普通莫队算法

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。

实现

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。

模板

```
inline 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：令每一块中 L 的最大值为 $\mathbf{max}_1,\mathbf{max}_2,\mathbf{max}_3,\cdots,\mathbf{max}_{\lceil\sqrt{n}\rceil}$。

由第一次排序可知，$\mathbf{max}_1\le\mathbf{max}_2\le\cdots\le\mathbf{max}_{\lceil\sqrt{n}\rceil}$。

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 $\mathbf{max}_{i-1}$ 修改至 $\mathbf{max}_i$ 或由 $\mathbf{max}_i$ 修改至 $\mathbf{max}_{i-1}$。

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 $O(\mathbf{max}_i-\mathbf{max}_{i-1})$ 的，所以在同一块中时间复杂度为 $O(\sqrt{n}\cdot(\mathbf{max}_i-\mathbf{max}_{i-1}))$。

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

$$
\begin{aligned}
&O(\sqrt{n}(\mathbf{max}_1-1)+\sqrt{n}(\mathbf{max}_2-\mathbf{max}_1)+\sqrt{n}(\mathbf{max}_3-\mathbf{max}_2)+\\
=&O(\sqrt{n}\cdot(\mathbf{max}_1-1+\mathbf{max}_2-\mathbf{max}_1+\mathbf{max}_3-\mathbf{max}_2+\cdots+\mathbf{max}_{\lceil\sqrt{n}\rceil-1}))\\
=&O(\sqrt{n}\cdot(\mathbf{max}_{\lceil\sqrt{n}\rceil-1}))
\end{aligned}
$$

（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 $O(n)$。

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 [https://www.luogu.com.cn/problem/P1494]

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。

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 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），表示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 $C_{col[k]+1}^2-C_{col[k]}^2$，如果是缩短就是 ans 减去 $C_{col[k]}^2-C_{col[k]-1}^2$。

而这个询问的答案就是 $\dfrac{ans}{C_{r-l+1}^2}$。

这里有个优化：$C_a^2=\dfrac{a(a-1)}{2}$。

所以

$$C_{a+1}^2-C_a^2=\frac{(a+1)a}{2}-\frac{a(a-1)}{2}=\frac{a}{2}\cdot(a+1-a+1)=\frac{a}{2}\cdot 2=a$$

。

所以 $C_{col[k]+1}^2 - C_{col[k]}^2 = col[k]$ 。

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 $l \leq r$ 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 $l \leq r+1$；执行完前两步后，$l \leq l' \leq r' \leq r$ 一定成立，再执行后两步只会把区间缩小到 $[l', r']$，依然有 $l \leq r+1$，因此这样写是正确的。

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;
const 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 {
    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() {
  scanf("%d%d", &n, &m);
  maxn = sqrt(n);
  for (int i = 1; i <= n; i++) scanf("%d", &c[i]);
  for (int i = 0; i < m; i++) scanf("%d%d", &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++) {
    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;
    printf("%lld/%lld\n", ans1[i], ans2[i]);
  }
```

```
    return 0;
  }
```

普通莫队的优化

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。

排序代码：

压行

```
// 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

```
struct node {
  int l, r, id;
  bool operator<(const node &x) const {
    if (l / unit != x.l / unit) return l < x.l;
    if ((l / unit) & 1)
      return r <
             x.r;  // 注意这里和下面一行不能写小于（大于）等于，否则
会出错（详见下面的小细节）
    return r > x.r;
  }
};
```

Warning

小细节：如果使用 sort 比较两个函数，不能出现 $a < b$ 和 $b < a$ 同时为真的情况，否则会运行错误。

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 [https://blog.sengxian.com/algorithms/mo-s-algorithm]