的 y, 那么 $\mathcal{U}((z,y'),z)$ 一定能在 $f(|\ z\ |+|\ y'|)$ 时间内停机, 从而 $M_z(z,y') \neq M_z(z,y')$，产生矛盾，所以假设不成立，确定性时间谱系定理证毕.

非确定性时间谱系定理

若 $g(n)$ 是一个时间可构造函数，并且 $f(n+1) = o(g(n))$，则 $NTIME(f(n)) \subsetneq NTIME(g(n))$.

由非确定性时间谱系定理可以得到 $NP \subsetneq NEXPTIME$.

空间谱系定理

若 $f(n)$ 是一个空间可构造函数且 $f(n) = \Omega(\log n)$，则 $SPACE(o(f(n))) \subsetneq SPACE(f(n))$.

其中 $SPACE$ 可以代指 $DSPACE$ 或 $NSPACE$.

由空间谱系定理可以得到 $PSPACE \subsetneq EXPSPACE$.

萨维奇定理

一台确定性图灵机可以在一台非确定性图灵机所消耗空间的平方内模拟它（尽管消耗的时间可能多很多），即:

若 $f(n) = \Omega(\log n)$，则:

$$NSPACE(f(n)) \subseteq DSPACE((f(n))^2)$$

推论：$PSPACE = NPSPACE$，$EXPSPACE = NEXPSPACE$.

P?=NP

复杂度类 P 与 NP 是否相等是计算复杂度理论中一个著名的尚未解决的问题.

若 $P = NP$，可以得到 $NP = co - NP$，但反之不行（目前没有基于 $NP = co - NP$ 证明 $P = NP$ 的方法）.

> **为什么 NP?=co-NP 不是显然的?**
>
> 由于 NP 问题和与其对应的 $co - NP$ 问题答案相反，很容易有这种想法：对于一个 $co - NP$ 问题，我只要将解决其补集的非确定性图灵机的输出反过来，就解决了该 $co - NP$ 问题，所以 $NP = co - NP$.
>
> 实际上，上面所说的这种方法确实能够解决该 $co - NP$ 问题，但并没有找到一个非确定性图灵机来解决它: 如果一个图灵机所做的事情是将一个非确定性图灵机的输出反过来, 该图灵机并不是一个非确定性图灵机. 因为，非确定性图灵机接受是在某个分支处接受，而拒绝是在所有分支处拒绝; 而将其输出反过来, 就变成了接受是在所有分支处，而拒绝是在一个分支处，而这样就不符合非确定性图灵机的定义了，所以能用该图灵机解决这个 $co - NP$ 问题并不能使这个 $co - NP$ 问题变成一个 NP 问题.

若 $P = NP$，还可以得到 $EXPTIME = NEXPTIME$.

若 $P \neq NP$，可以得到 NP-intermediate 不为空.

参考资料

1. 计算复杂性（1）自动机与正则语言[1];
2. 计算复杂性（2）图灵机与可计算性[2];
3. Wikipedia[3] 的相关词条以及这些词条的参考资料.

外部参考资料

[1] 计算复杂性（1）自动机与正则语言 

[2] 计算复杂性（2）图灵机与可计算性 

[3] Wikipedia 

13.10 字节顺序

本页面将简要介绍字节顺序的概念和分类.

简介

字节顺序是跨越多字节的程序对象的存储规则，表示一个对象的字节的排列方法.

分类

字节顺序有两种，分为小端序（little endian）和大端序（big endian）.

为方便介绍，接下来以一个位于 `0x100` 处，类型为 `int`，十六进制值为 `0x01234567` 的变量为例. 其中 `0x01` 是最高位有效字节，`0x67` 是最低位有效字节.

小端序

小端序是指机器选择在内存中按照从 **最低** 有效字节到 **最高** 有效字节的顺序存储对象.

上文提到的变量表示如下:

....	0x100	0x101	0x102	0x103	....
....	67	45	23	01	....

大端序

大端序是指机器选择在内存中按照从 **最高** 有效字节到 **最低** 有效字节的顺序存储对象.

上文提到的变量表示如下:

....	0x100	0x101	0x102	0x103	....
....	01	23	45	67	....

两种顺序的区别

事实上，这两种字节顺序没有孰优孰劣之分．这两种顺序的名字「小端」和「大端」，正是出自《格列佛游记》一书．书中，小人国里两个派别交战不休的原因是无法就从小端还是大端剥鸡蛋达成一致．就和剥鸡蛋的争论一样，选择何种字节顺序的争论是非技术性的.

当然, 字节顺序的不一致会导致二进制数据在不同类型的机器之间进行传输时被反序. 为了避免这件事情, 网络应用程序建立了一套标准, 保证发送过程中是使用约定好的网络标准, 而不是不同机器的内部表示.

顺序选择惯例

- 小端序：x86, ARM processors running Android, iOS, and Windows
- 大端序：Sun, PPC Mac, Internet

13.11 约瑟夫问题

约瑟夫问题由来已久, 而这个问题的解法也在不断改进, 只是目前仍没有一个极其高效的算法（log 以内）解决这个问题.

问题描述

> n 个人标号 $0, 1, \cdots, n-1$．逆时针站一圈，从 0 号开始，每一次从当前的人逆时针数 k 个，然后让这个人出局．问最后剩下的人是谁.

这个经典的问题由约瑟夫于公元 1 世纪提出，尽管他当时只考虑了 $k = 2$ 的情况．现在我们可以用许多高效的算法解决这个问题.

过程

朴素算法

最朴素的算法莫过于直接枚举．用一个环形链表枚举删除的过程，重复 $n-1$ 次得到答案．复杂度 $\Theta(n^2)$.

简单优化

寻找下一个人的过程可以用线段树优化．具体地，开一个 $0, 1, \cdots, n-1$ 的线段树，然后记录区间内剩下的人的个数．寻找当前的人的位置以及之后的第 k 个人可以在线段树上二分做.

线性算法

设 $J_{n,k}$ 表示规模分别为 n, k 的约瑟夫问题的答案．我们有如下递归式

$$J_{n,k} = (J_{n-1,k} + k) \bmod n$$

这个也很好推．你从 0 开始数 k 个，让第 $k-1$ 个人出局后剩下 $n-1$ 个人，你计算出在 $n-1$ 个人中选的答案后，再加一个相对位移 k 得到真正的答案．这个算法的复杂度显然是 $\Theta(n)$ 的.

实现

```
int josephus(int n, int k) {
  int res = 0;
  for (int i = 1; i <= n; ++i) res = (res + k) % i;
  return res;
}
```

对数算法

对于 k 较小 n 较大的情况，本题还有一种复杂度为 $\Theta(k \log n)$ 的算法.

考虑到我们每次走 k 个删一个，那么在一圈以内我们可以删掉 $\lfloor \frac{n}{k} \rfloor$ 个，然后剩下了 $n - \lfloor \frac{n}{k} \rfloor$ 个人. 这时我们在第 $\lfloor \frac{n}{k} \rfloor \cdot k$ 个人的位置上. 而你发现它等于 $n - n \bmod k$. 于是我们继续递归处理，算完后还原它的相对位置. 还原相对位置的依据是：每次做一次删除都会把数到的第 k 个人删除，他们的编号被之后的人逐个继承，也即用 $n - \lfloor \frac{n}{k} \rfloor$ 人环算时每 k 个人即有 1 个人的位置失算，因此在得数小于 0 时，用还没有被删去 k 倍数编号的 n 人环的 的 n 求模，在得数大于等于 0 时，即可以直接乘 $\frac{k}{k-1}$，于是得到如下的算法：

实现

```
int josephus(int n, int k) {
  if (n == 1) return 0;
  if (k == 1) return n - 1;
  if (k > n) return (josephus(n - 1, k) + k) % n;  // 线性算法
  int res = josephus(n - n / k, k);
  res -= n % k;
  if (res < 0)
    res += n;  // mod n
  else
    res += res / (k - 1);  // 还原位置
  return res;
}
```

可以证明这个算法的复杂度是 $\Theta(k \log n)$ 的. 我们设这个过程的递归次数是 x，那么每一次问题规模会大致变成 $n\left(1 - \frac{1}{k}\right)$，于是得到

$$n\left(1 - \frac{1}{k}\right)^x = 1$$

解这个方程得到

$$x = -\frac{\ln n}{\ln\left(1 - \frac{1}{k}\right)}$$

下面我们证明该算法的复杂度是 $\Theta(k \log n)$ 的.

证明

考虑 $\lim\limits_{k \to \infty} k \log\left(1 - \frac{1}{k}\right)$，我们有

$$\begin{aligned}\lim_{k\to\infty} k\log\left(1-\frac{1}{k}\right) &= \lim_{k\to\infty}\frac{\log\left(1-\frac{1}{k}\right)}{1/k}\\ &= \lim_{k\to\infty}\frac{\frac{\mathrm{d}}{\mathrm{d}k}\log\left(1-\frac{1}{k}\right)}{\frac{\mathrm{d}}{\mathrm{d}k}\left(\frac{1}{k}\right)}\\ &= \lim_{k\to\infty}\frac{\frac{1}{k^2\left(1-\frac{1}{k}\right)}}{-\frac{1}{k^2}}\\ &= \lim_{k\to\infty}-\frac{k}{k-1}\\ &= -\lim_{k\to\infty}\frac{1}{1-\frac{1}{k}}\\ &= -1\end{aligned}$$

所以 $x\sim k\ln n, k\to\infty$，即 $-\dfrac{\ln n}{\ln\left(1-\frac{1}{k}\right)}=\Theta(k\log n)$

本页面主要译自博文 Задача Иосифа[1] 与其英文翻译版 Josephus Problem[2]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

外部参考资料

[1] Задача Иосифа

[2] Josephus Problem

13.12 表达式求值

Authors: Ir1d, Anguei, hsfzLZH1, siger-young, HeRaNO, c8ef

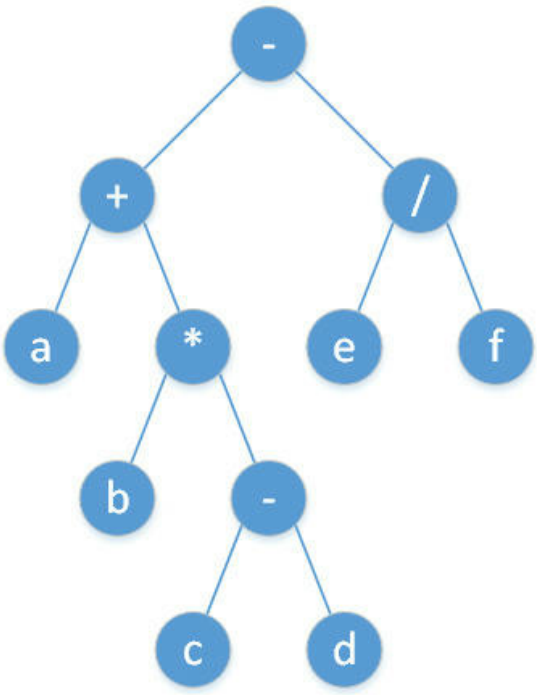
表达式求值要解决的问题一般是输入一个字符串表示的表达式，要求输出它的值．当然也有变种比如表达式中是否包含括号，指数运算，含多少变量，判断多个表达式是否等价，等等．

表达式一般需要先进行语法分析（grammer parsing）再求值，也可以边分析边求值，语法分析的作用是检查输入的字符串是否是一个合法的表达式，一般使用语法分析器（parser）解决．

表达式包含两类字符：运算数和运算符．对于长度为 n 的表达式，借助合适的分析方法，可以在 $O(n)$ 的时间复杂度内完成分析与求值．

表达式树与逆波兰表达式

一种递归分析表达式的方法是，将表达式当成普通的语法规则进行分析，分析后拆分成如图所示的表达式树，然后在树结构上自底向上进行运算．



表达式树上进行 树的遍历→11.6.1@p2621 可以得到不同类型的表达式．算术表达式分为三种，分别是前缀表达式、中缀表达式、后缀表达式．中缀表达式是日常生活中最常用的表达式；后缀表达式是计算机容易理解的表达式．

- 前序遍历对应前缀表达式（波兰式）
- 中序遍历对应中缀表达式
- 后序遍历对应后缀表达式（逆波兰式）

逆波兰表达式（后缀表达式）是书写数学表达式的一种形式，其中运算符位于其操作数之后．例如，以下表达式：

$$a + b*c*d + (e - f)*(g*h + i)$$

可以用逆波兰表达式书写：

$$abc*d* + ef - gh*i + * +$$

因此，逆波兰表达式与表达式树一一对应．逆波兰表达式不需要括号表示，它的运算顺序是唯一确定的．

逆波兰表达式的方便之处在于很容易在线性时间内计算．举个例子：在逆波兰表达式 3 2 * 1 − 中，首先计算 $3 \times 2 = 6$（使用最后一个运算符，即栈顶运算符），然后计算 $6 - 1 = 5$．可以看到：对于一个逆波兰表达式，只需要 **维护一个数字栈，每次遇到一个运算符，就取出两个栈顶元素，将运算结果重新压入栈中**．最后，栈中唯一一个元素就是该逆波兰表达式的运算结果．该算法拥有 $O(n)$ 的时间复杂度．

采用递归的办法分析表达式是否成功，依赖于语法规则的设计是否合理，即，是否能够成功地得到指定的表达式树．例如：

$$a + b*c$$

根据加号与乘号的运算优先级不同，该中缀表达式可能转化为两种不同的表达式树．可见，语法规则的设计高度依赖于运算符的优先级．借助运算符的优先级设计相应递归的语法规则，事实上是一件不容易的事情．

下文介绍的办法将运算符与它的优先级视为一个整体，采用非递归的办法，直接根据运算符的优先级来分析与计算表达式．

只含左结合的二元运算符的含括号表达式

考虑简化的问题. 假设所有运算符都是二元的: 所有运算符都有两个参数. 并且所有运算符都是左结合的: 如果运算符的优先级相等, 则从左到右执行. 允许使用括号.

对于这种类型的中缀表达式的计算, 可以将其转化为后缀表达式再进行计算. 定义两个 栈→10.2@p1978 来分别存储运算符和运算数, 每当遇到一个数直接放进运算数栈. 每个运算符块对应于一对括号, 运算符栈只对于运算符块的内部单调. 每当遇到一个操作符时, 要查找运算符栈中最顶部运算符块中的元素, 在运算符块的内部保持运算符按照优先级降序进行适当的弹出操作, 弹出的同时求出对应的子表达式的值.

以下部分用「输出」表示输出到后缀表达式, 即将该数字放在运算数栈上, 或者弹出运算符和两个操作数, 运算后再将结果压回运算数栈上. 从左到右扫描该中缀表达式:

1. 如果遇到数字, 直接输出该数字.
2. 如果遇到左括号, 那么将其放在运算符栈上.
3. 如果遇到右括号, 不断输出栈顶元素, 直至遇到左括号, 左括号出栈. 换句话说, 执行一对括号内的所有运算符.
4. 如果遇到其他运算符, 不断输出所有运算优先级大于等于当前运算符的运算符. 最后, 新的运算符入运算符栈.
5. 在处理完整个字符串之后, 一些运算符可能仍然在堆栈中, 因此把栈中剩下的符号依次输出, 表达式转换结束.

以下是四个运算符 +、−、*、/ 的此方法的实现:

示例代码

```
1
2  bool delim(char c) { return c == ' '; }
3
4  bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }
5
6  int priority(char op) {
7    if (op == '+' || op == '-') return 1;
8    if (op == '*' || op == '/') return 2;
9    return -1;
10 }
11
12 void process_op(stack<int>& st, char op) {  // 也可以用于计算后缀表达式
13   int r = st.top();                          // 取出栈顶元素，注意顺序
14   st.pop();
15   int l = st.top();
16   st.pop();
17   switch (op) {
18     case '+':
19       st.push(l + r);
20       break;
21     case '-':
22       st.push(l - r);
23       break;
24     case '*':
25       st.push(l * r);
26       break;
27     case '/':
28       st.push(l / r);
```

```
29        break;
30    }
31  }
32
33  int evaluate(string& s) {  // 也可以改造为中缀表达式转换后缀表达式
34    stack<int> st;
35    stack<char> op;
36    for (int i = 0; i < (int)s.size(); i++) {
37      if (delim(s[i])) continue;
38
39      if (s[i] == '(') {
40        op.push('(');  // 2. 如果遇到左括号，那么将其放在运算符栈上
41      } else if (s[i] == ')') {  // 3. 如果遇到右括号，执行一对括号内的所有运算符
42        while (op.top() != '(') {
43          process_op(st, op.top());
44          op.pop();  // 不断输出栈顶元素，直至遇到左括号
45        }
46        op.pop();                 // 左括号出栈
47      } else if (is_op(s[i])) {  // 4. 如果遇到其他运算符
48        char cur_op = s[i];
49        while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
50          process_op(st, op.top());
51          op.pop();  // 不断输出所有运算优先级大于等于当前运算符的运算符
52        }
53        op.push(cur_op);  // 新的运算符入运算符栈
54      } else {            // 1. 如果遇到数字，直接输出该数字
55        int number = 0;
56        while (i < (int)s.size() && isalnum(s[i]))
57          number = number * 10 + s[i++] - '0';
58        --i;
59        st.push(number);
60      }
61    }
62
63    while (!op.empty()) {
64      process_op(st, op.top());
65      op.pop();
66    }
67    return st.top();
68  }
69
```

这种隐式使用逆波兰表达式计算表达式的值的算法的时间复杂度为$O(n)$. 通过稍微修改上述实现, 还可以以显式形式获得逆波兰表达式.

一元运算符与右结合的运算符

现在假设表达式还包含一元运算符, 即只有一个参数的运算符. 一元加号和一元减号是一元运算符的常见示例.

这种情况的一个区别是, 需要确定当前运算符是一元运算符还是二元运算符.

注意到, 在一元运算符之前一般有另一个运算符或开括号, 如果一元运算符位于表达式的最开头则没有. 在二元运算符之前, 总是有一个运算数或右括号. 因此, 可以标记下一个运算符是否一元运算符.

此外，需要以不同的方式执行一元运算符和二元运算符，让一元运算符的优先级高于所有二元运算符．应注意，一些一元运算符，例如一元加号和一元减号，实际上是右结合的．

右结合意味着，每当优先级相等时，必须从右到左计算运算符．

如上所述，一元运算符通常是右结合的．右结合运算符的另一个示例是求幂运算符．对于 $a \wedge b \wedge c$，通常被视为 a^{b^c}，而不是 $(a^b)^c$．

为了正确地处理这类运算符，相应的改动是，如果优先级相等，将推迟运算符的出栈操作．

需要改动的代码如下．将：

```
1
2 while (!op.empty() && priority(op.top()) >= priority(cur_op))
```

换成

```
1
2 while (!op.empty() &&
3        ((left_assoc(cur_op) && priority(op.top()) >= priority(cur_op)) ||
4         (!left_assoc(cur_op) && priority(op.top()) > priority(cur_op))))
5
```

其中 left_assoc 是一个函数，它决定运算符是否为左结合的．

这里是二进制运算符 +、−、*、/ 和一元运算符 + 和 − 的实现：

示例代码

```
1
2 bool delim(char c) { return c == ' '; }
3
4 bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }
5
6 bool is_unary(char c) { return c == '+' || c == '-'; }
7
8 int priority(char op) {
9   if (op < 0)  // unary operator
10    return 3;
11  if (op == '+' || op == '-') return 1;
12  if (op == '*' || op == '/') return 2;
13  return -1;
14 }
15
16 void process_op(stack<int>& st, char op) {
17   if (op < 0) {
18     int l = st.top();
19     st.pop();
20     switch (-op) {
21       case '+':
22         st.push(l);
23         break;
24       case '-':
25         st.push(-l);
26         break;
27     }
28   } else {  // 取出栈顶元素，注意顺序
```

```
    int r = st.top();
    st.pop();
    int l = st.top();
    st.pop();
    switch (op) {
      case '+':
        st.push(l + r);
        break;
      case '-':
        st.push(l - r);
        break;
      case '*':
        st.push(l * r);
        break;
      case '/':
        st.push(l / r);
        break;
    }
  }
}

int evaluate(string& s) {
  stack<int> st;
  stack<char> op;
  bool may_be_unary = true;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim(s[i])) continue;

    if (s[i] == '(') {
      op.push('(');  // 2. 如果遇到左括号，那么将其放在运算符栈上
      may_be_unary = true;
    } else if (s[i] == ')') {  // 3. 如果遇到右括号，执行一对括号内的所有运算符
      while (op.top() != '(') {
        process_op(st, op.top());
        op.pop();  // 不断输出栈顶元素，直至遇到左括号
      }
      op.pop();  // 左括号出栈
      may_be_unary = false;
    } else if (is_op(s[i])) {  // 4. 如果遇到其他运算符
      char cur_op = s[i];
      if (may_be_unary && is_unary(cur_op)) cur_op = -cur_op;
      while (!op.empty() &&
             ((cur_op >= 0 && priority(op.top()) >= priority(cur_op)) ||
              (cur_op < 0 && priority(op.top()) > priority(cur_op)))) {
        process_op(st, op.top());
        op.pop();  // 不断输出所有运算优先级大于等于当前运算符的运算符
      }
      op.push(cur_op);  // 新的运算符入运算符栈
      may_be_unary = true;
    } else {  // 1. 如果遇到数字，直接输出该数字
      int number = 0;
      while (i < (int)s.size() && isalnum(s[i]))
```

```
        number = number * 10 + s[i++] - '0';
      --i;
      st.push(number);
      may_be_unary = false;
    }
  }

  while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
  }
  return st.top();
}

```

参考资料

本页面主要译自博文 Разбор выражений. Обратная польская нотация[1] 与其英文翻译版 Expression parsing[2]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

延申阅读

1. Operator-precedence_parser[3]
2. Shunting yard algorithm[4]

习题

1. NOIP2013 普及组 表达式求值[5]
2. 后缀表达式[6]
3. Transform the Expression[7]

外部参考资料

[1] Разбор выражений. Обратная польская нотация

[2] Expression parsing

[3] Operator-precedence_parser

[4] Shunting yard algorithm

[5] NOIP2013 普及组 表达式求值

[6] 后缀表达式

[7] Transform the Expression

13.13 在一台机器上规划任务

你有 n 个任务，要求你找到一个代价最小的顺序执行他们．第 i 个任务花费的时间是 t_i，而第 i 个任务等待 t 的时间会花费 $f_i(t)$ 的代价．

形式化地说，给出 n 个函数 f_i 和 n 个数 t_i，求一个排列 p，最小化

$$F(p) = \sum_{i=1}^{n} f_{p_i}\left(\sum_{j=1}^{i-1} t_{p_j}\right)$$

特殊的代价函数

线性代价函数

首先我们考虑所有的函数是线性的函数，即 $f_i(x) = c_i x + d_i$，其中 c_i 是非负整数．显然我们可以事先把常数项加起来，因此函数就转化为了 $f_i(x) = c_i x$ 的形式．

考虑两个排列 p 和 p'，其中 p' 是把 p 的第 i 个位置上的数和 $i+1$ 个位置上的数交换得到的排列．则

$$\begin{aligned} F(p') - F(p) &= c_{p'_i} \sum_{j=1}^{i-1} t_{p'_j} + c_{p'_{i+1}} \sum_{j=1}^{i} t_{p'_j} - \left(c_{p_i} \sum_{j=1}^{i-1} t_{p_j} + c_{p_{i+1}} \sum_{j=1}^{i} t_{p_j} \right) \\ &= c_{p_i} t_{p_{i+1}} - c_{p_{i+1}} t_{p_i} \end{aligned}$$

于是我们使用如果 $c_{p_i} t_{p_{i+1}} - c_{p_{i+1}} t_{p_i} < 0$ 就交换的策略做一下排序就可以了．写成 $\dfrac{c_{p_i}}{t_{p_i}} < \dfrac{c_{p_{i+1}}}{t_{p_{i+1}}}$ 的形式，就可以理解为将排列按 $\dfrac{c_i}{t_i}$ 降序排序．

处理这个问题，我们的思路是考虑微扰后的变换情况，贪心地选取最优解．

指数代价函数

考虑代价函数的形式为 $f_i(x) = c_i \mathrm{e}^{ax}$，其中 $c_i \geq 0, a > 0$．

我们沿用之前的思路，考虑将 i 和 $i+1$ 的位置上的数交换引起的代价变化．最终得到的算法是将排列按照 $\dfrac{1 - \mathrm{e}^{at_i}}{c_i}$ 升序排序．

相同的单增函数

我们考虑所有的 $f_i(x)$ 是同一个单增函数．那么显然我们将排列按照 t_i 升序排序即可．

Livshits–Kladov 定理

Livshits–Kladov 定理成立，当且仅当代价函数是以下三种情况：

- 线性函数：$f_i(t) = c_i t + d_i$，其中 $c_i \geq 0$；
- 指数函数：$f_i(t) = c_i \mathrm{e}^{at} + d_i$，其中 $c_i, a > 0$；
- 相同的单增函数：$f_i(t) = \phi(t)$，其中 $\phi(t)$ 是一个单增函数．

定理是在假设代价函数足够平滑（存在三阶导数）的条件下证明的．在这三种情况下，问题的最优解可以通过简单的排序在 $O(n \log n)$ 的时间内解决．

本页面主要译自博文 Задача Джонсона с одним станком[1] 与其英文翻译版 Scheduling jobs on one machine[2]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

外部参考资料

[1] Задача Джонсона с одним станком 

[2] Scheduling jobs on one machine 

13.14 主元素问题

Authors: SDLTF, Ethkuil

问题介绍

给一个有 n 个元素的序列，保证有一个元素 a 出现的次数 **严格大于** $n/2$，求这个元素．

做法

离线算法

若一开始就可以知道整个序列，一个自然的思路是统计序列中各元素的出现次数，出现次数大于 $n/2$ 的就是主元素．可以创建一个桶来统计每种元素的出现次数，输出出现次数大于 $n/2$ 的元素即可．

但上述方案引入了桶进行统计，空间效率并不优．显然，若序列存在主元素，那么在排序后，序列的第 $\lfloor n/2 \rfloor + 1$ 个元素一定是主元素，利用 `nth_element`[1] 就可以找到这个元素．这样我们就在不引入额外空间的情况下，以线性时间复杂度求得主元素了．

在线算法

在一些情况下，我们需要在线处理流式数据，此时我们需要一种不需要预知全体数据，而是只利用当前给出的数据逐步求得答案的算法．**多数投票算法**[1]就是一种可以在线解决主元素问题的算法．

由于主元素的出现的次数超过 $n/2$，那么对于一个完整的序列，在不断消掉一个主元素和一个与主元素不同的元素之后，最后一定剩下主元素．借助这个观察，我们可以设计一个在线算法进行这样的消除操作．设 `val` 和 `cnt` 两个变量分别代表当前的主元素候选和目前如果进行了这样的消除操作后主元素候选会剩多少个．初始时 `cnt` 置为 0．每次从数据流中取出一个元素，如果当前 `cnt` 为 0，则代表主元素候选已经被消除完了，当前记录的 `val` 一定不是主元素，因此设置当前元素为主元素候补．之后检查当前元素是否是主元素候补，如果是，则 `cnt` 增加 1，如果不是，则该元素应与一个主元素候补一起消除，`cnt` 减少 1．重复上述操作直到数据流读取完成，`val` 即为主元素．

⚠ 注意

当原数据中不存在主元素时，此算法给出的结果是错误的．如要判断序列中是否存在主元素，需要再次读入数据流，统计 `val` 出现次数，判断其是否超过 $n/2$．

为了再次读入数据流，可以选择重置输入位置指示器，可利用 `std::basic_istream<CharT,Traits>::seekg`[3]（流式输入）或 `rewind`[4]、`fseek`[5]（C 风格输入）等库函数．

例题

洛谷 P2397 yyy loves Maths VI (mode)[6]

求给定序列的主元素．

参考代码

```cpp
#include <cstdio>

int main() {
  int n;
  scanf("%d", &n);
  int val, cnt = 0;
  while (n--) {
    int x;
    scanf("%d", &x);
    if (!cnt) val = x;
```

[1] 多数投票算法 - 维基百科[2]

```
    (val == x) ? ++cnt : --cnt;
  }
  printf("%d\n", val);
  return 0;
}

```

LeetCode 229. 多数元素 II[7]

给定一个大小为 n 的整数数组，找出其中所有出现超过 $\lfloor n/3 \rfloor$ 次的元素.

参考代码

```
class Solution {
 public:
  vector<int> majorityElement(vector<int>& nums) {
    // 将多数投票算法的「抵消2个不同元素」变为「抵消3个两两不同的元素」
    int maj1, maj2;
    int cnt1 = 0, cnt2 = 0;
    for (int num : nums) {
      if (num == maj1) {
        ++cnt1;
      } else if (num == maj2) {
        ++cnt2;
      } else if (cnt1 == 0) {
        maj1 = num;
        ++cnt1;
      } else if (cnt2 == 0) {
        maj2 = num;
        ++cnt2;
      } else {
        --cnt1;
        --cnt2;
      }
    }

    // 由于题目没有保证存在2个超过 ⌊ n/3 ⌋ 次的元素，故需检验
    vector<int> ans;
    cnt1 = 0, cnt2 = 0;
    for (int num : nums) {
      if (num == maj1)
        ++cnt1;
      else if (num == maj2)
        ++cnt2;
    }
    int n = nums.size();
    if (cnt1 > n / 3) ans.push_back(maj1);
    if (cnt2 > n / 3) ans.push_back(maj2);

    return ans;
  }
```

```
39 };
40
```

参考资料

[1] nth_element 

[2] 多数投票算法 - 维基百科 

[3] std::basic_istream<Char, Traits>::seekg 

[4] rewind 

[5] fseek 

[6] 洛谷 P2397 yyy loves Maths VI (mode) 

[7] LeetCode 229. 多数元素 II 

13.15 Garsia–Wachs 算法

简介

Garsia–Wachs 算法（Garsia–Wachs Algorithm）是计算机用来在 **线性时间** 内构建 **最优二叉查找树** 和 **字母霍夫曼码** 的有效算法．它以 Adriano Garsia 和 Michelle L. Wachs 的名字命名，他们于 1977 年发表了相关论文.

问题描述

一个整数 n，对于 $n+1$ 个非负权值 $w_0, w_1, ..., w_n$，构造一棵有根且 n 个内部节点都有两个子节点的二叉树，这意味着这棵二叉树有 $n+1$ 个叶节点．我们将 $n+1$ 的输入序列与二叉树结点顺序一一映射，目标是在所有具有 n 个内部节点的可能的树结构中找到一棵树，使外部从根到每个叶子的路径长度的权重和最小.

最优二叉查找树

这个问题可以理解为 n 个有序键构造二叉查找树的问题，假设树将仅用于搜索树中不存在的值．在这种情况下，n 个键将搜索值的空间划分为 $n+1$ 个区间，并且这些区间之一的权重可以作为搜索值落在那个区间的概率．外部路径长度的加权和控制了查找的预期时间.

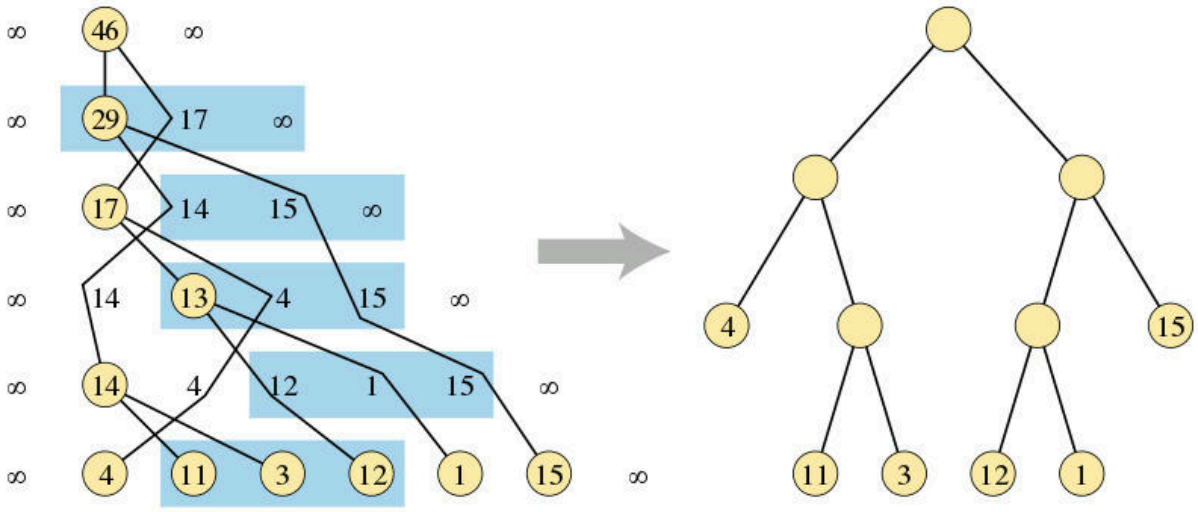
字母霍夫曼码

这个问题也可以用作构建霍夫曼码．这是一种通过使用二进制值的可变长度序列明确编码 $n+1$ 给定值的方法．在这种解释中，值的代码由从树中的根到叶子的路径上从父到子的左步和右步序列给出（例如，左为 0，右为 1）．与标准霍夫曼码不同，以这种方式构造的霍夫曼码是按字母顺序排列的，也就是说这些二进制码的排序顺序与值的输入顺序相同．如果一个值的权重是它在编码消息中的频率，那么 Garsia–Wachs 算法的输出是将消息长度压缩到最短的，按字母顺序排列的霍夫曼代码.

过程

Garsia–Wachs 算法一般包括三个阶段：

1. 构建一个值位于叶子的二叉树，注意顺序可能错误.
2. 计算树中根到每个叶子的距离.
3. 构建另一个二叉树，叶子的距离相同，但顺序正确.



如上图所示，在算法的第一阶段，通过查找合并输入序列的无序三元组构建的二叉树（左侧），和算法输出的正确排序的二叉树，其中叶子高度与另一棵树一样.

如果输入在序列的开始和结束处增加了两个标记值 ∞（或任何足够大的有限值），则算法的第一阶段更容易描述. 所以在竞赛题解中使用 Garsia–Wachs 算法时，对于一个长度为 n 的数组 num，我们一般定义 $num[0] = num[n+1] = \infty$.

第一阶段维护了一个由最初为每个非标志（non-sentinel）输入权重创建的单节点树组成的森林. 每棵树都与一个值相关联，其叶子的权重之和为每个非标志输入权重构成一个树节点. 为了维护这些值的序列，每端会有两个标记值. 初始序列只是叶权重作为输入的顺序. 然后重复执行以下步骤，每一步都减少输入序列的长度，直到只有一棵树包含了所有叶子：

- 在序列中找到前三个连续的权重值 x，y，z 使得 $x \le z$. 因为序列结尾的标志值大于之前的任意两个有限值，所以总是存在这样的三元组.
- 从序列中移除 x 和 y，并创建一个新的树节点作为 x 和 y 节点的父节点，值为 $x + y$.
- 在原来 x 的位置以前大于或等于 $x + y$ 且距 x 最近的值的右边重新插入新节点. 因为左标志值的存在，所以总是存在这样的位置.

为了有效地实现这一阶段，该算法可以在任何平衡二叉查找树结构中维护当前值序列. 这样的结构允许我们在对数时间内移除 x 和 y，并重新插入它们的新父节点. 在每一步中，数组中位于偶数索引上直到 y 值的权重形成了一个递减序列，位于奇数索引位的权重形成另一个递减序列. 因此，重新插入 $x + y$ 的位置可以通过在对数时间内对这两个递减序列使用平衡树执行两次二分查找找到. 通过从前一个三元组 z 值开始的线性顺序搜索，我们可以在总线性时间复杂度内执行对满足 $x \le z$ 的第一个位置的搜索.

Garsia–Wachs 算法的第三阶段的证明，即存在另一棵具有相同距离的树并且这棵树提供了问题的最优解，是很重要的. 但是由于其证明方式有多种且过于复杂，此处略去. 在第三阶段为正确的前提下，第二和第三阶段很容易在线性时间内实现. 因此，在长度为 n 的输入序列上，Garsia–Wachs 算法的总时间复杂度为 $O(n \log n)$.

应用

函数性编程语言 Haskell 的 garsia-wachs package[1] 对 Garsia–Wachs 算法做了函数性实现. 它主要用于构建最佳搜索表，或者以最优复杂度平衡 rope[2] 数据结构.

> **注释**
>
> **rope** 是 Haskell 语言中用于操作带有可选注释的字节串（bytestring）手指树→10.23@p2587 的工具.

例题

POJ 1738 An old Stone Game[3]

有一个古老的石头游戏．在游戏开始时，玩家将 $n(1 \leq n \leq 50000)$ 堆石头排成一行．目标是将石头合并成一堆，规则如下：在游戏的每一步，玩家可以将相邻的两个堆合并成一个新的堆．分数是新堆的石头总数．请计算总分中的最小值．

解题思路

石子合并的题目很经典，一般我们可以用区间 DP 解答，但是当数据量很大，例如此题中的 n $(1 \leq n \leq 50000)$ 时，用 Garsia–Wachs 算法求解更高效：第一步，初始化一个大小为 n 的数组 $num[n]$，其中 $num[0] = num[n+1] = \infty$．第二步，每次找到一个最小的 i 使 $num[i-1] \leq num[i+1]$，并将 $num[i-1], num[i]$ 合并为 $temp$ 找到前面一个最大的 j 使得 $num[j] > temp$, 将 $temp$ 移到 j 后面．重复这一步直到剩余堆数为 1． 关于每次只能合并相邻石子堆的要求，因为 $num[j] \geq num[i-1] + num[i]$，我们可以将 $num[j+1]$ 到 $num[i-2]$ 看成一个 $num[mid]$ 的整体，所以一定是先合并 sum．因此没有违背题目要求．

ATCODER N-Slimes[4]

N 个史莱姆排成一排．最初左边第 i 个史莱姆的大小为 a_i．Taro 试图将所有史莱姆组合成一个更大的史莱姆．他会反复执行以下操作，直到只有一个史莱姆： 选择两个相邻的史莱姆，并将它们组合成一个新的史莱姆．新的史莱姆的大小为 $x + y$，其中 x 和 y 是组合之前史莱姆的大小．这一步骤有产生 $x + y$ 的成本．合成史莱姆时史莱姆的位置关系不会改变．找出可能发生的最小总成本．

参考资料与拓展阅读

1. Garsia–Wachs algorithm - Wikipedia[5]
2. Data.Algorithm.GarsiaWachs - Hackage Haskell[6]
3. garsia-wachs: A Functional Implementation of the Garsia-Wachs Algorithm[7]
4. Sentinel value - Wikipedia[8]
5. A new proof of the Garsia-Wachs algorithm[9]

外部参考资料

[1] garsia-wachs package

[2] rope

[3] POJ 1738 An old Stone Game

[4] ATCODER N-Slimes

[5] Garsia–Wachs algorithm - Wikipedia

[6] Data.Algorithm.GarsiaWachs - Hackage Haskell

[7] garsia-wachs: A Functional Implementation of the Garsia-Wachs Algorithm

[8] Sentinel value - Wikipedia

[9] A new proof of the Garsia-Wachs algorithm

13.16 15-puzzle

简介

15 - 拼图（英文：15-puzzle, 又名 Gem Puzzle，Boss Puzzle，Game of 15，Mystic Square，N-puzzle, etc）是一个滑块类游戏（英文：sliding puzzle）．滑块方盘的长宽均为 4×4 个方块，其中 15 个位置放序号打乱的

方块，剩下一个为空位．与空位同行或同列的方块可以通过水平或垂直滑动来移动．拼图的目标是按编号顺序排列方块．

15 - 拼图常见别称为 **n - 拼图**，其中数字 n 指的是方盘中的方块总数．15 - 拼图的不同尺寸变体亦使用了类似的名称，例如 8 拼图指的是置于 3×3 方盘中的 8 个方块．但 15 拼图也可以称为 16 拼图，此处的 16 指的是方块容量．它的扩展问题有时也包括了 $n \times m$ 的滑动方盘．

15 - 拼图是涉及 启发式算法→6.5@p665 建模的经典问题．此问题的常见形式是 曼哈顿距离→12.4@p3089 和错位方块的数量计算，二者都是可接受启发（英文：admissible heuristic），即它们永远不会高估剩余的移动次数，这确保了某些搜索算法（例如 A * 算法→6.6@p667）的最优性．

> **注释**
>
> **滑块游戏** 是一类在平面上滑动方块以组成特定排列的智力游戏．常见的滑块游戏包括数字拼图、华容道和塞车时间．其中 15 - 拼图是最古老的滑块类游戏，发明者是 Noyes Chapman，该游戏风靡于 1880 年代．不像其它 tour 类的解谜游戏，滑块游戏禁止任何一个方块离开盘面，这个特性区别于重新排列类的解谜游戏．

定义

给定一个 4×4 的方盘，其中 15 个方块随意排列．我们需要将它按照序号排列成下图所示的样子．移动规则为每次只能交换空方块和其相邻一个方块的位置．常见问题为找到可解决此问题的最少步骤，计算错位方位的个数，或找出是否能得到最终的有序排列．

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

可解性证明

Johnson & Story (1879) 证明，如果 m 和 n 都至少为 2，则逆向适用于大小为 $m \times n$ 的棋盘：通过从 $m = n = 2$ 开始对 m 和 n 进行归纳证明，所有偶数排列都是可解的．Archer (1999) 给出了另一个证明，基于通过汉密尔顿路径定义等价类．

算法

寻找数字滑盘游戏的一个解相对容易，但寻找 **最优解** 是一个 NP **困难** 问题．15-Puzzle 的最优解至多有 80 步；而 8-Puzzle 的最优解至多有 31 步．

N-Puzzle 支持常见的基于图的搜索算法，如广度优先搜索和深度优先搜索，同样我们也可以用 A * 搜索→6.6@p667 算法寻找最优解．启发式函数 $h(n)$ 可以是

- 放错的方块的数量．
- 所有放错的方块到各自目标位置的欧几里得距离之和．
- 所有放错的方块到各自目标位置的曼哈顿距离之和．

群理论

因为 15 块的数字推盘游戏组合可以由「3 循环」（英文：3-cycles）产生，所以可以证明 15 块的数字推盘游戏可以用交错群 A_{15} 表示．事实上，任何使用 $2 \times k - 1$ 块相同面积正方形方块的数字滑盘游戏皆可以以交错群 A_{2k-1} 表示．

习题

- N Puzzle[1]
- A. Amity Assessment[2]
- Sliding Puzzle[3]
- POJ 1077 - Eight[4]

参考资料与拓展阅读

1. 15 puzzle - Wikipedia[5]
2. jrdnjacobson,How to Solve the 15 Puzzle - instructables[6]
3. Korf, R. E. (2000),"Recent Progress in the Design and Analysis of Admissible Heuristic Functions"[7], in Choueiry, B. Y.; Walsh, T. (eds.), Abstraction, Reformulation, and Approximation (PDF), SARA 2000. Lecture Notes in Computer Science, vol. 1864, Springer, Berlin, Heidelberg, pp. 45–55, doi:10.1007/3-540-44914-0_3, ISBN 978-3-540-67839-7, retrieved 2010-04-26
4. Welcome to N-Puzzle - web demo[8]

外部参考资料

[1] N Puzzle

[2] A. Amity Assessment

[3] Sliding Puzzle

[4] POJ 1077 - Eight

[5] 15 puzzle - Wikipedia

[6] How to Solve the 15 Puzzle - instructables

[7] "Recent Progress in the Design and Analysis of Admissible Heuristic Functions"

[8] Welcome to N-Puzzle - web demo

13.17 Kahan 求和

引入

Kahan 求和 算法，又名补偿求和或进位求和算法，是一个用来 **降低有限精度浮点数序列累加值误差** 的算法．它主要通过保持一个单独变量用来累积误差（常用变量名为 c）来完成的．

该算法主要由 William Kahan 于 1960s 发现．因为 Ivo Babuška 也曾独立提出了一个类似的算法，Kahan 求和算法又名为 Kahan–Babuška 求和算法．

舍入误差

在计算机程序中，我们需要用有限位数对实数做近似表示，如今的大多数计算机都使用 IEEE-754[1] 规定的浮点数来作为这个近似表示．对于 $\frac{1}{3}$，由于我们不能在有限位数内对它进行精准表示，因此在使用

IEEE-754 表示法时，必须四舍五入一部分数值（truncate）．这种 **舍入误差**（Rounding off error）是浮点计算的一个特征．

在浮点加法计算中，交换律（commutativity）成立，但结合律（associativity）不成立．也就是说，$a+b=b+a$ 但 $(a+b)+c \neq a+(b+c)$．因此在浮点序列加法计算中，我们可以从左到右一个个累加，也可以在原有顺序上，将他们两两分成一对．第二种算法会相对较慢并需要更多内存，也常被一些语言的特定求和函数使用，但相对结果更准确．

为了得到更准确的浮点累加结果，我们需要使用 Kahan 求和算法．

在计算 $S_{new}=S_{old}+a$（a 为浮点序列的一个数值）时，定义实际计算加入 S 的值为 $a_{eff}=S_{new}-S_{old}$，如果 a_{eff} 比 a 大，则证明有向上舍入误差；如果 a_{eff} 比 a 小，则证明有向下舍入误差．则舍入误差定义为 $E_{roundoff}=a_{eff}-a$．那么用来纠正这部分舍入误差的值就为 $a-a_{eff}$，即 $E_{roundoff}$ 的负值．定义 c 是对丢失的低位进行运算补偿的变量，就可以得到 $c_{new}=c_{old}+(a-a_{eff})$．

过程

Kahan 求和算法主要通过一个单独变量用来累积误差．如下方参考代码所示，sum 为最终返回的累加结果．c 是对丢失的低位进行运算补偿的变量（其被舍去的部分），也是 Kahan 求和算法中的必要变量．

因为 sum 大，y 小，所以 y 的低位数丢失．$(t-sum)$ 抵消了 y 的高阶部分，减去 y 则会恢复负值（y 的低价部分）．因此代数值中 c 始终为零．在下一轮迭代中，丢失的低位部分会被更新添加到 y．

实现

参考代码

```cpp
float kahanSum(vector<float> nums) {
  float sum = 0.0f;
  float c = 0.0f;
  for (auto num : nums) {
    float y = num - c;
    float t = sum + y;
    c = (t - sum) - y;
    sum = t;
  }
  return sum;
}
```

习题

在 OI 中，Kahan 求和主要作为辅助工具存在，为计算结果提供误差更小的值．

例题 CodeForces Contest 800 Problem A. Voltage Keepsake[2]

有 n 个同时使用的设备．第 i 个设备每秒使用 a_i 单位的功率．这种用法是连续的．也就是说，在 λ 秒内，设备将使用 $\lambda \times a_i$ 单位的功率．第 i 个设备当前存储了 b_i 单位的电力．所有设备都可以存储任意数量的电量．有一个可以插入任何单个设备的充电器．充电器每秒会为设备增加 p 个单位的电量．这种充电是连续的．也就是说，如果将设备插入 λ 秒，它将获得 $\lambda \times p$ 单位的功率．我们可以在任意时间单位内（包括实数）切换哪个设备正在充电（切换所需时间忽略不计）．求其中一个设备达到 0 单位功率前，可以使用这些设备的最长时间．

例题 CodeForces Contest 504 Problem B. Misha and Permutations Summation[3]

定义数字 $0, 1, \cdots, (n-1)$ 的两个排列 p 和 q 的和为 $Perm((Ord(p) + Ord(q)) \bmod n!)$，其中 $Perm(x)$ 是数字 $0, 1, \cdots, (n-1)$ 的第 x 个字典排列（从零开始计数），$Ord(p)$ 是字典序排列 p 的个数. 例如，$Perm(0) = (0, 1, \cdots, n-2, n-1)$，$Perm(n!-1) = (n-1, n-2, \cdots, 1, 0)$). Misha 有两个排列 p 和 q，找到它们的总和.

编程语言的求和

Python 的标准库指定了精确舍入求和的 fsum[4] 函数可用于返回可迭代对象中值的准确浮点总和，它通过使用 Shewchuk 算法跟踪多个中间部分和来避免精度损失.

Julia 语言中，sum[5] 函数的默认实现是成对求和，以获得高精度和良好的性能. 同时外部库函数 sum_kbn[6] 为需要更高精度的情况提供了 Neumaier 变体的实现，具体可见 KahanSummation.jl[7].

参考资料与注释

1. Kahan_summation_algorithm - Wikipedia[8]
2. Kahan summation - Rosetta Code[9]
3. VK Cup Round 2 + Codeforces Round 409 Announcement[10]
4. Rounding off errors in Java - GeeksforGeeks[11]

外部参考资料

[1] IEEE-754

[2] CodeForces Contest 800 Problem A. Voltage Keepsake

[3] CodeForces Contest 504 Problem B. Misha and Permutations Summation

[4] fsum

[5] sum

[6] sum_kbn

[7] KahanSummation.jl

[8] Kahan_summation_algorithm - Wikipedia

[9] Kahan summation - Rosetta Code

[10] VK Cup Round 2 + Codeforces Round 409 Announcement

[11] Rounding off errors in Java - GeeksforGeeks

13.18 珂朵莉树/颜色段均摊

简介

珂朵莉树（Chtholly Tree），又名老司机树 ODT（Old Driver Tree）. 起源自 CF896C[1].

这个名称指代的是一种「使用平衡树（`std::set`、`std::map` 等）或链表（`std::list`、手写链表等）维护颜色段均摊」的技巧，而不是一种特定的数据结构. 其核心思想是将值相同的一段区间合并成一个结点处理. 相较于传统的线段树等数据结构，对于含有区间覆盖的操作的问题，珂朵莉树可以更加方便地维护每个被覆盖区间的值.

实现（std::set）

结点类型

```
struct Node_t {
  int l, r;
  mutable int v;
```

```
4
5   Node_t(const int &il, const int &ir, const int &iv) : l(il), r(ir), v(iv) {}
6
7   bool operator<(const Node_t &o) const { return l < o.l; }
8 };
```

其中，`int v` 是你自己指定的附加数据.

> **`mutable` 关键字的含义是什么?**
>
> `mutable` 的意思是「可变的」，让我们可以在后面的操作中修改 `v` 的值. 在 C++ 中，mutable 是为了突破 const 的限制而设置的. 被 mutable 修饰的变量（mutable 只能用于修饰类中的非静态数据成员），将永远处于可变的状态，即使在一个 const 函数中.
>
> 这意味着, 我们可以直接修改已经插入 `set` 的元素的 `v` 值, 而不用将该元素取出后重新加入 `set`.

结点存储

我们希望维护所有结点, 使得这些结点所代表的区间左端点单调增加且两两不交, 最好可以保证所有区间的并是一个极大的连续范围. 此处以 `std::set` 为例，用一个 `set<Node_t> odt;` 维护所有结点.

初始化时, 向珂朵莉树中插入一个极长区间(如题目要求维护位置 1 到 n 的信息, 插入区间 $[1, n+1]$).

split 操作

`split` 操作是珂朵莉树的核心. 它接受一个位置 x，将原本包含点 x 的区间（设为 $[l, r]$）分裂为两个区间 $[l, x)$ 和 $[x, r]$，并返回指向后者的迭代器.

参考代码如下:

```
auto split(int x) {
  auto it = odt.lower_bound(Node_t(x, 0, 0));
  if (it != odt.end() && it->l == x) return it;
  --it;
  int l = it->l, r = it->r, v = it->v;
  odt.erase(it);
  odt.insert(Node_t(l, x - 1, v));
  return odt.insert(Node_t(x, r, v)).first;
}
```

在不支持使用 `auto` 进行返回类型推导的编译器上，可以将函数的返回类型改为 `set<Node_t>::iterator`.

assign 操作

另外一个重要的操作：`assign`. 用于对一段区间进行赋值. 设将要对区间 $[l, r]$ 赋值为 v.

首先, 将区间 $[l, r]$ 截取出来. 依次调用 `split(r + 1), split(l)`, 将此两者返回的迭代器记作 itr, itl, 那么 $[itl, itr)$ 这个迭代器范围就指向了珂朵莉树中 $[l, r]$ 包含的所有区间.

然后，将原有的信息删除. `std::set` 有成员方法 `erase`, 签名如同 `iterator erase( const_iterator first, const_iterator last );`, 可以移除范围 `[first; last)` 中的元素. 于是我们调用 `odt.erase(itl, itr);` 以删除原有的信息.

最后，插入区间 $[l, r]$ 的新值. 调用 `odt.insert(Node_t(l, r, v))` 即可.

参考代码如下:

```
void assign(int l, int r, int v) {
```

```
  auto itr = split(r + 1), itl = split(l);
  odt.erase(itl, itr);
  odt.insert(Node_t(l, r, v));
}
```

为什么需要先 split(r + 1) 再 split(l)?

1. std::set::erase 方法将使指向被擦除元素的引用和迭代器失效. 而其他引用和迭代器不受影响.
2. std::set::insert 方法不会使任何迭代器或引用失效.
3. split 操作会将区间拆开. 调用 split(r + 1) 之后 $r+1$ 会成为两个新区间中右边区间的左端点, 此时 split 左区间, 必然不会访问到 $r+1$ 为左端点的那个区间, 也就不会将其拆开, 删去 $r+1$ 为左端点的区间, 使迭代器失效. 反之, 先 split(l), 再 split(r + 1), 可能会把 l 为左端点的区间删去, 使迭代器失效.

perform 操作

将珂朵莉树上的一段区间提取出来并进行操作. 与 assign 操作类似, 只不过是将删除区间改为遍历区间.

参考代码如下:

```
void perform(int l, int r) {
  auto itr = split(r + 1), itl = split(l);
  for (; itl != itr; ++itl) {
    // Perform Operations here
  }
}
```

注意不应该滥用这样的提取操作, 可能使得时间复杂度错误. 见下文「复杂度分析」一栏.

实现（std::map）

相较于 std::set 的实现, std::map 的实现的 split 操作写法更简单. 除此之外, 其余操作与 std::set 并无二异.

结点存储

由于珂朵莉树存储的区间是连续的, 我们不一定要记下右端点是什么. 不妨使用一个 map<int, int> mp; 存储所有区间, 其键维护左端点, 其值维护其对应的左端点到下一个左端点之前的值.

初始化时, 如题目要求维护位置 1 到 n 的信息, 则调用 mp[1] = -1, mp[n + 1] = -1 表示将 $[1, n+1)$ 即 $[1, n]$ 都设为特殊值 -1, $[n+1, +\infty)$ 这个区间当作哨兵使用, 也可以对它进行初始化.

split 操作

参考代码: (第一份)

```
void split(int x) {
  auto it = prev(mp.upper_bound(x));  // 找到左端点小于等于 x 的区间.
  mp[x] = it->second;  // 设立新的区间, 并将上一个区间储存的值复制给本区间.
}
```

参考代码: (第二份)

```
auto split(int pos) {
```

```
  auto it = prev(mp.upper_bound(pos));  // 找到左端点小于等于 x 的区间.
  return mp.insert(it, make_pair(pos, it->second));
  // 设立新的区间，并将上一个区间储存的值复制给本区间.
}
```

这里使用了 `std::map::insert` 的重载 `iterator insert( const_iterator pos, const value_type& value );`，其插入 `value` 到尽可能接近正好在 `pos` 之前的位置. 如果插入恰好发生在正好在 `pos` 之前的位置，那么复杂度是均摊常数，否则复杂度与容器大小成对数.

assign 操作

对于 assign 操作，我们需要把 $[l, r-1]$ 内所有区间左端点删除，再建立新的区间.

```
void assign(int l, int r, int v) {  // 注意，这里的r是区间右端点+1
  split(l);
  split(r);
  auto it = mp.find(l);
  while (it->first != r) {
    it = mp.erase(it);
  }
  mp[l] = v;
}
```

perform 操作

```
void perform(int l, int r) {  // 注意，这里的r是区间右端点+1
  split(l);
  split(r);
  auto it = mp.find(l);
  while (it->first != r) {
    // Perform Operations here
    it = next(it);
  }
}
```

实现（链表）

目前主流的实现是基于 `set` 来维护节点，但由于平均维护的区间个数很小，`set` 的优势并不明显. 相比之下，链表（或数组）能更简洁地维护分裂与合并操作.

结点存储

```
using i64 = int64_t;

struct Block {
  Block *next;  // 链表下一节点
  int l, r;     // 区间范围
  i64 val;      // 区间上的值

  Block(Block *next, int l, int r, i64 val)
      : next(next), l(l), r(r), val(val) {}

  bool operator<(const Block &b) const { return val < b.val; }
} *root;
```

split 操作

```
// 返回左端点为 mid+1 的区间
Block *split(int mid) {
  for (Block *b = root; b; b = b->next) {  // 遍历链表
    if (b->l == mid + 1) {                 // 左端点为 mid+1
      return b;
    }
    // 寻找能包含 mid 和 mid+1 的区间 [l, r]，将其被拆分成 [l, mid] 和 [mid+1,
    // r]
    if (b->l <= mid && mid + 1 <= b->r) {
      b->next = new Block(b->next, mid + 1, b->r, b->val);
      b->r = mid;
      return b->next;
    }
  }
  return nullptr;  // 未找到，返回空
}
```

在操作区间时，由于不能只维护区间的一部分，所以下面的操作进行之前都需要预先分裂区间，再完成相应操作.

```
Block *lb, *rb;

// 预分裂，保证后续操作在 [l, r] 内部
void prepare(int l, int r) {
  lb = split(l - 1);
  rb = split(r);
}
```

assign 操作

```
void assign(int l, int r, i64 val) {
  prepare(l, r);
  lb->r = r;  // 将区间 [lb.l, lb.r] 修改成 [lb.l, r]
  lb->val = val;
  lb->next = rb;  // 将 [lb.l, r] 链至其右侧相邻区间
}

// 注：这里没有释放被删除节点的内存，若有需要可自行添加
```

perform 操作

```
void perform(int l, int r) {
  prepare(l, r);
  for (Block *b = lb; b != rb; b = b->next) {
    // Perform Operations here
  }
}
```

复杂度分析

perform 以后立即对同一区间调用 assign

此时观察发现，两次 `split` 操作至多增加两个区间；一次 `assign` 将删除范围内的所有区间并增加一个区间，同时遍历所删除的区间．所以，我们所遍历的区间与所删除的区间数量成线性，而每次操作都只会增加 $O(1)$ 个区间，所以我们操作的区间数量关于操作次数（包括初始化）成线性，时间复杂度为均摊 $O(m \log n)$，其中 m 为操作次数，n 为珂朵莉树中最大区间个数（可以认为 $n \leq m$）．

perform 以后不进行 assign

如果允许特殊构造数据，这样一定是能被卡掉的，只需要使珂朵莉树中有足够多的不同区间并反复遍历，就能使珂朵莉树的复杂度达到甚至高于平方级别．

如果要保证复杂度正确，必须保证数据随机．详见 Codeforces 上关于珂朵莉树的复杂度的证明[2]．更详细的严格证明见 珂朵莉树的复杂度分析[3]．证明的结论是：用 `std::set` 实现的珂朵莉树的复杂度为 $O(n \log \log n)$，而用链表实现的复杂度为 $O(n \log n)$．

习题

- 「Luogu 1840」Color the Axis[4]
- ~~「SCOI2010」序列操作~~[5]（该题目来源已添加 Hack 数据）
- 「SHOI2015」脑洞治疗仪[6]
- 「Luogu 4979」矿洞：坍塌[7]
- 「Luogu 8146」risrqnis[8]

扩展阅读

ODT 的映射思想的推广 - 洛谷专栏 (luogu.com.cn)[9]

参考资料和注释

- Problem - 896C - Codeforces[10]（珂朵莉树的起源）
- CF896C Willem, Chtholly and Seniorious 题解 - 洛谷专栏 (luogu.com.cn)[11]（`std::set` 实现参考）
- 珂朵莉树的 map 实现 - 知乎 (zhihu.com)[12]（`std::map` 实现参考）
- 题解 CF896C【Willem, Chtholly and Seniorious】- 洛谷专栏 (luogu.com.cn)[13]（链表实现参考）
- Codeforces Round #449 Editorial - Codeforces[14]（关于珂朵莉树的复杂度的证明）
- 珂朵莉树的复杂度分析 - 知乎 (zhihu.com)[15]（珂朵莉树的复杂度分析）

外部参考资料

[1] CF896C

[2] Codeforces 上关于珂朵莉树的复杂度的证明

[3] 珂朵莉树的复杂度分析

[4]「Luogu 1840」Color the Axis

[5]「SCOI2010」序列操作

[6]「SHOI2015」脑洞治疗仪

[7]「Luogu 4979」矿洞：坍塌

[8]「Luogu 8146」risrqnis

[9] ODT 的映射思想的推广 - 洛谷专栏 (luogu.com.cn)

[10] Problem - 896C - Codeforces

[11] CF896C Willem, Chtholly and Seniorious 题解 - 洛谷专栏 (luogu.com.cn)

[12] 珂朵莉树的 map 实现 - 知乎 (zhihu.com)

[13] 题解 CF896C【Willem, Chtholly and Seniorious】- 洛谷专栏 (luogu.com.cn)

[14] Codeforces Round #449 Editorial - Codeforces

[15] 珂朵莉树的复杂度分析 - 知乎 (zhihu.com)

13.19 空间优化简介

空间优化相关技巧在算法竞赛中不太常见，但仍有讨论价值.

信息熵

信息熵描述了存储数据所占用的空间下限，若实际可用的空间低于这个下限则必然损失信息.

定义

对随机变量 X，定义信息熵为

$$H(X)=-\sum_{x}P(X=x)\log_2 P(X=x).$$

定义中对数底数为 2 是因为计算机中存储的信息每位只有 2 种取值：0 和 1.

例如设 X 服从 $\{1,2,\ldots,n\}$ 上的均匀分布，则其信息熵为

$$H(X)=-\sum_{i=1}^{n}\frac{1}{n}\log_2\frac{1}{n}=\log_2 n,$$

所以我们至少需要 $\log_2 n$ 位来存储 1 到 n 的整数.

例题

[WC2022] 猜词[1]

交互题，你需要在有限次数内猜一个 5 个字母的单词. 每次猜测都需要猜一个词库中存在的单词. 如果猜对了，游戏结束；在每次猜错后，交互库会返回哪些字母的位置是正确的，以及哪些字母在待猜单词中出现了但位置是错误的.

解法

参见 用信息论解 Wordle 谜题 - 3Blue1Brown[2].

考虑计算信息熵，显然每次猜词时选择信息熵高的词可使得期望猜词次数尽可能小.

由于本题在猜测之前给出了答案首字母，所以我们可以预处理出每种首字母的最优猜测.

另外若剩余的词很少的话，我们可以考虑优先输出可能是答案的词，从而减小次数.

常见技巧

避免存储不必要的数据

例如：

- 在 可持久化线段树→10.17.2@p2424 中，由于单次修改只会产生 $O(\log n)$ 个新结点，所以我们不需要把每个版本的线段树都完整地存储下来，只需要记录新结点即可.
- 考虑 图的存储→11.3@p2605，对稀疏图来说，若使用邻接矩阵则会存储大量无用的 0，所以一般使用邻接表存稀疏图.
- `bool` 数组的每个元素均会占用一个字节的空间，必要时可用每个元素只占用一位的 `std::vector<bool>` 或 bitset→4.3.4@p389 代替.
- 在 背包 DP→7.4.1@p724 中，对 01 背包而言，每次计算 DP 值只会用到上一次计算时的数据，所以我们可以用滚动数组优化空间，只需要记录当前 DP 值即可.

利用数据特性

考虑支持路径压缩和启发式合并的 并查集→10.6.1@p1999，传统做法需要两个数组，分别记录父结点编号和子树大小.

注意到：

1. 在应用了路径压缩后，对于并查集中的一棵树，我们只需要记录根结点对应的子树大小.
2. 根结点的父亲一定是自己.

我们可以利用有符号整数的特性，用负数表示根结点，正数表示非根结点，所以我们只需一个数组即可实现支持路径压缩和启发式合并的并查集.

实现

```cpp
#include <cstdint>
#include <vector>
using i32 = int32_t;
using u32 = uint32_t;

class dsu {
  // p[i] < 0 时表示 i 为根节点，其对应的子树大小为 -p[i]
  // p[i] >= 0 时表示 i 不为根节点，其父亲节点的编号为 p[i]
  std::vector<i32> p;

 public:
  // 节点编号从 0 到 sz-1
  explicit dsu(u32 sz) : p(sz, -1) {}

  i32 find(u32 x) { return p[x] < 0 ? (i32)x : p[x] = find((u32)p[x]); }

  u32 size(u32 x) { return (u32)-p[(u32)find(x)]; }

  bool same_root(u32 x, u32 y) { return find(x) == find(y); }

  bool merge(u32 x, u32 y) {
    if ((x = (u32)find(x)) == (y = (u32)find(y))) return false;
    if (p[x] > p[y]) std::swap(x, y);  // 启发式合并
    p[x] += p[y], p[y] = (i32)x;
    return true;
  }
};
```

习题

- QOJ 6669 Mapa[3]
- [SDOI/SXOI2022] 无处存储[4]

参考资料与拓展阅读

1. 陈知轩.《浅谈信息学竞赛中的空间优化问题》. 2022 国家集训队论文
2. Information theory - Wikipedia[5]
3. 浅谈信息论 - 洛谷专栏[6]

外部参考资料

[1] [WC2022] 猜词

[2] 用信息论解 Wordle 谜题 - 3Blue1Brown

[3] QOJ 6669 Mapa

[4] [SDOI/SXOI2022] 无处存储

[5] Information theory - Wikipedia

[6] 浅谈信息论 - 洛谷专栏

第十四章

专题

14.1 RMQ

简介

RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值.

在接下来的描述中，默认初始数组大小为 n，询问次数为 m.

在接下来的描述中，默认时间复杂度标记方式为 $O(A) \sim O(B)$，其中 $O(A)$ 表示预处理时间复杂度，而 $O(B)$ 表示单次询问的时间复杂度.

单调栈

由于 OI Wiki 中已有此部分的描述，本文仅给出 链接→10.9@p2067. 这部分不再展开.

时间复杂度 $O(m \log m) \sim O(\log n)$，空间复杂度 $O(n)$.

ST 表

由于 OI Wiki 中已有此部分的描述，本文仅给出 链接→10.11@p2072. 这部分不再展开.

时间复杂度 $O(n \log n) \sim O(1)$，空间复杂度 $O(n \log n)$.

线段树

由于 OI Wiki 中已有此部分的描述，本文仅给出 链接→10.13.1@p2098. 这部分不再展开.

时间复杂度 $O(n) \sim O(\log n)$，空间复杂度 $O(n)$.

Four Russian

Four russian 是一个由四位俄罗斯籍的计算机科学家提出来的基于 ST 表的算法.

在 ST 表的基础上 Four russian 算法对其做出的改进是序列分块.

具体来说，我们将原数组——我们将其称之为数组 A——每 S 个分成一块，总共 n/S 块.

对于每一块我们预处理出来块内元素的最小值，建立一个长度为 n/S 的数组 B，并对数组 B 采用 ST 表的方式预处理.

同时，我们对于数组 A 的每一个零散块也建立一个 ST 表.

询问的时候，我们可以将询问区间划分为不超过 1 个数组 B 上的连续块区间和不超过 2 个数组 A 上的整块内的连续区间. 显然这些问题我们通过 ST 表上的区间查询解决.

在 $S = \log n$ 时候，预处理复杂度达到最优，为 $O((n/\log n)\log n + (n/\log n) \times \log n \times \log\log n) = O(n \log\log n)$.

时间复杂度 $O(n \log\log n) \sim O(1)$，空间复杂度 $O(n \log\log n)$.

当然询问由于要跑三个 ST 表，该实现方法的常数较大.

一些小小的算法改进

我们发现，在询问的两个端点在数组 A 中属于不同的块的时候，数组 A 中块内的询问是关于每一块前缀或者后缀的询问.

显然这些询问可以通过预处理答案在 $O(n)$ 的时间复杂度内被解决.

这样子我们只需要在询问的时候进行至多一次 ST 表上的查询操作了.

一些玄学的算法改进

由于 Four russian 算法以 ST 表为基础，而算法竞赛一般没有非常高的时间复杂度要求，所以 Four russian 算法一般都可以被 ST 表代替，在算法竞赛中并不实用．这里提供一种在算法竞赛中更加实用的 Four russian 改进算法．

我们将块大小设为 $\sqrt{n}$，然后预处理出每一块内前缀和后缀的 RMQ，再暴力预处理出任意连续的整块之间的 RMQ，时间复杂度为 $O(n)$．

查询时，对于左右端点不在同一块内的询问，我们可以直接 $O(1)$ 得到左端点所在块的后缀 RMQ，左端点和右端点之间的连续整块 RMQ，和右端点所在块的前缀 RMQ，答案即为三者之间的最值．

而对于左右端点在同一块内的询问，我们可以暴力求出两点之间的 RMQ，时间复杂度为 $O(\sqrt{n})$，但是单个询问的左右端点在同一块内的期望为 $O(\frac{\sqrt{n}}{n})$，所以这种方法的时间复杂度为期望 $O(n)$．

而在算法竞赛中，我们并不用非常担心出题人卡掉这种算法，因为我们可以通过在 $\sqrt{n}$ 的基础上随机微调块大小，很大程度上避免算法在根据特定块大小构造的数据中出现最坏情况．并且如果出题人想要卡掉这种方法，则暴力有可能可以通过．

这是一种期望时间复杂度达到下界，并且代码实现难度和算法常数均较小的算法，因此在算法竞赛中比较实用．

以上做法参考了 P3793 由乃救爷爷[1] 中的题解．

加减 1RMQ

若序列满足相邻两元素相差为 1，在这个序列上做 RMQ 可以成为加减 1RMQ，根究这个特性可以改进 Four Russian 算法，做到 $O(n) \sim O(1)$ 的时间复杂度，$O(n)$ 的空间复杂度．

由于 Four russian 算法的瓶颈在于块内 RMQ 问题，我们重点去讨论块内 RMQ 问题的优化．

由于相邻两个数字的差值为 ± 1，所以在固定左端点数字时 长度不超过 $\log n$ 的右侧序列种类数为 $\sum_{i=1}^{\log n} 2^{i-1}$，而这个式子显然不超过 n．

这启示我们可以预处理所有不超过 n 种情况的 最小值 - 第一个元素 的值．

在预处理的时候我们需要去预处理同一块内相邻两个数字之间的差，并且使用二进制将其表示出来．

在询问的时候我们找到询问区间对应的二进制表示，查表得出答案．

这样子 Four russian 预处理的时间复杂度就被优化到了 $O(n)$．

笛卡尔树在 RMQ 上的应用

不了解笛卡尔树的朋友请移步 笛卡尔树→10.15.6@p2311．

不难发现，原序列上两个点之间的 min/max，等于笛卡尔树上两个点的 LCA 的权值．根据这一点就可以借助 $O(n) \sim O(1)$ 求解树上两个点之间的 LCA 进而求解 RMQ．$O(n) \sim O(1)$ 树上 LCA 在 LCA - 标准 RMQ→11.6.5@p2646 已经有描述，这里不再展开．

总结一下，笛卡尔树在 RMQ 上的应用，就是通过将普通 RMQ 问题转化为 LCA 问题，进而转化为加减 1 RMQ 问题进行求解，时间复杂度为 $O(n) \sim O(1)$．当然由于转化步数较多，$O(n) \sim O(1)$ RMQ 常数较大．

如果数据随机，还可以暴力在笛卡尔树上查找．此时的时间复杂度为期望 $O(n) \sim O(\log n)$，并且实际使用时这种算法的常数往往很小．

例题 Luogu P3865【模板】ST 表[2]

基于状压的线性 RMQ 算法

隐性要求

- 序列的长度 n 满足 $\log_2 n \leq 64$.

前置知识

- Sparse Table→10.11@p2072
- 基本位操作
- 前后缀极值

算法原理

将原序列 $A[1\cdots n]$ 分成每块长度为 $O(\log_2 n)$ 的 $O(\frac{n}{\log_2 n})$ 块.

> 听说令块长为 $1.5\times\log_2 n$ 时常数较小.

记录每块的最大值，并用 ST 表维护块间最大值，复杂度 $O(n)$.

记录块中每个位置的前、后缀最大值 $Pre[1\cdots n], Sub[1\cdots n]$（$Pre[i]$ 即 $A[i]$ 到其所在块的块首的最大值），复杂度 $O(n)$.

若查询的 l, r 在两个不同块上，分别记为第 bl, br 块，则最大值为 $[bl+1, br-1]$ 块间的最大值，以及 $Sub[l]$ 和 $Pre[r]$ 这三个数的较大值.

现在的问题在于若 l, r 在同一块中怎么办.

将 $A[1\cdots r]$ 依次插入单调栈中，记录下标和值，满足值从栈底到栈顶递减，则 $A[l, r]$ 中的最大值为从栈底往上，单调栈中第一个满足其下标 $p \geq l$ 的值.

由于 $A[p]$ 是 $A[l, r]$ 中的最大值，因而在插入 $A[p]$ 时，$A[l\cdots p-1]$ 都被弹出，且在插入 $A[p+1\cdots r]$ 时不可能将 $A[p]$ 弹出.

而如果用 0/1 表示每个数是否在栈中，就可以用整数状压，则 p 为第 l 位后的第一个 1 的位置.

由于块大小为 $O(\log_2 n)$，因而最多不超过 64 位，可以用一个整数存下（即隐性条件的原因）.

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>

constexpr int MAXN = 1e5 + 5;
constexpr int MAXM = 20;

struct RMQ {
  int N, A[MAXN];
  int blockSize;
  int S[MAXN][MAXM], Pow[MAXM], Log[MAXN];
  int Belong[MAXN], Pos[MAXN];
  int Pre[MAXN], Sub[MAXN];
  int F[MAXN];

  void buildST() {
    int cur = 0, id = 1;
```

```
18      Pos[0] = -1;
19      for (int i = 1; i <= N; ++i) {
20        S[id][0] = std::max(S[id][0], A[i]);
21        Belong[i] = id;
22        if (Belong[i - 1] != Belong[i])
23          Pos[i] = 0;
24        else
25          Pos[i] = Pos[i - 1] + 1;
26        if (++cur == blockSize) {
27          cur = 0;
28          ++id;
29        }
30      }
31      if (N % blockSize == 0) --id;
32      Pow[0] = 1;
33      for (int i = 1; i < MAXM; ++i) Pow[i] = Pow[i - 1] * 2;
34      for (int i = 2; i <= id; ++i) Log[i] = Log[i / 2] + 1;
35      for (int i = 1; i <= Log[id]; ++i) {
36        for (int j = 1; j + Pow[i] - 1 <= id; ++j) {
37          S[j][i] = std::max(S[j][i - 1], S[j + Pow[i - 1]][i - 1]);
38        }
39      }
40    }
41
42    void buildSubPre() {
43      for (int i = 1; i <= N; ++i) {
44        if (Belong[i] != Belong[i - 1])
45          Pre[i] = A[i];
46        else
47          Pre[i] = std::max(Pre[i - 1], A[i]);
48      }
49      for (int i = N; i >= 1; --i) {
50        if (Belong[i] != Belong[i + 1])
51          Sub[i] = A[i];
52        else
53          Sub[i] = std::max(Sub[i + 1], A[i]);
54      }
55    }
56
57    void buildBlock() {
58      static int S[MAXN], top;
59      for (int i = 1; i <= N; ++i) {
60        if (Belong[i] != Belong[i - 1])
61          top = 0;
62        else
63          F[i] = F[i - 1];
64        while (top > 0 && A[S[top]] <= A[i]) F[i] &= ~(1 << Pos[S[top--]]);
65        S[++top] = i;
66        F[i] |= (1 << Pos[i]);
67      }
68    }
69
```

```
  void init() {
    for (int i = 1; i <= N; ++i) scanf("%d", &A[i]);
    blockSize = log2(N) * 1.5;
    buildST();
    buildSubPre();
    buildBlock();
  }

  int queryMax(int l, int r) {
    int bl = Belong[l], br = Belong[r];
    if (bl != br) {
      int ans1 = 0;
      if (br - bl > 1) {
        int p = Log[br - bl - 1];
        ans1 = std::max(S[bl + 1][p], S[br - Pow[p]][p]);
      }
      int ans2 = std::max(Sub[l], Pre[r]);
      return std::max(ans1, ans2);
    } else {
      return A[l + __builtin_ctz(F[r] >> Pos[l])];
    }
  }
} R;

int M;

int main() {
  scanf("%d%d", &R.N, &M);
  R.init();
  for (int i = 0, l, r; i < M; ++i) {
    scanf("%d%d", &l, &r);
    printf("%d\n", R.queryMax(l, r));
  }
  return 0;
}
```

习题

[BJOI 2020] 封印[3]：SAM+RMQ

外部参考资料

[1] P3793 由乃救爷爷  [2] Luogu P3865【模板】ST 表  [3] [BJOI 2020] 封印 

14.2 并查集应用

Authors: sshwy

并查集、Kruskal 重构树的思维方式是很类似的，它们都能用于处理与连通性有关的问题．本文通过例题讲解的方式给大家介绍并查集思想的应用．

A

A

有 n 个点，初始时均为孤立点.

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边．设 $L(i,j)$ 表示结点 i 和 j 最早在第 $L(i,j)$ 次操作后连通.

在 m 次操作完后，你要求出 $\sum_{i=1}^{n}\sum_{j=i+1}^{n}L(i,j)$ 的值.

这是基础并查集的应用，并查集记录一下子树的大小．考虑统计每次操作的贡献．如果第 i 次操作 a_i 和 b_i 分属于两个不同子树，就将这两个子树合并，并将两者子树大小的乘积乘上 i 累加到答案里．时间复杂度 $O(n\alpha(n))$.

B

B

有 n 个点，初始时均为孤立点.

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边.

接下来有 q 次询问，第 i 次询问 u_i 和 v_i 最早在第几次操作后连通.

考虑在并查集合并的时候记录「并查集生成树」，也就是说如果第 i 次操作 a_i 和 b_i 分属于两个不同子树，那么把 (a_i,b_i) 这条边纳入生成树中．边权是 i．那么查询就是询问 u 到 v 路径上边权的最大值，可以使用树上倍增或者树链剖分的方法维护．时间复杂度 $O(n\log n)$.

另外一个方法是维护 Kruskal 重构树，其本质与并查集生成树是相同的．复杂度亦相同.

C

C

有 n 个点，初始时均为孤立点.

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边.

接下来有 q 次询问，第 i 次询问第 x_i 个点在第 t_i 次操作后所在连通块的大小.

离线算法：考虑将询问按 t_i 从小到大排序．在加边的过程中使用并查集顺便处理询问即可．时间复杂度 $O(q\log q+(n+q)\alpha(n))$.

在线算法：本题的在线算法只能使用 Kruskal 重构树．Kruskal 重构树与并查集的区别是：第 i 次操作 a_i 和 b_i 分属于两个不同子树，那么 Kruskal 会新建一个结点 u，然后让 a_i 所在子树的根和 b_i 所在子树的根分别连向 u，作为 u 的两个儿子．不妨设 u 的点权是 i．对于初始的 n 个点，点权为 0.

对于询问，我们只需要求出 x_i 在重构树中最大的一个连通块使得连通中的点权最大值不超过 t_i，询问的答案就是这个连通块中点权为 0 的结点个数，即叶子结点个数.

由于我们操作的编号是递增的，因此重构树上父结点的点权总是大于子结点的点权．这意味着我们可以在重构树上从 x_i 到根结点的路径上倍增找到点权最大的不超过 t_i 的结点．这样我们就求出了答案．时间复杂度 $O(n\log n)$.

D

D

给一个长度为 n 的 01 序列 $a_1, ..., a_n$，一开始全是 0，接下来进行 m 次操作：

- 令 $a_x = 1$；
- 求 $a_x, a_{x+1}, ..., a_n$ 中左数第一个为 0 的位置.

建立一个并查集，f_i 表示 $a_i, a_{i+1}, ..., a_n$ 中第一个 0 的位置．初始时 $f_i = i$.

对于一次 $a_x = 1$ 的操作，如果 a_x 原本就等于 1，就不管．否则我们令 $f_x = f_{x+1}$.

时间复杂度 $O(n \log n)$，如果要使用按秩合并的话实现会较为麻烦，不过仍然可行．也就是说时间复杂度或为 $O(n\alpha(n))$.

E

E

给出三个长度为 n 的正整数序列 a，b，c．枚举 $1 \le i \le j \le n$，求 $a_i \cdot b_j \cdot \min_{i \le k \le j} c_k$ 的最大值.

本题同样有许多做法，这里我们重点讲解并查集思路．按权值从大到小考虑 c_k．相当于我们在 k 上加入一个点，然后将 $k-1$ 和 $k+1$ 位置上的点所在的连通块与之合并（如果这两个位置上有点的话）．连通块上记录 a 的最大值和 b 的最大值，即可在合并的时候更新答案．时间复杂度 $O(n \log n)$.

F

F

给出一棵 n 个点的树，接下来有 m 次操作：

- 加一条从 a_i 到 b_i 的边.
- 询问两个点 u_i 和 v_i 之间是否有至少两条边不相交的路径.

询问可以转化为：求 u_i 和 v_i 是否在同一个简单环上．按照双连通分量缩点的想法，每次我们在 a_i 和 b_i 间加一条边，就可以把 a_i 到 b_i 树上路径的点缩到一起．如果两条边 (a_i, b_i) 和 (a_j, b_j) 对应的树上路径有交，那么这两条边就会被缩到一起.

换言之，加边操作可以理解为，将 a_i 到 b_i 树上路径的边覆盖一次．而询问就转化为了：判断 u_i 到 v_i 路径上是否存在未被覆盖的边．如果不存在，那么 u_i 和 v_i 就属于同一个双连通分量，也就属于同一个简单环.

考虑使用并查集维护．给树定根，设 f_i 表示 i 到根的路径中第一个未被覆盖的边．那么每次加边操作，我们就暴力跳并查集．覆盖了一条边后，将这条边对应结点的 f 与父节点合并．这样，每条边至多被覆盖一次，总复杂度 $O(n \log n)$．使用按秩合并的并查集同样可以做到 $O(n\alpha(n))$.

本题的维护方式类似于 D 的树上版本.

G

G

无向图 G 有 n 个点，初始时均为孤立点（即没有边）.

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边.

> 每次操作后，你均需要求出图中桥的个数.
>
> 桥的定义为：对于一条 G 中的边 (x, y)，如果删掉它会使得连通块数量增加，则 (x, y) 被称作桥.
>
> 强制在线.

本题考察对并查集性质的理解．考虑用并查集维护连通情况．对于边双树，考虑维护有根树，设 p_i 表示结点 i 的父亲．也就是不带路径压缩的并查集.

如果第 i 次操作 a_i 和 b_i 属于同一个连通块，那么我们就需要将边双树上 a_i 到 b_i 路径上的点缩起来．这可以用并查集维护．每次缩点，边双连通分量的个数减少 1，最多减少 $n-1$ 次，因此缩点部分的并查集复杂度是 $O(n\alpha(n))$.

为了缩点，我们要先求出 a_i 和 b_i 在边双树上的 LCA．对此我们可以维护一个标记数组．然后从 a_i 和 b_i 开始轮流沿着祖先一个一个往上跳，并标记沿途经过的点．一但跳到了某个之前就被标记过的点，那么这个点就是 a_i 和 b_i 的 LCA．这个算法的复杂度与 a_i 到 b_i 的路径长度是线性相关的，可以接受.

如果 a_i 和 b_i 分属于两个不同连通块，那么我们将这两个连通块合并，并且桥的数量加 1．此时我们需要将两个点所在的边双树连起来，也就是加一条 a_i 到 b_i 的边．因此我们需要将其中一棵树重新定根，然后接到另一棵树上．这里运用启发式合并的思想：我们把结点数更小的重新定根．这样的总复杂度是 $O(n \log n)$ 的.

综上，该算法的总复杂度是 $O(n \log n + m \log n)$ 的.

小结

并查集与 Kruskal 重构树有许多共通点，而并查集的优化（按秩合并）正是启发式合并思想的应用．因此灵活运用并查集可以方便地处理许多与连通性有关的图论问题.

本页面部分内容译自博文 Поиск мостов в режиме онлайн[1] 与其英文翻译版 Finding Bridges Online[2]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

外部参考资料

[1] Поиск мостов в режиме онлайн  [2] Finding Bridges Online 

14.3 括号序列

Authors: sshwy

定义一个合法括号序列（balanced bracket sequence）为仅由 (和) 构成的字符串且：

- 空串 ε 是一个合法括号序列.
- 如果 s 是合法括号序列，那么 (s) 也是合法括号序列.
- 如果 s, t 都是合法括号序列，那么 st 也是合法括号序列.

例如 (())() 是合法括号序列，而)() 不是.

有时候会有多种不同的括号，如 [()]{}．这样的变种括号序列与朴素括号序列有相似的定义.

本文将会介绍与括号序列相关的经典问题.

注：英语中一般称左括号为 opening bracket，而右括号是 closing bracket.

判断是否合法

判断 s 是否为合法括号序列的经典方法是贪心思想．该算法同样适用于变种括号序列.

我们维护一个栈，对于 $i = 1, 2, \ldots, |s|$ 依次考虑：

- 如果 s_i 是右括号且栈非空且栈顶元素是 s_i 对应的左括号，就弹出栈顶元素.
- 若不满足上述条件，则将 s_i 压入栈中.

在遍历整个 s 后，若栈是空的，那么 s 就是合法括号序列，否则就不是. 时间复杂度 $O(n)$.

合法括号序列计数

考虑求出长度为 $2n$ 的合法括号序列 s 的个数 f_n. 不妨枚举与 s_1 匹配的括号的位置，假设是 $2i+2$. 它将整个序列又分成了两个更短的合法括号序列. 因此

$$f_n = \sum_{i=0}^{n-1} f_i f_{n-i-1}$$

这同样是卡特兰数的递推式. 也就是说 $f_n = \frac{1}{n+1}\binom{2n}{n}$.

当然，对于变种合法括号序列的计数，方法是类似的. 假设有 k 种不同类型的括号，那么有 $f'_n = \frac{1}{n+1}\binom{2n}{n}k^n$.

字典序后继

给出合法的括号序列 s，我们要求出按字典序升序排序的长度为 $|s|$ 的所有合法括号序列中，序列 s 的下一个合法括号序列. 在本问题中，我们认为左括号的字典序小于右括号，且不考虑变种括号序列.

我们需要找到一个最大的 i 使得 s_i 是左括号. 然后，将其变成右括号，并将 $s[i+1,|s|]$ 这部分重构一下. 另外，i 必须满足：$s[1,i-1]$ 中左括号的数量 **大于** 右括号的数量.

不妨设当 s_i 变成右括号后，$s[1,i]$ 中左括号比右括号多了 k 个. 那么我们就让 s 的最后 k 个字符变成右括号，而 $s[i+1,|s|-k]$ 则用 $((...(())...))$ 的形式填充即可，因为这样填充的字典序最小.

该算法的时间复杂度是 $O(n)$.

参考实现

```cpp
bool next_balanced_sequence(string& s) {
  int n = s.size();
  int depth = 0;
  for (int i = n - 1; i >= 0; i--) {
    if (s[i] == '(')
      depth--;
    else
      depth++;

    if (s[i] == '(' && depth > 0) {
      depth--;
      int open = (n - i - 1 - depth) / 2;
      int close = n - i - 1 - open;
      string next =
          s.substr(0, i) + ')' + string(open, '(') + string(close, ')');
      s.swap(next);
      return true;
    }
  }
  return false;
}
```

字典序计算

给出合法的括号序列 s，我们要求出它的字典序排名.

考虑求出字典序比 s 小的括号序列 p 的个数.

不妨设 $p_i < s_i$ 且 $\forall 1 \le j < i, p_j = s_i$. 显然 p_i 是左括号而 s_i 是右括号. 枚举 i（满足 s_i 为右括号），假设 $p[1,i]$ 中左括号比右括号多 k 个，那么相当于我们要统计长度为 $|s|-i$ 且存在 k 个未匹配的右括号且不存在未匹配的左括号的括号序列的个数.

不妨设 $f(i,j)$ 表示长度为 i 且存在 j 个未匹配的右括号且不存在未匹配的左括号的括号序列的个数.

通过枚举括号序列第一个字符是什么，可以得到 f 的转移：$f(i,j) = f(i-1,j-1) + f(i-1,j+1)$. 初始时 $f(0,0) = 1$. 其实 f 是 OEIS - A053121[1].

这样我们就可以 $O(|s|^2)$ 计算字典序了.

对于变种括号序列，方法是类似的，只不过我们需要对每个 s_i 考虑比它小的那些字符进行计算（在上述算法中因为不存在比左括号小的字符，所以我们只考虑了 s_i 为右括号的情况）.

另外，利用 f 数组，我们同样可以求出字典序排名为 k 的合法括号序列.

本页面主要译自博文 http://e-maxx.ru/algo/bracket_sequences[2] 与其英文翻译版 Balanced bracket sequences[3]．其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0．

外部参考资料

[1] OEIS - A053121

[2] http://e-maxx.ru/algo/bracket_sequences

[3] Balanced bracket sequences

14.4 线段树与离线询问

Authors: xiezheyuan

线段树与离线询问结合的问题在 OI 领域也有出现. 这种技巧又被称为线段树分治.

假如你需要维护一些信息，这些信息会在某一个时间段内出现，要求在离线的前提下回答某一个时刻的信息并，则可以考虑使用线段树分治的技巧.

实际上线段树分治常有以下用途：

1. 用原本不支持删除但是支持撤销的数据结构来模拟删除操作. 如朴素的并查集无法高效支持删边操作.
2. 不同属性的数据分别计算. 如需要求出除了某一种颜色外，其他颜色数据的答案.

如果大家现在不明白没有关系，这两种用途我们都会在例题中阐述.

过程

首先我们建立一个线段树来维护时刻，每一个节点维护一个 `vector` 来存储位于这一段时刻的信息.

插入一个信息到线段树中和普通线段树的区间修改是类似的.

然后我们考虑如何处理每一个时间段的信息并. 考虑从根节点开始分治，维护当前的信息并，然后每到一个节点的时候将这个节点的所有信息进行合并. 回溯时撤销这一部分的贡献. 最后到达叶子节点时的信息并就是对应的答案.

如果更改信息的时间复杂度为 $O(T(n))$，可以通过设置一个栈保留更改，以 $O(T(n))$ 的时间复杂度撤销. 撤销不维持均摊复杂度.

整个分治流程的总时间复杂度是 $O(n\log n(T(n)+M(n)))$ 的，其中 $O(M(n))$ 为合并信息的时间复杂度，空间复杂度为 $O(n\log n)$.

实现

```
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)

vector<Object> tree[N << 2];  // 线段树

void update(int ql, int qr, Object obj, int i, int l, int r) {  // 插入
  if (ql <= l && r <= qr) {
    tree[i].push_back(obj);
    return;
  }
  if (ql <= mid) update(ql, qr, obj, ls, l, mid);
  if (qr > mid) update(ql, qr, obj, rs, mid + 1, r);
}

stack<Object> sta;  // 用于撤销的栈
Object now;         // 当前的信息并
Object ans[N];      // 答案

void solve(int i, int l, int r) {
  auto lvl = sta.size();  // 记录一下应当撤销到第几个
  for (Object x : tree[i]) sta.push(now), now = Merge(now, x);  // 合并信息
  if (l == r)
    ans[i] = now;  // 记录一下答案
  else
    solve(ls, l, mid), solve(rs, mid + 1, r);  // 分治
  while (sta.size() != lvl) {                  // 撤销信息
    now = sta.top();
    sta.pop();
  }
}
```

例题

luogu P5787 二分图/【模板】线段树分治[1]

你需要维护一个 n 个点 m 条边的无向图．第 i 条边为 (x_i,y_i)，出现的时刻为 $[l_i,r_i)$，其余时刻消失.

对于每一个时刻，若此时该图为二分图，输出 `Yes`，否则输出 `No`.

解题思路

使用种类并查集来维护一个图是否是二分图，然后就可以套用线段树分治了.

注意可撤销的并查集不能路径压缩，只能按秩合并.

参考代码

```cpp
#include <iostream>
#include <stack>
#include <vector>
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

int n, m, k;
constexpr int N = 2e5 + 5;
int fa[N << 1], siz[N << 1];

struct UndoObject {
  int pos, val;

  UndoObject(int p, int v) { pos = p, val = v; }
};

stack<UndoObject> undo_sz, undo_fa;

int find(int x) {
  if (fa[x] == x)
    return x;
  else
    return find(fa[x]);
}

void merge(int u, int v) {
  int x = find(u), y = find(v);
  if (x == y) return;
  if (siz[x] < siz[y]) {
    swap(x, y);
  }
  undo_sz.push(UndoObject(x, siz[x]));
  siz[x] += siz[y];
  undo_fa.push(UndoObject(y, fa[y]));
  fa[y] = x;
}

void undo() {
  fa[undo_fa.top().pos] = undo_fa.top().val;
  undo_fa.pop();
  siz[undo_sz.top().pos] = undo_sz.top().val;
  undo_sz.pop();
}

vector<int> tree[N << 2];

void update(int ql, int qr, int v, int i, int l, int r) {
```

```
  if (ql <= l && r <= qr) {
    tree[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

struct edge {
  int u, v;
} g[N];

vector<bool> ret;

void solve(int i, int l, int r) {
  auto level = undo_fa.size();
  bool ans = true;
  for (int u : tree[i]) {
    int a = find(g[u].u);
    int b = find(g[u].v);
    if (a == b) {
      for (int k = l; k <= r; k++) {
        ret.push_back(false);
      }
      ans = false;
      break;
    }
    merge(g[u].u, g[u].v + n);
    merge(g[u].v, g[u].u + n);
  }
  if (ans) {
    if (l == r) {
      ret.push_back(true);
    } else {
      solve(ls, l, mid);
      solve(rs, mid + 1, r);
    }
  }
  while (undo_fa.size() > level) {
    undo();
  }
}

signed main() {
  cin >> n >> m >> k;
  for (int i = 1; i <= (n << 1); i++) {
    fa[i] = i;
    siz[i] = 1;
  }
  for (int i = 1, l, r; i <= m; i++) {
    cin >> g[i].u >> g[i].v >> l >> r;
```

```
    update(l + 1, r, i, 1, 1, k);
  }
  solve(1, 1, k);
  for (bool i : ret) {
    cout << (i ? "Yes" : "No") << '\n';
  }
  return 0;
}
```

颜色限制 restriction

一个 n 点 m 边的无向图，有 k 种颜色编号为 $0 \sim k-1$，每条边有一种颜色.

对于每种颜色，请判断假如删去所有这种颜色的边，得到的图是否连通？是否是一棵树？

输出满足删去后图连通的颜色数和删去后图是树的颜色数.

解题思路

对于每一种颜色，建立一个时间，在这个时间内没有这个颜色的边，其他边都有. 用一个并查集维护一下即可.

参考代码

```
#include <bitset>
#include <iostream>
#include <stack>
#include <vector>
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

int n, m, k;
constexpr int N = 1e5 + 5;

struct edge {
  int u, v, c;
} g[N];

vector<int> t[N << 2];
int fa[N], siz[N], cnt[N];

void update(int ql, int qr, int v, int i, int l, int r) {
  if (ql > qr) return;
  if (ql <= l && r <= qr) {
    t[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
```

```
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

stack<pair<int, int>> fas, sizs;

int find(int x) { return fa[x] == x ? x : find(fa[x]); }

void merge(int u, int v) {
  int fu = find(u), fv = find(v);
  if (fu == fv) return;
  if (siz[fu] < siz[fv]) swap(fu, fv);
  fas.push(make_pair(fv, fa[fv]));
  fa[fv] = fu;
  sizs.push(make_pair(fu, siz[fu]));
  siz[fu] += siz[fv];
}

bitset<N> ans;

void solve(int i, int l, int r) {
  unsigned lvl = fas.size();
  for (int eg : t[i]) merge(g[eg].u, g[eg].v);
  if (l == r)
    ans[l] = (siz[find(1)] == n);
  else
    solve(ls, l, mid), solve(rs, mid + 1, r);
  while (fas.size() != lvl) {
    auto p1 = fas.top(), p2 = sizs.top();
    fas.pop(), sizs.pop();
    fa[p1.first] = p1.second;
    siz[p2.first] = p2.second;
  }
}

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cin >> n >> m >> k;
  for (int i = 1; i <= m; i++) {
    cin >> g[i].u >> g[i].v >> g[i].c;
    g[i].c++;
    update(1, g[i].c - 1, i, 1, 1, k);
    update(g[i].c + 1, k, i, 1, 1, k);
    cnt[g[i].c]++;
  }
  for (int i = 1; i <= n; i++) {
    fa[i] = i;
    siz[i] = 1;
  }
  solve(1, 1, k);
  int ans1 = 0, ans2 = 0;
```

```
  for (int i = 1; i <= k; i++) {
    ans1 += ans[i];
    ans2 += (ans[i] && (m - cnt[i]) == (n - 1));
  }
  cout << ans1 << ' ' << ans2;
  return 0;
}
```

luogu P4219 [BJOI2014] 大融合[2]

需要维护一个 n 个点的森林，初始时是散点.

有 q 个操作，支持:

- `A x y` 连边 (x, y).
- `Q x y` 输出经过边 (x, y) 的路径数.

允许离线.

解题思路

考虑允许离线，因此可以想到线段树分治.

然后考虑如何支持 Q 操作. 如果不存在 (x, y) 这条边，答案就是 x 所在连通块大小乘上 y 所在连通块大小. 可以用并查集维护.

因此你可以将 Q 拆成三个时间 $k-1, k, k+1$. 其中 $k-1$ 是这条边的终止时刻，$k+1$ 是这条边的起始时刻. 这样 k 就没有这条边，正好回答询问.

参考代码

```
#include <iostream>
#include <map>
#include <stack>
#include <vector>
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

int n, m;
constexpr int N = 1e5 + 5;
int fa[N], siz[N], tim;

struct UndoObject {
  int pos, val;

  UndoObject(int p, int v) { pos = p, val = v; }
};

stack<UndoObject> undo_sz, undo_fa;
```

```
21
22 int find(int x) {
23   if (fa[x] == x)
24     return x;
25   else
26     return find(fa[x]);
27 }
28
29 void merge(int u, int v) {
30   int x = find(u), y = find(v);
31   if (x == y) return;
32   if (siz[x] < siz[y]) {
33     swap(x, y);
34   }
35   undo_sz.push(UndoObject(x, siz[x]));
36   siz[x] += siz[y];
37   undo_fa.push(UndoObject(y, fa[y]));
38   fa[y] = x;
39 }
40
41 void undo() {
42   fa[undo_fa.top().pos] = undo_fa.top().val;
43   undo_fa.pop();
44   siz[undo_sz.top().pos] = undo_sz.top().val;
45   undo_sz.pop();
46 }
47
48 vector<pair<int, int>> tree[N << 4];
49
50 void update(int ql, int qr, pair<int, int> v, int i, int l, int r) {
51   if (ql <= l && r <= qr) {
52     tree[i].push_back(v);
53     return;
54   }
55   if (ql <= mid) update(ql, qr, v, ls, l, mid);
56   if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
57 }
58
59 map<pair<int, int>, int> tims;
60
61 struct ops {
62   int l, r;
63   pair<int, int> v;
64 } opp[N << 3];
65
66 int opcnt;
67 map<int, int> queries;
68 map<int, pair<int, int>> querylr;
69 int ans[N << 3];
70
71 void solve(int i, int l, int r) {
```

```
  auto level = undo_fa.size();
  for (auto u : tree[i]) {
    merge(u.first, u.second);
  }
  if (l == r) {
    if (queries[l]) {
      int x = querylr[l].first;
      int y = querylr[l].second;
      // cout<<siz[find(x)]<<' '<<siz[find(y)]<<'\n';
      ans[l] = siz[find(x)] * siz[find(y)];
    }
  } else {
    solve(ls, l, mid);
    solve(rs, mid + 1, r);
  }
  while (undo_fa.size() > level) {
    undo();
  }
}

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    fa[i] = i;
    siz[i] = 1;
  }
  while (m--) {
    char op;
    int x, y;
    cin >> op >> x >> y;
    if (op == 'A') {
      tims[make_pair(x, y)] = ++tim;
    } else {
      pair<int, int> xy = make_pair(x, y);
      opp[++opcnt] = {tims[xy], (++tim), xy};
      queries[++tim] = 1;
      // cout<<tim<<'\n';
      querylr[tim] = xy;
      tims[xy] = ++tim;
    }
  }
  int tm = ++tim;
  for (auto i : tims) {
    opp[++opcnt] = {tims[i.first], tm, i.first};
  }
  for (int i = 1; i <= opcnt; i++) {
    // cout<<opp[i].l<<' '<<opp[i].r<<' '<<opp[i].v.first<<'
    // '<<opp[i].v.second<<'\n';
    update(opp[i].l, opp[i].r, opp[i].v, 1, 1, tim);
```

```
  }
  // cout<<tim<<'\n';
  solve(1, 1, tim);
  for (int i = 1; i <= tim; i++) {
    if (queries[i]) {
      cout << ans[i] << "\n";
    }
  }
  return 0;
}
```

luogu P2056 [ZJOI2007] 捉迷藏[3]

出一个 n 个点的树，每个点有黑白两种颜色．初始时每个点都是黑色的．q 次操作，支持：

- `C x` 将第 x 个点的颜色反转．
- `G` 询问树上两个黑色点的最远距离．特别地，若不存在黑色点，输出 -1．

允许离线．

解题思路

首先考虑如何维护树上点集的直径，有下面的推论：

> 对于一个集合 S 和只有一个点的集合 $\{P\}$．若集合 S 的直径为 (U,V)．则点集 $S\cap\{P\}$ 的直径只可能为 $(U,V),(U,P)$ 或 (V,P)．

然后考虑解决原问题．我们可以考虑维护黑色点集，维护每一个点在黑色点集中的若干个时间段（具体你开一个桶记录一下上一次进入黑色点集的时刻即可）．

然后就自然地想到离线，将所有时间刻插入到线段树中．然后在线段树上分治，每次线段树上的点会记录当前时间段点集新增的点，新增点可以使用上面的推论，找到新点集直径的两个端点．

撤销是平凡的，开一个栈记录一下直径端点的变化即可．

参考代码

```
#include <algorithm>
#include <bitset>
#include <iostream>
#include <stack>
#include <vector>
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

constexpr int N = 5e5 + 5, M = 5e5 + 5;

int siz[N], dep[N], father[N], top[N], son[N];
```

```cpp
int n, q;

struct edge {
  int nxt, to;
} g[N << 1];

int head[N], ec;

void add(int u, int v) {
  g[++ec].nxt = head[u];
  g[ec].to = v;
  head[u] = ec;
}

void dfs1(int u, int fa) {
  dep[u] = dep[fa] + 1;
  siz[u] = 1;
  father[u] = fa;
  for (int i = head[u]; i; i = g[i].nxt) {
    int v = g[i].to;
    if (v == fa) continue;
    dfs1(v, u);
    siz[u] += siz[v];
    if (siz[son[u]] < siz[v]) son[u] = v;
  }
}

void dfs2(int u, int fa) {
  if (son[u]) {
    top[son[u]] = top[u];
    dfs2(son[u], u);
  }
  for (int i = head[u]; i; i = g[i].nxt) {
    int v = g[i].to;
    if (v == fa || v == son[u]) continue;
    top[v] = v;
    dfs2(v, u);
  }
}

int lca(int x, int y) {
  while (top[x] != top[y]) {
    if (dep[top[x]] < dep[top[y]]) swap(x, y);
    x = father[top[x]];
  }
  return dep[x] < dep[y] ? x : y;
}

int dis(int x, int y) { return dep[x] + dep[y] - (dep[lca(x, y)] <<
1); }

```

```cpp
vector<int> t[N << 2];

void update(int ql, int qr, int v, int i, int l, int r) {
  if (ql <= l && r <= qr) {
    t[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

stack<pair<int, int>> stk;
int u, v;
int ans[M];

void solve(int i, int l, int r) {
  auto lvl = stk.size();
  for (int x : t[i]) {
    stk.push(make_pair(u, v));
    if (!u && !v)
      u = x, v = x;
    else {
      vector<int> vct = {dis(u, v), dis(u, x), dis(v, x)};
      sort(vct.begin(), vct.end(), greater<int>());
      if (vct[0] == dis(u, x))
        v = x;
      else if (vct[0] == dis(x, v))
        u = x;
    }
  }
  if (l == r)
    ans[l] = (!u || !v) ? -1 : dis(u, v);
  else
    solve(ls, l, mid), solve(rs, mid + 1, r);
  while (stk.size() != lvl) {
    auto top = stk.top();
    u = top.first, v = top.second;
    stk.pop();
  }
}

int lst[N];
bitset<N> col;
bitset<M> haveq;

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v;
```

```
    cin >> u >> v;
    add(u, v);
    add(v, u);
  }
  top[1] = 1;
  dfs1(1, 0);
  dfs2(1, 0);
  for (int i = 1; i <= n; i++) lst[i] = 1;
  cin >> q;
  for (int i = 2; i <= (q + 1); i++) {
    char c;
    int x;
    cin >> c;
    if (c == 'C') {
      cin >> x;
      if (!col[x]) {
        col[x] = 1;
        update(lst[x], i, x, 1, 1, q + 2);
      } else
        col[x] = 0, lst[x] = i;
    } else
      haveq[i] = 1;
  }
  for (int i = 1; i <= n; i++) {
    if (!col[i]) update(lst[i], q + 2, i, 1, 1, q + 2);
  }
  solve(1, 1, q + 2);
  for (int i = 1; i <= (q + 2); i++) {
    if (haveq[i]) cout << ans[i] << '\n';
  }
  return 0;
}

```

习题

- CF601E A Museum Robbery[4] 线段树分治 + 背包 dp.
- CF19E Fairy[5] 线段树分治 + 种类并查集.
- luogu P5227 [AHOI2013] 连通图[6] 线段树分治 + 并查集.
- luogu P4319 变化的道路[7] 线段树分治 + Link Cut Tree 维护最小生成树.
- luogu P3733 [HAOI2017] 八纵八横[8] 线段树分治 + 线性基.

本页面部分内容参考自博文 Deleting from a data structure[9]，版权协议为 CC-BY-SA 4.0.

外部参考资料

[1] luogu P5787 二分图/【模板】线段树分治

[2] luogu P4219 [BJOI2014] 大融合

[3] luogu P2056 [ZJOI2007] 捉迷藏

[4] CF601E A Museum Robbery

[5] CF19E Fairy

[6] luogu P5227 [AHOI2013] 连通图

[7] luogu P4319 变化的道路 

[8] luogu P3733 [HAOI2017] 八纵八横 

[9] Deleting from a data structure 

https://oi-wiki.org