第6章 数据结构提高

在第 2 章的数据结构基础中，我们已经接触到了不少常用的数据结构，例如栈、队列、优先队列等等。本章在第 2 章的基础之上，详细介绍了算法竞赛中涉及到的一些更为复杂的，应用广泛的内容，例如线段树、树状数组、分块、并查集等，这些对于很多题目的优化起着举足轻重的作用。

6.1 线段树

线段树是一种基于分治思想的二叉树结构，常用于维护区间信息。

想象这样的问题：给定一个n个元素的数组$a_1, a_2, \ldots, a_n$。设计一个数据结构，使其支持以下两种操作。

1） 单点修改：将a_x的值修改为v。

2） 区间查询：计算$\min\{a_1, a_2, \ldots, a_n\}$。

如果使用朴素的方法，对于每次修改，都需要进行一次复杂度为$O(n)$的区间查询，时间复杂度难以承受。使用线段树就可以很好的解决这类问题。

6.1.1 基础线段树

线段树将每个长度不为1的区间划分成左右两个区间递归求解，把整个区间划分为一个树形结构，通过合并左右两区间信息来求得该区间的信息。这种数据结构可以方便的进行大部分的区间操作。从区间上来看，线段树结构如图 6.1 所示：

1） 线段树的每个节点都代表一个区间。

2） 线段树具有唯一的根节点，代表的区间是整个统计范围$[1, N]$。

3） 线段树的每个叶节点都代表一个长度为1的元区间$[x, x]$。

4） 对于每个内部节点$[l, r]$， 它的左子节点是$[l, mid]$， 右子节点是$[mid + 1, r]$，其中$mid = (l + r)/2$ (向下取整)。

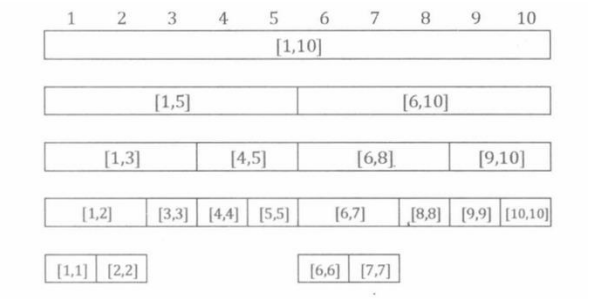

图 6.1

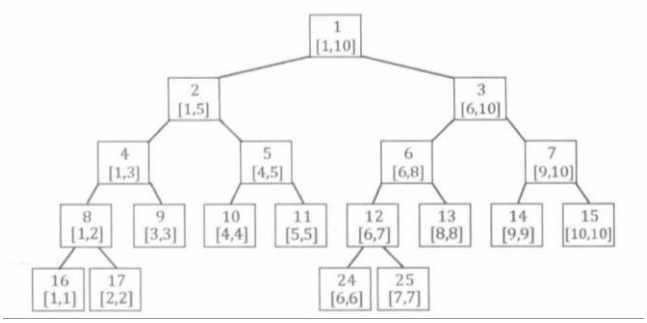

图 6.2

图 6.2 展示了一颗线段树，可以发现这是一个二叉树结构，采用与二叉树相似的编号方法：

1） 根节点编号为1。

2） 编号为x的节点的左子节点编号为$x*2$，右子节点编号为$x*2+1$。

这样一来，就能简单地使用一个$struct$数组来保存线段树。所以保存线段树的数组长度要不小于$4*N$才能保证不会越界。

6.1.1.1 建树

以区间最小值为例，将数组$a_1, a_2, \ldots, a_n$转化成一颗线段树，结点1保存$[1,n]$区间的最小值，结点2保存$[1,\frac{n+1}{2}]$区间的最小值，以此类推，每个叶子结点$[i,i]$保存a_i的值。线段树的二叉树结构可以方便的从下往上传递区间信息。

【代码】

```
1. struct t
2. {
3.   int l, r;
4.   int s;
5. } tree[4 * N];
6.
7. void build(int x, int l, int r)
8. {
9.   tree[x].l = l;
10.  tree[x].r = r;
11.  if (l == r)
12.  {
13.    tree[x].s = a[l];
14.    return;
15.  }
16.  int mid = (l + r) >> 1;
17.  build(x * 2, l, mid);
```

```
18.   build(x * 2 + 1, mid + 1, r);
19.   tree[x].s = min(tree[x * 2].s, tree[x * 2 + 1].s);
20. }
```

6.1.1.2 单点修改

单点修改命令：将a_x的值修改为v。

在线段树中，根节点(编号为1的节点)是执行各种指令的入口。我们需要从根节点出发，递归找到代表区间$[x,x]$的叶节点，然后从下往上更新$[x,x]$以及它的所有祖先节点上保存的信息，时间复杂度为$O(logN)$。

【代码】

```
1. void change(int x, int pos, int val)
2. {
3.     if (tree[x].l == tree[x].r)
4.     {
5.         tree[x].s = val;
6.         return;
7.     }
8.     int mid = (tree[x].l + tree[x].r) >> 1;
9.     if (pos <= mid)
10.        change(x * 2, pos, val);
11.    else
12.        change(x * 2 + 1, pos, val);
13.
14.    tree[x].s = min(tree[x * 2].s, tree[x * 2 + 1].s);
15. }
```

6.1.1.3 区间查询

区间查询命令：以区间最小值为例，查询序列A在区间$[l,r]$上的最小值，即$min_{l\le i\le r}\{A[i]\}$。

我们只需要从根节点开始，递归执行以下过程：

1） 若$[l,r]$完全覆盖了当前节点代表的区间，则立即回溯，并且该节点存储的最小值为候选答案。

2） 若左子节点与$[l,r]$有重叠部分，则递归访问左子节点。

3） 若右子节点与$[l,r]$有重叠部分，则递归访问右子节点。

【代码】

```
1. int ask(int x, int l, int r)
2. {
3.     if (l <= tree[x].l && tree[x].r <= r)
```

```
4.         return tree[x].s;
5.     int mid = (tree[x].l + tree[x].r) >> 1;
6.     int ans = INF;
7.     if (l <= mid)
8.         ans = min(ans, ask(x * 2, l, r));
9.     if (r > mid)
10.        ans = min(ans,ask(x * 2 + 1,l,r);
11.    return sum;
12. }
```

6.1.2 带标记的线段树

与单点修改不同的是，在线段树的“区间查询”指令中，每当遇到被询问区间$[l,r]$完全覆盖的节点时，可以立即把该节点上存储的信息作为候选答案返回。显然被询问区间$[l,r]$在线段树。上会被分成$O(logN)$个小区间(节点)，从而在$O(logN)$的时间内求出答案。不过，在“区间修改”指令中，如果某个节点被修改区间$[l,r]$完全覆盖，那么以该节点为根的整棵子树中的所有节点存储的信息都会发生变化，若逐一进行更新，将使得一次区间修改指令的时间复杂度增加到$O(N)$，这是我们不能接受的。

可以想到，如果在一次修改指令中发现节点p代表的区间$[p_l,p_r]$被修改区间$[l,r]$完全覆盖，并且逐一更新了子树p中的所有节点,但是在之后的查询指令中却根本没有用到$[l,r]$的子区间作为候选答案，那么更新p的整棵子树就是徒劳的。也就是说，我们在执行修改指令时，同样可以在$l \le p_i \le p_r \le r$的情况下立即返回，只不过在回溯之前向节点p增加一个标记，标识“该节点曾经被修改，但其子节点尚未被更新”。

如果在后续的指令中，需要从节点p向下递归，我们再检查p是否具有标记。若有标记，就根据标记信息更新p的两个子节点,同时为p的两个子节点增加标记，然后清除p的标记。除了在修改指令中直接划分成的$O(logN)$个节点之外，对任意节点的修改都延迟到“在后续操作中递归进入它的父节点时”再执行。这样一来，每条查询或修改指令的时间复杂度都降低到了$O(logN)$。这些标记被称为“延迟标记”。延迟标记提供了线段树中从上往下传递信息的方式”。这种“延迟”也是设计算法与解决问题的一个重要思路。

【代码】

以下是区间修改的代码，其中$lazy()$函数代表延迟标记（懒惰标记）。

```
1. void change(int f, int l, int r, int val)
2. {
3.     if (l <= tree[x].l && r >= tree[x].r)
4.     {
5.         tree[x].s += (long long)val * (tree[x].r - tree[x].l + 1);
```

```
        tree[x].f += val;
        return;
    }
    lazy(x);
    int mid = (tree[x].l + tree[x].r) >> 1;
    if (l <= mid)
        change(x * 2, l, r, val);
    if (r > mid)
        change(x * 2 + 1, l, r, val);
    tree[x].s = tree[x * 2].s + tree[x * 2 + 1].s;
}
```

其中懒惰标记的代码如下：

```
void lazy(int x)
{
    if (tree[x].f)
    {
        tree[x * 2].s += tree[x].f * (tree[x * 2].r - tree[x * 2].l + 1);
        tree[x * 2 + 1].s += tree[x].f * (tree[x * 2 + 1].r - tree[x * 2 + 1].l + 1);
        tree[x * 2].f += tree[x].f;
        tree[x * 2 + 1].f += tree[x].f;
        tree[x].f = 0;
    }
}
```

相对应的，区间查询的代码也有所变化：

```
long long ask(int x, int l, int r)
{
    if (l <= tree[x].l && r >= tree[x].r)
        return tree[x].s;
    lazy(x);
    int mid = (tree[x].l + tree[x].r) >> 1;
    long long ans = 0;
    if (l <= mid)
        ans += ask(x * 2, l, r);
    if (r > mid)
        ans += ask(x * 2 + 1, l, r);
    return ans;
}
```

例题 【模板】线段树 1（luoguP3372）

如题，已知一个数列，你需要进行下面两种操作：

1） 将某区间每一个数加上 k。

2） 求出某区间每一个数的和。

【输入格式】

第一行包含两个整数 n,m，分别表示该数列数字的个数和操作的总个数。

第二行包含n个用空格分隔的整数，其中第i个数字表示数列第i项的初始值。

接下来m行每行包含 3 或4个整数，表示一个操作，具体如下：

$1\ x\ y\ k$：将区间 $[x,y]$ 内每个数加上 k。

$2\ x\ y$：输出区间 $[x,y]$ 内每个数的和。

对于 30% 的数据：$n \leq 8$，$m \leq 10$。

对于 70% 的数据：$n \leq 10^3$，$m \leq 10^4$。

对于100% 的数据：$1 \leq n,m \leq 10^5$。

保证任意时刻数列中所有元素的绝对值之和$\leq 10^{18}$。

【输出格式】

输出包含若干行整数，即为所有操作 2 的结果。

【分析】

如题，线段树模版题，要求实现区间修改和区间查询操作。

【代码】

```
const int N = 1e5 + 10;
typedef long long ll;

struct t
{
    int l, r;
    ll s;
    ll f;
} tree[4 * N];
ll a[N];
int n, m, opt;

void build(int x, int l, int r)
{
    tree[x].l = l;
    tree[x].r = r;
    if (l == r)
    {
        tree[x].s = a[l];
        return;
    }
    int mid = (l + r) >> 1;
```

```
23.     build(x * 2, l, mid);
24.     build(x * 2 + 1, mid + 1, r);
25.     tree[x].s = tree[x * 2].s + tree[x * 2 + 1].s;
26. }
27.
28. void lazy(int x)
29. {
30.     if (tree[x].f)
31.     {
32.         tree[x * 2].s += tree[x].f * (tree[x * 2].r - tree[x * 2].l + 1);
33.         tree[x * 2 + 1].s += tree[x].f * (tree[x * 2 + 1].r - tree[x * 2 + 1].l + 1);
34.         tree[x * 2].f += tree[x].f;
35.         tree[x * 2 + 1].f += tree[x].f;
36.         tree[x].f = 0;
37.     }
38. }
39.
40. void change(int x, int l, int r, ll val)
41. {
42.     if (l <= tree[x].l && r >= tree[x].r)
43.     {
44.         tree[x].s += val * (tree[x].r - tree[x].l + 1);
45.         tree[x].f += val;
46.         return;
47.     }
48.     lazy(x);
49.     int mid = (tree[x].l + tree[x].r) >> 1;
50.     if (l <= mid)
51.         change(x * 2, l, r, val);
52.     if (r > mid)
53.         change(x * 2 + 1, l, r, val);
54.     tree[x].s = tree[x * 2].s + tree[x * 2 + 1].s;
55. }
56.
57. ll ask(int x, int l, int r)
58. {
59.     if (l <= tree[x].l && r >= tree[x].r)
60.         return tree[x].s;
61.     lazy(x);
62.     int mid = (tree[x].l + tree[x].r) >> 1;
63.     ll ans = 0;
64.     if (l <= mid)
65.         ans += ask(x * 2, l, r);
66.     if (r > mid)
```

```
67.         ans += ask(x * 2 + 1, l, r);
68.     return ans;
69. }
70.
71. int main()
72. {
73.     scanf("%d%d", &n, &m);
74.     for (int i = 1; i <= n; ++i)
75.         scanf("%lld", &a[i]);
76.     build(1, 1, n);
77.     while (m--)
78.     {
79.         int x, y;
80.         ll val;
81.         scanf("%d%d%d", &opt, &x, &y);
82.         if (opt == 1)
83.         {
84.             scanf("%lld", &val);
85.             change(1, x, y, val);
86.         }
87.         else
88.         {
89.             printf("%lld\n", ask(1, x, y));
90.         }
91.     }
92.     return 0;
93. }
```

例题　求最大值（牛客 NC14402）

给出一个序列，你的任务是求每次操作之后序列中$\frac{(a[j]-a[i])}{(j-i)}(1 <= i < j <= n)$的最大值。

操作次数有Q次，每次操作需要将位置p处的数字变成y。

【输入格式】

本题包含多组输入，每组输入第一行一个数字n，表示序列的长度。

然后接下来一行输入n个数，表示原先序列的样子。

再接下来一行一个数字Q，表示需要进行的操作的次数。

最后Q行每行两个元素p，y，表示本操作需要将位置p处的数字变成y。

数据范围：

$3 <= n <= 2 * 10^5$，$1 <= Q <= 2 * 10^5$，$-10^9 <= a[i] <= 10^9$。

【输出格式】

每组数据输出 Q 行，每行输出一个浮点数，保留两位小数，表示所求的最大值。

【分析】

首先要将问题转化，因为我们无法直接维护这个最大值。如果把问题放到一个二维坐标系下，数组下标是横坐标，那么原数组对应的值是纵坐标，本题就转化为求两点之间的最大斜率。显然的，这个最大斜率只可能出现在相邻的两个点之间。故我们只要维护出差分数组的区间最大值即可。

题目要求的修改是单点修改。那么一个点改变就会改变差分数组中的两个点（除第一个点和最后一个点，这两点需特判）。建立一棵差分数组作为最底层的线段树，每次修改就要修改最底层的两个点。本题修改完毕就立刻查询，此时查询的最大值其实就已经保留在线段树根节点中。

【代码】

```
int m, n, num, p, y;
const int N = 2e5 + 10, INF = 1e9 + 10;
int d[N];

struct node
{
    int l, r;
    int num;
} tree[N * 4 + 10];

void build(int x, int l, int r)
{
    tree[x].l = l;
    tree[x].r = r;
    if (l == r)
    {
        tree[x].num = d[l] - d[l - 1];
        return;
    }
    int mid = (l + r) / 2;
    build(x * 2, l, mid);
    build(x * 2 + 1, mid + 1, r);
    tree[x].num = max(tree[x * 2].num, tree[x * 2 + 1].num);
}

void modify(int x, int pos)
```

```
{
    if (tree[x].l == tree[x].r && tree[x].l == pos)
    {
        tree[x].num = d[pos] - d[pos - 1];
        return;
    }
    int mid = (tree[x].l + tree[x].r) / 2;
    if (pos > tree[x].r || pos < tree[x].l)
        return;
    if (pos <= mid)
        modify(x * 2, pos);
    else
        modify(x * 2 + 1, pos);
    tree[x].num = max(tree[x * 2].num, tree[x * 2 + 1].num);
}

int main()
{
    while (scanf("%d", &n) != EOF)
    {
        memset(d, 0, sizeof d);
        d[0] = INF;
        for (int i = 1; i <= n; i++)
        {
            scanf("%d", &d[i]);
        }
        build(1, 1, n);
        scanf("%d", &m);
        while (m--)
        {
            scanf("%d%d", &p, &y);
            d[p] = y;
            if (p != 1)
                modify(1, p);
            if (p + 1 <= n)
                modify(1, p + 1);
            int ans = tree[1].num;
            printf("%.2f\n", (double)ans);
        }
    }
    return 0;
}
```

例题 【模版】线段树 2（luoguP3373）

如题，已知一个数列，你需要进行下面三种操作：

1） 将某区间每一个数乘上 x。

2） 将某区间每一个数加上 x。

3） 求出某区间每一个数的和。

【输入格式】

第一行包含三个整数n,m,p，分别表示该数列数字的个数、操作的总个数和模数。

第二行包含 n 个用空格分隔的整数，其中第i个数字表示数列第i项的初始值。

接下来m行每行包含若干个整数，表示一个操作，具体如下：

操作1： 格式：$1\ x\ y\ k$，含义：将区间 $[x,y]$ 内每个数乘上k。

操作2： 格式：$2\ x\ y\ k$，含义：将区间$[x,y]$ 内每个数加上k。

操作3： 格式：$3\ x\ y$，含义：输出区间$[x,y]$ 内每个数的和对p取模所得的结果。

【输出格式】

输出包含若干行整数，即为所有操作3的结果。

【分析】

题目要求有三种操作，两种是不同的在线修改，分别是区间乘法和区间加法，以及查询取模后的结果。可以发现这两种修改操作对于取模运算来说都是自由的。

面对这两种操作，可以使用线段树的“延迟标记”，只计算出确实需要访问的区间的真实值，其他的保存在$lazytag$里面，这样可以近似$O(NlogN)$的运行。但只有一个$lazytag$是不能够解决问题的，需要建立分别表示乘法意义上的$lazytag$和加法意义上的$lazytag$。在进行$lazy$操作的时候，需要在在向下传递$lazytag$的时候人为地为这两个$lazytag$规定一个先后顺序，排列组合一下只有两种情况：

1） 加法优先，即规定好

$$tree[root*2].s=((tree[root*2].s+tree[root].add)*tree[root].mul)\%p,$$

但问题在于这种操作不容易进行更新，如果需要改变add的数值，mul的数值也需要进行修改，并且可能产生分数从而损失精度。

2） 乘法优先，即规定好

$$tree[root*2].s=(tree[root*2]*tree[root].mul+tree[root].add*(\textit{区间长度}))\%p,$$

改变add的数值时不需要变动mul，改变mul的时候只需要把add也对应的乘一下。

综合评估下，选择乘法优先的原则进行$lazy$操作。

【代码】

```
1. typedef long long ll;
```

```
 2. const int N = 1e5 + 10;
 3. int n, m, a[N], p;
 4.
 5. struct node
 6. {
 7.     ll sum, l, r, mu, add;
 8. } tree[N * 4];
 9.
10. void build(ll x, ll l, ll r)
11. {
12.     tree[x].l = l;
13.     tree[x].r = r;
14.     tree[x].mu = 1;
15.     if (l == r)
16.     {
17.         tree[x].sum = a[l] % p;
18.         return;
19.     }
20.     ll mid = (l + r) / 2;
21.     build(x * 2, l, mid);
22.     build(x * 2 + 1, mid + 1, r);
23.     tree[x].sum = (tree[x * 2].sum + tree[x * 2 + 1].sum) % p;
24. }
25.
26. void lazy(ll x)
27. {
28.     tree[x * 2].sum = (ll)(tree[x].mu * tree[x * 2].sum + ((tree[x * 2].r - tree[x *
2].l + 1) * tree[x].add) % p) % p;
29.     tree[x * 2 + 1].sum = (ll)(tree[x].mu * tree[x * 2 + 1].sum + (tree[x].add *
(tree[x * 2 + 1].r - tree[x * 2 + 1].l + 1)) % p) % p;
30.
31.     tree[x * 2].mu = (ll)(tree[x * 2].mu * tree[x].mu) % p;
32.     tree[x * 2 + 1].mu = (ll)(tree[x * 2 + 1].mu * tree[x].mu) % p;
33.
34.     tree[x * 2].add = (ll)(tree[x * 2].add * tree[x].mu + tree[x].add) % p;
35.     tree[x * 2 + 1].add = (ll)(tree[x * 2 + 1].add * tree[x].mu + tree[x].add) % p;
36.
37.     tree[x].mu = 1, tree[x].add = 0;
38. }
39.
40. void add(ll x, ll l, ll r, ll k)
41. {
42.     if (tree[x].l >= l && tree[x].r <= r)
43.     {
```

```
44.         tree[x].add = (tree[x].add + k) % p;
45.         tree[x].sum = (ll)(tree[x].sum + k * (tree[x].r - tree[x].l + 1)) % p;
46.         return;
47.     }
48.
49.     lazy(x);
50.     tree[x].sum = (tree[x * 2].sum + tree[x * 2 + 1].sum) % p;
51.     ll mid = (tree[x].l + tree[x].r) / 2;
52.     if (l <= mid)
53.         add(x * 2, l, r, k);
54.     if (mid < r)
55.         add(x * 2 + 1, l, r, k);
56.     tree[x].sum = (tree[x * 2].sum + tree[x * 2 + 1].sum) % p;
57. }
58.
59. void mu(ll x, ll l, ll r, ll k)
60. {
61.     if (tree[x].l >= l && tree[x].r <= r)
62.     {
63.         tree[x].add = (tree[x].add * k) % p;
64.         tree[x].mu = (tree[x].mu * k) % p;
65.         tree[x].sum = (tree[x].sum * k) % p;
66.         return;
67.     }
68.
69.     lazy(x);
70.     tree[x].sum = tree[x * 2].sum + tree[x * 2 + 1].sum;
71.     ll mid = (tree[x].l + tree[x].r) / 2;
72.     if (l <= mid)
73.         mu(x * 2, l, r, k);
74.     if (mid < r)
75.         mu(x * 2 + 1, l, r, k);
76.     tree[x].sum = (tree[x * 2].sum + tree[x * 2 + 1].sum) % p;
77. }
78.
79. ll ask(ll x, ll l, ll r)
80. {
81.     if (tree[x].l >= l && tree[x].r <= r)
82.     {
83.         return tree[x].sum;
84.     }
85.     lazy(x);
86.     ll val = 0;
87.     ll mid = (tree[x].l + tree[x].r) / 2;
```

```
    if (l <= mid)
        val = (val + ask(x * 2, l, r)) % p;
    if (mid < r)
        val = (val + ask(x * 2 + 1, l, r)) % p;
    return val;
}

int main()
{
    scanf("%d%d%d", &n, &m, &p);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
    }
    build(1, 1, n);
    for (int i = 1; i <= m; i++)
    {
        int ty;
        ll cn, cm, cw;
        scanf("%d", &ty);
        if (ty == 1)
        {
            scanf("%lld%lld%lld", &cn, &cm, &cw);
            mu(1, cn, cm, cw);
        }
        else if (ty == 2)
        {
            scanf("%lld%lld%lld", &cn, &cm, &cw);
            add(1, cn, cm, cw);
        }
        else
        {
            scanf("%lld%lld", &cn, &cm);
            printf("%lld\n", ask(1, cn, cm));
        }
    }
    return 0;
}
```

6.1.3 扫描线

例题 Atlantis（POJ1151）

给定平面直角坐标系中的N个矩形，求它们的面积并，即这些矩形的并集在坐标系中覆盖的总面积，如下图 6.3 所示。

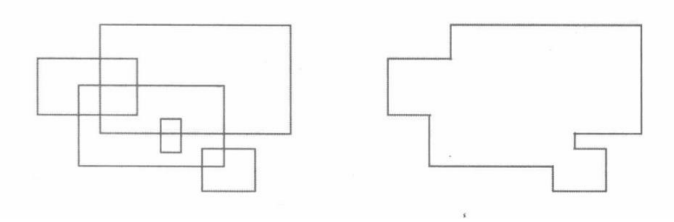

图 6.3

【分析】

如果用一条竖直直线从左到右扫过整个坐标系，那么直线上被并集图形覆盖的长度只会在每个矩形的左右边界处发生变化。换言之，整个并集图形可以被分成$2*N$段，每一段在直线上覆盖的长度(记为L)是固定的，因此该段的面积就是L *该段的宽度，各段面积之和即为所求。这条直线就被称为扫描线，这种解题思路被称为扫描线法。

对本题具体的处理方法为：

(1)每个矩形的上下边加入扫描线并标记，下边标记为1，上边标记为-1。

(2)扫描线从低到高排序，而矩形的左右边从小到大排序。

(3)考虑数据范围，进行离散化处理。

(3)建立线段树，区间可表示为区域的$y1,y2$，以及长度。

(4)遍历所有相邻左右边，每次通过线段树询问出高度，然后面积相加。

注意这里要维护的是线段长度而非点的值，所以我们要对线段树进行修改，即左孩子的右值=右孩子的左值，这样才能使维护不出现缝隙。

【代码】

```
const int N = 205;
typedef long long ll;

struct L
{
    double x, y1, y2, flag;
    L(double X = 0, double Y1 = 0, double Y2 = 0, double T = 0)
    {
        x = X, y1 = Y1, y2 = Y2, flag = T;
    }
} line[N];

struct T
{
    int l, r;
    double len;
```

```
17.     int lazy;
18. } tree[N * 4];
19.
20. bool cmp(L a, L b)
21. {
22.     return a.x < b.x;
23. }
24.
25. int n, idx, ans, cnt;
26. double x1, x2, y1, y2;
27. double seg[N];
28. map<double, int> mp;
29.
30. void build(int x, int l, int r)
31. {
32.     tree[x].l = l;
33.     tree[x].r = r;
34.     tree[x].len = 0;
35.     tree[x].lazy = 0;
36.     if (l == r)
37.         return;
38.     int mid = (l + r) >> 1;
39.     build(x * 2, l, mid);
40.     build(x * 2 + 1, mid + 1, r);
41. }
42.
43. void update(int x)
44. {
45.     if (tree[x].lazy)
46.         tree[x].len = seg[tree[x].r + 1] - seg[tree[x].l];
47.     else if (tree[x].l == tree[x].r)
48.         tree[x].len = 0;
49.     else
50.         tree[x].len = tree[x * 2].len + tree[x * 2 + 1].len;
51. }
52.
53. void change(int x, int l, int r, int k)
54. {
55.     if (l <= tree[x].l && tree[x].r <= r)
56.     {
57.         tree[x].lazy += k;
58.         update(x);
59.         return;
60.     }
```

```
61.     int mid = (tree[x].l + tree[x].r) / 2;
62.     if (l <= mid)
63.         change(x * 2, l, r, k);
64.     if (r > mid)
65.         change(x * 2 + 1, l, r, k);
66.     update(x);
67. }
68.
69. signed main()
70. {
71.     ios::sync_with_stdio(0);
72.     cin.tie(0);
73.     cout.tie(0);
74.     cin >> n;
75.     while (n)
76.     {
77.         cnt = 0;
78.         for (int i = 1; i <= n; i++)
79.         {
80.             cin >> x1 >> y1 >> x2 >> y2;
81.             line[++cnt] = L(x1, y1, y2, 1);
82.             seg[cnt] = y1;
83.             line[++cnt] = L(x2, y1, y2, -1);
84.             seg[cnt] = y2;
85.         }
86.         sort(line + 1, line + 1 + cnt, cmp);
87.         sort(seg + 1, seg + 1 + cnt);
88.         int m = unique(seg + 1, seg + 1 + cnt) - (seg + 1);
89.         for (int i = 1; i <= m; i++)
90.             mp[seg[i]] = i;
91.         build(1, 1, m);
92.         double ans = 0;
93.         for (int i = 1; i < cnt; i++)
94.         {
95.             int L = mp[line[i].y1], R = mp[line[i].y2] - 1;
96.             change(1, L, R, line[i].flag);
97.             ans += tree[1].len * (line[i + 1].x - line[i].x);
98.         }
99.         printf("Test case #%d\nTotal explored area: %.2lf\n\n", ++idx, ans);
100.        cin >> n;
101.    }
102.    return 0;
103. }
```

6.2 树状数组

同线段树类似，树状数组也是一种支持单点修改和区间查询的，代码量小的数据结构。事实上，树状数组能解决的问题是线段树能解决的问题的子集：树状数组能做的，线段树一定能做；线段树能做的，树状数组不一定可以。然而，树状数组的代码要远比线段树短，时间效率常数也更小，因此仍有学习价值。

6.2.1 基础树状数组

树状数组基本用途是维护序列的前缀和。对于给定的序列a, 建立一个数组c。规定$c[x]$管辖的区间长度为2^k，其中：

1) 设二进制最低位为第 0位，则 k恰好为x二进制表示中最低位的 1 所在的二进制位数；
2) 2^k（$c[x]$的管辖区间长度）恰好为x二进制表示中最低位的 1 以及后面所有 0 组成的数。

我们记x二进制最低位1以及后面的0组成的数为$lowbit(x)$，那么$c[x]$管辖的区间就是$[x\ -\ lowbit(x)+1, x]$。即$\sum_{i=x-lowbit(x)+1}^{x} a[i]$。

事实上，数组c可以看作一个如图 6.4 所示的树形结构，图中最下边一行是N个叶节点$(N = 16)$，代表数值$a[1\sim N]$。 该结构满足以下性质:

1. 每个内部节点$c[x]$ 保存以它为根的子树中所有叶节点的和。
2. 每个内部节点$c[x]$的子节点个数等于$lowbit(x)$ 的位数。
3. 除树根外，每个内部节点$c[x]$的父节点是$c[x\ +\ lowbit(x)]$。
4. 树的深度为$0(logN)$。

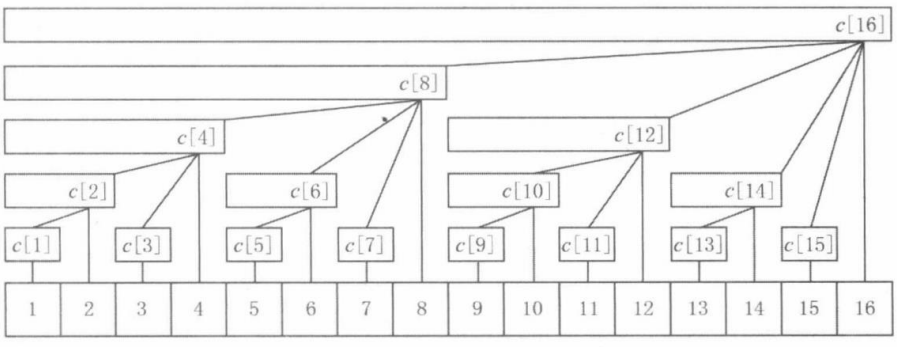

图 6.4

树状数组支持的基本操作有两个，第一个操作是查询前缀和，即序列a第$1\sim x$个数的和。按照刚才提出的方法，应该求出x的二进制表示中每个等于1的位，把$[1, x]$ 分成

$O(logN)$个小区间，而每个小区间的区间和都已经保存在数组c中。可以在$O(logN)$ 的时间内查询前缀和。

【代码】

```
1. int lowbit(int x)
2. {
3.     return x & -x;
4. }
5.
6. int getsum(int x)
7. {
8.     int ans = 0;
9.     while (x > 0)
10.    {
11.        ans = ans + c[x];
12.        x = x - lowbit(x);
13.    }
14.    return ans;
15. }
```

当然，若要查询序列a的区间$[l,r]$ 中所有数的和，只需计算$getsum(r) - getsum(l - 1)$。

树状数组支持的第二个基本操作是单点增加。意思是给序列中的一个数$a[x]$加上y，同时正确维护序列的前缀和。根据上面给出的树形结构和它的性质，只有节点$c[x]$及其所有祖先节点保存的“区间和”包含$a[x]$， 而任意一个节点的祖先至多只有$logN$个，我们逐一对它们的c值进行更新即可。下面的代码在$O(logN)$ 时间内执行单点增加操作。

【代码】

```
1. void add(int x, int k)
2. {
3.     while (x <= n)
4.     {
5.         c[x] = c[x] + k;
6.         x = x + lowbit(x);
7.     }
8. }
```

在执行所有操作之前，我们需要对树状数组进行初始化一针对原始序列 a构造一个树状数组。为了简便，比较一般的初始化方法是:直接建立一个全为0的数组c，然后对每个位置x执行$add(x,a[x])$，就完成了对原始序列a构造树状数组的过程，时间复杂度为$O(N\ logN)$。通常采用这种初始化方法已经足够。

例题 【模板】树状数组 1（luoguP3374）

如题，已知一个数列，你需要进行下面两种操作：

1） 将某一个数加上 x。

2） 求出某区间每一个数的和。

【输入格式】

第一行包含两个正整数n,m，分别表示该数列数字的个数和操作的总个数。

第二行包含n个用空格分隔的整数，其中第i个数字表示数列第i项的初始值。

接下来m行每行包含3个整数，表示一个操作，具体如下：

1 x k 含义：将第x个数加上k。

2 x y 含义：输出区间 $[x,y]$ 内每个数的和。

【输出格式】

输出包含若干行整数，即为所有操作2的结果。

【分析】

显然，区间查询和单点修改使用树状数组比线段树代码更加简洁，大大提高解题速度，减少代码篇幅。

【代码】

```
const int N = 5e5 + 10;
int c[N];
int n, m;

int lowbit(int x)
{
    return x & -x;
}

int getsum(int x)
{
    int ans = 0;
    while (x > 0)
    {
        ans = ans + c[x];
        x = x - lowbit(x);
    }
    return ans;
}

void add(int x, int k)
```

```
22. {
23.     while (x <= n)
24.     {
25.         c[x] = c[x] + k;
26.         x = x + lowbit(x);
27.     }
28. }
29.
30. int main()
31. {
32.     scanf("%d%d", &n, &m);
33.     int opt, x, y;
34.     for (int i = 1; i <= n; ++i)
35.     {
36.         scanf("%d", &x);
37.         add(i, x);
38.     }
39.     while (m--)
40.     {
41.         scanf("%d%d%d", &opt, &x, &y);
42.         if (opt == 1)
43.             add(x, y);
44.         else
45.             printf("%d\n", getsum(y) - getsum(x - 1));
46.     }
47.     return 0;
48. }
```

例题 【模板】树状数组 2（luoguP3368）

如题，已知一个数列，你需要进行下面两种操作：

1） 将某区间每一个数加上 x；

2） 求出某一个数的值。

【输入格式】

第一行包含两个整数 N、M，分别表示该数列数字的个数和操作的总个数。

第二行包含N个用空格分隔的整数，其中第i个数字表示数列第i项的初始值。

接下来M行每行包含 2 或 4个整数，表示一个操作，具体如下：

操作1： 格式：$1\ x\ y\ k$ 含义：将区间 $[x,y]$ 内每个数加上k；

操作2： 格式：$2\ x$ 含义：输出第x个数的值。

【输出格式】

输出包含若干行整数，即为所有操作 2的结果。

【分析】

本题看起来与单点修改和区间查询差不多，但实际上有很大的区别。如果按照原来的效率，得到的就是 $O(n*q)$ 。本题的切入点在于差分思想的应用。

用树状数组维护一个差分数组的前缀和，因为可推得若$b[i]=a[i]-a[i-1]$，则 $a[i]=b[1]+\ldots+b[i]$ $(b[1]=a[1]-a[0], a[0]=0)$ 。 可发现$a[i]$只与$b[j]$ （$j<=i$）有关，若将$b[j]$加上x，其后所有值都将加x，因此只需改变$b[i]$就可实现$b[i]$到$b[n]$的区间修改。而将$b[j+1]$减去x，对$a[j]$无影响，其后所有值也减去x，恢复原值，即实现了区间修改操作。 因为求取a值用到的是前缀和，因此设$t[i]$为$b[1]$到$b[i]$的前缀和，$a[i]=t[i]=b[1]+\ldots b[i]$，即可大大降低时间复杂度。

【代码】

```
const int N = 5e5 + 10;
int t[N << 2], a[N];
int n, m;

int lowbit(int x)
{
    return x & (-x);
}

int getsum(int x)
{
    int sum = 0;
    while (x > 0)
    {
        sum += t[x];
        x -= lowbit(x);
    }
    return sum;
}

void update(int x, int dlt)
{
    while (x <= n)
    {
        t[x] += dlt;
        x += lowbit(x);
    }
}

```

```
30. int main()
31. {
32.     scanf("%d%d", &n, &m);
33.     for (int i = 1; i <= n; i++)
34.     {
35.         scanf("%d", &a[i]);
36.         update(i, a[i] - a[i - 1]);
37.     }
38.     int opt, x, l, r;
39.     while (m--)
40.     {
41.         scanf("%d", &opt);
42.         if (opt == 2)
43.         {
44.             scanf("%d", &x);
45.             printf("%d\n", getsum(x));
46.         }
47.         else
48.         {
49.             scanf("%d%d%d", &l, &r, &x);
50.             update(r + 1, -x);
51.             update(l, x);
52.         }
53.     }
54.     return 0;
55. }
```

6.2.2 树状数组求第 K 大

例题 Lost Cows（POJ2182）

有n头奶牛$(n \le 10^5)$， 已知它们的身高为$1 \sim n$且各不相同，但不知道每头奶牛的具体身高。现在这n头奶牛站成一列，已知第i头奶牛前面有A头比它低，求每头奶牛的身高。

【分析】

如果最后一头奶牛前面有A_n头牛比它低，那么显然它的身高$H_n = A_n + 1$。如果倒数第二头奶牛前有A_{n-1}头牛比它低，那么：

1） 若$A_{n-1} < A_n$， 则它的身高$H_{n-1} = A_{n-1} + 1$。

2） 若$A_{n-1} \ge A_n$，则它的身高$H_{n-1} = A_{n-1} + 1$。

依此类推，如果第k头牛前面有A_k头比它低，那么它的身高H_k是数值$1 \sim n$中第A_{k+1}小的没有在$\{H_{k+1}, H_{k+2}, \ldots, H_n\}$中出现过的数。

具体来说，我们建立一个长度为n的01序列b，起初全部为1。然后，从n到1倒序扫描每个A, 对每个A_t执行以下两个操作：

1） 查询序列b中第A_{i+1}个1在什么位置，这个位置号就是第i头奶牛的身高H_i。

2） 把$b[H_i]$减1 (从1变为0)。

也就是说，我们需要实时维护一个01序列，支持查询第k个1的位置(k为任意整数)，以及修改序列中的一个数值。

方法一：树状数组+二分，单次操作$O(log^2\ n)$

用树状数组c维护01序列b的前缀和，在每次查询时二分答案，通过$ask(mid)$即可得到前mid个数中有多少个1，与k比较大小即可确定二分上下界的变化。

方法二：树状数组+倍增，单次操作$O(logn)$

用树状数组c维护01序列b的前缀和，在每次查询时：

1） 初始化两个变量$ans = 0$和$sum = 0$。

2） 从$logn$ (下取整)到0倒序考虑每个整数p。

对于每个p, 若$ans + 2^p \le n$且$sum + c[ans + 2^p] < k$, 则令$sum \mathrel{+}= c[ans + 2^p]$, $ans \mathrel{+}= 2^p$。

3）最后$H_i = ans + 1$即为所求。

【代码】

```
const int N = 1e5 + 10;
int c[N], n, a[N], b[N];

int lowbit(int x)
{
    return x & -x;
}

int ask(int x)
{
    int res = 0;
    while (x > 0)
    {
        res += c[x];
        x -= lowbit(x);
    }
    return res;
}

```

```
20. void update(int x, int d)
21. {
22.     while (x <= n)
23.     {
24.         c[x] += d;
25.         x += lowbit(x);
26.     }
27. }
28.
29. int main()
30. {
31.     scanf("%d", &n);
32.     a[1] = 0;
33.     for (int i = 2; i <= n; ++i)
34.         scanf("%d", &a[i]);
35.     for (int i = 1; i <= n; ++i)
36.         update(i, 1);
37.     for (int i = n; i >= 1; --i)
38.     {
39.         int l = 0, r = n;
40.         while (l < r)
41.         {
42.             int mid = (l + r + 1) / 2;
43.             if (ask(mid) > a[i])
44.                 r = mid - 1;
45.             else
46.                 l = mid;
47.         }
48.         b[i] = l + 1;
49.         update(l + 1, -1);
50.     }
51.     for (int i = 1; i <= n; ++i)
52.         printf("%d\n", b[i]);
53.     return 0;
54. }
```

6.2.3 多维树状数组

一维树状数组是求前缀和，$sum(p) = sum\{a[i], i <= p\}$，二维树状数组则更多用于求矩形和，$sum(x, y) = sum\{a[i][j], i <= x \&\& j <= y\}$ 。多维同样类似，在本文中我们只讨论二维树状数组的应用。

我们了解了一维树状数组的原理，二维树状数组和一维树状数组类似，在二维树状数组中， $arr[x][y]$记录的是右下角为(x, y)，高度为$lowbit(x)$，宽度为$lowbit(y)$的区间和。

单点修改+区间查询：

【代码】

```
void updata(int x, int y, int d)
{
    int temp = y;
    while (x <= n)
    {
        y = temp;
        while (y <= n)
        {
            arr[x][y] += d;
            y += lowbit(y);
        }
        x += lowbit(x);
    }
}

int getsum(int x, int y)
{
    int res = 0, temp = y;
    while (x > 0)
    {
        y = temp;
        while (y > 0)
        {
            res += arr[x][y];
            y -= lowbit(y);
        }
        x -= lowbit(x);
    }
    return res;
}
```

区间修改+单点查询：一维树状数组进行差分，就是先将原数组进行差分，在进行树状数组的操作，即一维差分+树状数组。同样的二维树状数组的区间修改+单点查询就是在二维差分的基础上进行树状数组操作。

【代码】

```
void updata(int x, int y, int d)
{
    int j = y;
    while (x <= n)
    {
```

```
        int y = j;
        while (y <= m)
        {
            tree[x][y] += d;
            y += lowbit(y);
        }
        x += lowbit(x);
    }
}

ll getsum(int x, int y)
{
    ll res = 0;
    int j = y;
    while (x > 0)
    {
        y = j;
        while (y > 0)
        {
            res += tree[x][y];
            y -= lowbit(y);
        }
        x -= lowbit(x);
    }
    return res;
}

void range_add(int a, int b, int c, int d, int k)
{
    updata(a, b, k);
    updata(a, d + 1, -k);
    updata(c + 1, b, -k);
    updata(c + 1, d + 1, k);
}
```

6.3 树的其他应用

6.3.1 树链剖分

在学习树链剖分前，请确保自己已经熟练掌握 DFS 序以及线段树的使用。

树链剖分是解决树上问题的一种常见数据结构，对于树上路径修改及路径信息查询等问题有着较优的复杂度。树链剖分分为两种：重链剖分和长链剖分，因为长链剖分不常

见，应用也不广泛，所以通常说的树链剖分指的是重链剖分。在这里讲解并总结一下树链剖分的实现、优秀性质及应用。

先来介绍几个重链剖分的专业名词：

- 重儿子：每个点的子树中，子树大小(即节点数)最大的子节点。
- 轻儿子：除重儿子外的其他子节点。
- 重边：每个节点与其重儿子间的边。
- 轻边：每个节点与其轻儿子间的边。
- 重链：重边连成的链。
- 轻链：轻边连成的链。

重链剖分顾名思义是按轻重链进行剖分，对于每个点找到重儿子，如果多个子树节点数同样多，随便选一个作为重儿子就好了，一个点也可以看做一条重链。

重链剖分的实现是由两次 dfs 来实现的，第一次 dfs 处理出每个点的重儿子$son[x]$，子树大小$size[x]$，深度$d[x]$及父节点$f[x]$，回溯时直接比较当前子节点和重儿子子树大小关系来更新重儿子。而第二遍 dfs 则是要处理出每个点所在重链的链头$top[x]$。

【代码】

```
1. void dfs(int x)
2. {
3.     size[x] = 1;
4.     d[x] = d[f[x]] + 1;
5.     for (int i = head[x]; i; i = edge[i].next)
6.     {
7.         if (edge[i].v != f[x])
8.         {
9.             f[edge[i].v] = x;
10.            dfs(edge[i].v);
11.            size[x] += size[edge[i].v];
12.            if (size[edge[i].v] > size[son[x]])
13.            {
14.                son[x] = edge[i].v;
15.            }
16.        }
17.    }
18. }
19.
20. void dfs2(int x, int tp)
21. {
22.    top[x] = tp;
```

```
23.     if (son[x])
24.         dfs2(son[x], tp);
25.     for (int i = head[x]; i; i = edge[i].next)
26.     {
27.         if (edge[i].v != f[x] && edge[i].v != son[x])
28.         {
29.             dfs2(edge[i].v, edge[i].v);
30.         }
31.     }
32. }
```

可以发现重链剖分的一些性质：

1） 所有重链互不相交，即每个点只属于一条重链。

2） 所有重链长度和等于节点数(链长指链上节点数)。

3） 一个点到根节点的路径上经过的边中轻边最多只有log条。

树链剖分与线段树相结合的问题很多，这里不一一列举，感兴趣的读者可自行学习。

6.3.2 笛卡尔树

例题 【模板】笛卡尔树（luoguP5854）

给定一个$1 \sim n$的排列p，构建其笛卡尔树。即构建一棵二叉树，满足：

1） 每个节点的编号满足二叉搜索树的性质。

2） 节点i的权值为p_i，每个节点的权值满足小根堆的性质。

【输入格式】

第一行一个整数n。

第二行一个排列$1 \ldots n$。

【输出格式】

设l_i, r_i分别表示节点i的左右儿子的编号（若不存在则为0）。

一行两个整数，分别表示$xor_{i=1}^{n} i \times (l_i + 1)$和$xor_{i=1}^{n} i \times (r_i + 1)$。

【分析】

笛卡尔树是一种二叉树，每一个结点由一个键值二元组(k, w)构成。要求k满足二叉搜索树的性质，而w满足堆的性质。一个有趣的事实是，如果笛卡尔树的(k, w)键值确定、并且k不相同，w不相同，那么这个笛卡尔树的结构是唯一的。

给定一个序列A（默认其没有重复元素），下面建树的过程以小根堆为例，定义序列中第i个元素的键值为(i, A_i)，也就是i对应k，A_i对应w。定义一棵树的右链为从根出发一

直往右儿子方向能够到达的所有点按照深度从浅到深排序后而形成的一条链。因为我们的键值k是数组下标，所以我们不需要排序，直接从数组的左边往右边插入可以。

假设我们现在插入的是节点u，为了维护下标满足二叉搜索树的性质，每次都是往树的右链的末端插入，但是此时要维护堆的性质。

1） 如果恰好 w_u大于当前右链末端端点的 w 即直接将 u 插入到右链的末端。

2） 如果 w_u小于当前右链的末端端点的w，那么意味着 u应当在树上是当前右链末端端点的祖先，继续往上找，直到遇到满足第一种情况的点。

假如我们在右链上找到了一个点j使得$w_j < w_u$，那么就把j的右儿子作为u的左儿子，j的右儿子变为u，然后我们就完成了插入这个u。

如果没有找到任何一个点可以小于w_u，那么w_u就会成为新的右链的根节点。

笛卡尔树的主要作用是处理区间 RMQ 问题，尽管不太常用，但它与平衡树与堆密切相关，希望读者能够尽量熟练。

【代码】

注：此代码若要通过全部用例需要加入快读。

```
typedef long long ll;
const int N = 1e7 + 10;
int n, a[N], s[N], l[N], r[N];
ll L, R;

int main()
{
    scanf("%d", &n);
    for (int i = 1, pos = 0, top = 0; i <= n; ++i)
    {
        scanf("%d", &a[i]);
        pos = top;
        while (pos && a[s[pos]] > a[i])
            pos--;
        if (pos)
            r[s[pos]] = i;
        if (pos < top)
            l[i] = s[pos + 1];
        s[top = ++pos] = i;
    }
    for (int i = 1; i <= n; ++i)
        L ^= 1LL * i * (l[i] + 1), R ^= 1LL * i * (r[i] + 1);
    printf("%lld %lld", L, R);
```

```
24.     return 0;
25. }
```

6.3.3 树的启发式合并

启发式合并作为一种思想，表示我们在合并两个集合时，优先将小集合合并到大集合中去。这样就能够保证合并的总时间复杂度控制在$O(nlogn)$以内。

树上启发式合并（dsu on tree）对于某些树上离线问题可以速度大于等于大部分算法且更易于理解和实现的算法。

具体做法：

1） 我们先将整颗树进行轻重链剖分，重儿子相当于一个大集合，我们在合并的过程中都是轻儿子不断的和重儿子合并。

2） 用一个全局变量记录答案和要维护的值，每次合并完都直接记录答案，是离线算法。

3） 每次递归先进入轻儿子，因为轻儿子只负责自己内部的统计，因此轻儿子所统计的结果要全部删掉。

4） 轻儿子统计完后，统计重儿子。重儿子的结果在统计完后不删除保留。因为此时轻儿子们已经都算完，直接遍历所有轻儿子的所有子树，子树对应的是一段区间，直接将这些儿子合并。

5） 统计答案，再判断 u 节点是来自于轻边还是重边，是轻边要记得删掉以 u 为子树的所有信息。

【代码】

```
1. void dfs2(int u, int fa, int del)
2. {
3.     for (int i = head[u]; ~i; i = edge[i].next)
4.     {
5.         int j = edge[i].v;
6.         if (j == son[u] || j == fa)
7.             continue;
8.         dfs2(j, u, 1);
9.     }
10.
11.     if (son[u])
12.         dfs2(son[u], u, 0);
13.
14.     for (int i = head[u]; ~i; i = edge[i].next)
15.     {
16.         int j = edge[i].v;
```

```
17.         if (j == son[u] || j == fa)
18.             continue;
19.         for (int k = l[j]; k <= r[j]; k++)
20.             update(rw[k], 1);
21.     }
22.     update(u, 1);
23.     ans[u] = sum;
24.
25.     if (del)
26.     {
27.         for (int k = l[u]; k <= r[u]; k++)
28.             update(rw[k], -1);
29.         maxv = sum = 0;
30.     }
31. }
```

例题　树上数颜色（luoguU41492）

给一棵根为1的树，每次询问子树颜色种类数。

【输入格式】

第一行一个整数n，表示树的结点数。

接下来$n-1$行，每行一条边。

接下来一行n个数，表示每个结点的颜色$c[i]$。

接下来一个数m，表示询问数。

接下来m行表示询问的子树。

【输出格式】

对于每个询问，输出该子树颜色数。

【分析】

本题支持离线，考虑预处理后$O(1)$输出答案。直接暴力预处理的时间复杂度为$O(n^2)$，即对每一个子节点进行一次遍历，每次遍历的复杂度显然与n同阶，有n个节点，故复杂度为$O(n^2)$。

可以发现，每个节点的答案由其子树和其本身得到，考虑利用这个性质处理问题。

先预处理出每个节点子树的大小和它的重儿子，重儿子同树链剖分一样，是拥有节点最多子树的儿子，这个过程显然可以$O(n)$完成。

用 $cnt[i]$ 表示颜色i的出现次数，$ans[u]$ 表示结u的答案。遍历一个节点u，我们按以下的步骤进行遍历：

1） 先遍历u的轻（非重）儿子，并计算答案，但 不保留遍历后它对 cnt 数组的影响；

2） 遍历它的重儿子，保留它对 cnt 数组的影响；

3） 再次遍历 u的轻儿子的子树结点，加入这些结点的贡献，以得到 u的答案。

【代码】

```
const int maxN = 100005;
int n, m, c[maxN]; // c[i]: 颜色标号

struct ed
{
    int adj, to;
    ed(int a = -1, int b = 0) : adj(a), to(b) {}
} edge[maxN << 1];
int head[maxN], cnt;

void add_edge(int u, int v)
{
    edge[cnt] = ed(head[u], v);
    head[u] = cnt++;
}

int siz[maxN], son[maxN]; // 子树大小 //重儿子标号

void dfs(int u, int fa)
{
    siz[u] = 1;
    for (int i = head[u]; ~i; i = edge[i].adj)
    {
        int v = edge[i].to;
        if (v == fa)
            continue;
        dfs(v, u);
        siz[u] += siz[v];
        if (siz[son[u]] < siz[v])
            son[u] = v;
    }
}

int color[maxN], ans[maxN], tot, nowSon;

void cal(int u, int fa, int val)
{
    if (!color[c[u]])
        ++tot;
    color[c[u]] += val;
```

```
41.     for (int i = head[u]; ~i; i = edge[i].adj)
42.     {
43.         int v = edge[i].to;
44.         if (v == fa || v == nowSon)
45.             continue;
46.         cal(v, u, val);
47.     }
48. }
49.
50. void dsu(int u, int fa, bool op)
51. {
52.     for (int i = head[u]; ~i; i = edge[i].adj)
53.     {
54.         int v = edge[i].to;
55.         if (v == fa || v == son[u])
56.             continue;
57.         dsu(v, u, false);
58.     }
59.     if (son[u])
60.         dsu(son[u], u, true), nowSon = son[u];
61.     cal(u, fa, 1), nowSon = 0;
62.     ans[u] = tot;
63.     if (!op)
64.     {
65.         cal(u, fa, -1);
66.         tot = 0;
67.     }
68. }
69.
70. int main()
71. {
72.     memset(head, -1, sizeof(head));
73.     n = read();
74.     for (int i = 0; i < n - 1; ++i)
75.     {
76.         int u, v;
77.         u = read();
78.         v = read();
79.         add_edge(u, v);
80.         add_edge(v, u);
81.     }
82.     for (int i = 1; i <= n; ++i)
83.         c[i] = read();
84.     dfs(1, 0);
```

```
85.     dsu(1, 0, true);
86.     m = read();
87.     for (int i = 0; i < m; ++i)
88.     {
89.         int u = read();
90.         printf("%d\n", ans[u]);
91.     }
92.     return 0;
93. }
```

6.4 并查集

6.4.1 基础并查集

并查集是用来对集合进行合并与查询操作的一种数据结构。合并就是将两个不相交的集合合并成一个集合。查询就是查询两个元素是否属于同一集合。

并查集的精妙之处在于用树来表示集合。例如，若有3个集合{1,3}、{2,5,6}、{4}，则需要用3棵树来表示。这3棵树的具体形态无关紧要，只要有一棵树包含1、3两个点，一棵树包含2、5、6这3个点，还有一棵树只包含4这一个点即可。规定每棵树的根结点是这棵树所对应的集合的代表元。

查询：只需要判断两个元素是否具有相同的头结点。

合并：只需要将一个集合的头结点挂到另一个集合的头结点下即可。

上述两个操作的时间复杂度都与“获取头结点”这一过程，也就是树的高度有关。因此，假如生成的树只有有限高度的话，合并和查询的操作都是$O(1)$的时间复杂度。

如果把x的父结点保存在$f[x]$中（如果x没有父结点，则$f[x]$等于x），则不难写出“查找结点 x 所在树的根结点”的程序, 通俗地讲就是：“如果$f[x]$等于x，说明x本身就是树根，因此返回x；否则返回x的父结点$f[x]$所在树的树根。”

```
1. int find(int x)
2. {
3.         return f[x] == x ? x : find(f[x]);
4. }
```

例题　【模板】并查集（luoguP3367）

如题，现在有一个并查集，你需要完成合并和查询操作。

【输入格式】

第一行包含两个整数N,M, 表示共有 N 个元素和M个操作。

接下来 M 行，每行包含三个整数Z_i,X_i,Y_i 。

当$Z_i = 1$时，将X_i与Y_i所在的集合合并。

当$Z_i = 2$时，输出X_i与Y_i是否在同一集合内，是的输出Y；否则输出N。

【输出格式】

对于每一个$Z_i = 2$的操作，都有一行输出，每行包含一个大写字母，为Y或者N。

【代码】

```
int find(int k)
{
    if (f[k] == k)
        return k;
    return f[k] = find(f[k]);
}

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        f[i] = i;
    for (int i = 1; i <= m; i++)
    {
        int x, y, z;
        cin >> z >> x >> y;
        if (z == 1)
            f[find(x)] = find(y);
        else if (find(x) == find(y))
            printf("Y\n");
        else
            printf("N\n");
    }
    return 0;
}
```

6.4.2 带权并查集

不同于普通的并查集，带权并查集是在并查集的基础上，根据题目的含义，对每一个节点赋予权值含义，在并查集的合并操作中，同时对权值进行操作。

例题　银河英雄传说（luoguP1196）

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成30000 列，每列依次编号为1,2, …,30000。之后，他把自己的战舰也依次编号为1,2, …,30000，让第i号战舰处于第i列，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰

集中在某几列上，实施密集攻击。合并指令为 $M\ i\ j$，含义为第i号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第j号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令：$C\ i\ j$。该指令意思是，询问电脑，杨威利的第i号战舰与第j号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

【输入格式】

第一行有一个整数 T（$1 \le T \le 5 \times 10^5$），表示总共有 T 条指令。

以下有 T行，每行有一条指令。指令有两种格式：

1. $M\ i\ j$：i和j是两个整数（$1 \le i,j \le 30000$），表示指令涉及的战舰编号。该指令是莱因哈特窃听到的杨威利发布的舰队调动指令，并且保证第i号战舰与第j号战舰不在同一列。
2. $C\ i\ j$：i和j是两个整数（$1 \le i,j \le 30000$），表示指令涉及的战舰编号。该指令是莱因哈特发布的询问指令。

输出格式

依次对输入的每一条指令进行分析和处理：

- 如果是杨威利发布的舰队调动指令，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息。
- 如果是莱因哈特发布的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 i 号战舰与第 j 号战舰之间布置的战舰数目。如果第 i 号战舰与第 j 号战舰当前不在同一列上，则输出-1。

【分析】

对于题目的数据量来说，直接模拟肯定是超过时间限制的。

先来分析一下这些指令的特点，对于每个M指令，只可能一次移动整个队列，并且是把两个队列首尾相接合并成一个队列，不会出现把一个队列分开的情况，因此，需要找到一个可以一次操作合并两个队列的方法。对于C指令：判断飞船i和飞船j是否在同一列，若

在，则输出它们中间隔了多少艘飞船。结合刚刚分析过的M指令，很容易就想到要用并查集来实现。

两艘飞船之间的飞船数量，其实就是艘飞船之间的距离，转换为了一个求距离的问题。两艘飞船都是在队列里的，最简单的求距离的方法就是前后一个一个查找，但这个方法太低效，看见多次求两个点的距离的问题，便想到用前缀和来实现：开一个$front$数组，$front[i]$表示飞船i到其所在队列队头的距离，然后飞船i和飞船j之间的飞船数量即为$abs(front[i]-front[j])-1$。

并查集的特点就是不是直接把一个队列里的所有飞船移到另一个队列后面，而是通过将要移动的队列的队头连接到另一个队列的队头上，从而间接连接两个队列。因此在这个算法的基础上，在每次合并的时候，只要更新合并前队头到目前队头的距离就可以，之后其它的就可以利用它来算出自己到队头的距离。

【代码】

```
const int N = 3e4 + 10;
int f[N], front[N], num[N];
int x, y, T;
char opt;

int find(int x)
{
    if (f[x] == x)
        return f[x];
    int fx = find(f[x]);
    front[x] += front[f[x]];
    return f[x] = fx;
}

int main()
{
    cin >> T;
    for (int i = 1; i <= 30000; ++i)
    {
        f[i] = i;
        front[i] = 0;
        num[i] = 1;
    }
    while (T--)
    {
        cin >> opt >> x >> y;
```

```
27.         int fx = find(x);
28.         int fy = find(y);
29.
30.         if (opt == 'M')
31.         {
32.             front[fx] += num[fy];
33.             f[fx] = fy;
34.             num[fy] += num[fx];
35.             num[fx] = 0;
36.         }
37.
38.         if (opt == 'C')
39.         {
40.             if (fx != fy)
41.                 cout << "-1" << endl;
42.             else
43.                 cout << abs(front[x] - front[y]) - 1 << endl;
44.         }
45.     }
46.     return 0;
47. }
```

6.5 分块

分块是一种思想，而不是一种数据结构。分块的基本思想是，通过对原数据的适当划分，并在划分后的每一个块上预处理部分信息，从而较一般的暴力算法取得更优的时间复杂度。分块的时间复杂度主要取决于分块的块长，一般可以通过均值不等式求出某个问题下的最优块长，以及相应的时间复杂度。

分块是一种很灵活的思想，相较于树状数组和线段树，分块的优点是通用性更好，可以维护很多树状数组和线段树无法维护的信息。

当然，分块的缺点是渐进意义的复杂度，相较于线段树和树状数组不够好。不过在大多数问题上，分块仍然是解决这些问题的一个不错选择。

6.5.1 莫队

例题 [国家集训队]小 z 的袜子（luoguP1494）

作为一个生活散漫的人，小 Z 每天早上都要耗费很久从一堆五颜六色的袜子中找出一双来穿。终于有一天，小 Z 再也无法忍受这恼人的找袜子过程，于是他决定听天由命……

具体来说，小 Z 把这 N 只袜子从1到N编号，然后从编号L到R（尽管小 Z 并不在意两只袜子是不是完整的一双，甚至不在意两只袜子是否一左一右，他却很在意袜子的颜色，毕竟穿两只不同色的袜子会很尴尬。

你的任务便是告诉小 Z，他有多大的概率抽到两只颜色相同的袜子。当然，小 Z 希望这个概率尽量高，所以他可能会询问多个 (L,R) 以方便自己选择。

然而数据中有$L=R$ 的情况，请特判这种情况，输出$0/1$。

【输入格式】

输入文件第一行包含两个正整数N和M。N 为袜子的数量，M 为小 Z 所提的询问的数量。接下来一行包含N个正整数C_i，其中C_i表示第i只袜子的颜色，相同的颜色用相同的数字表示。再接下来M行，每行两个正整数L,R 表示一个询问。

【输出格式】

包含M行，对于每个询问在一行中输出分数A/B 表示从该询问的区间 $[L,R]$ 中随机抽出两只袜子颜色相同的概率。若该概率为 0 则输出 $0/1$，否则输出的A/B 必须为最简分数。

【分析】

莫队算法是离线处理一类区间不修改查询类问题的算法。就是如果你知道了$[L,R]$的答案。可以在$O(1)$的时间下得到$[L,R-1]$和$[L,R+1]$和$[L-1,R]$和$[L+1,R]$的情况下可以使用莫队算法。

对于L,R的询问。设其中颜色为$x,y,z....$的袜子的个数为$a,b,c....$那么答案即为$(a*(a-1)/2+b*(b-1)/2+c*(c-1)/2....)/((R-L+1)*(R-L)/2)$，也就是：$(a^2+b^2+c^2+...x^2-(R-L+1))/((R-L+1)*(R-L))$。所以关键在于求一个区间内每种颜色数目的平方和。

莫队算法只是预先知道了所有的询问。可以合理的组织计算每个询问的顺序以此来降低复杂度。算完$[L,R]$的答案后，要算$[L',R']$的答案，花的时间为$|L-L'|+|R-R'|$。如果把询问$[L,R]$看做平面上的点$a(L,R)$。询问$[L',R']$看做点$b(L',R')$。那么时间开销就为两点的曼哈顿距离。所以对于每个询问看做一个点。我们要按一定顺序计算每个值。那开销就为曼哈顿距离的和。要计算到每个点。那么路径至少是一棵树。所以问题就变成了求二维平面的最小曼哈顿距离生成树。

但这种方法编程复杂度稍高。所以考虑其他做法，先对序列分块。然后对于所有询问按照L所在块的大小排序。如果一样再按照R排序。然后按照排序后的顺序计算。复杂度大致在$O(n^{1.5})$左右。

【代码】

```
typedef long long ll;
const int N = 5e4 + 10;

int n, m, unit, col[N], be[N];
ll sum[N], ans;

struct Mo
{
    int l, r, id;
    ll A, B;
};
Mo q[N];

ll S(ll x)
{
    return x * x;
}

ll gcd(ll a, ll b)
{
    while (b ^= a ^= b ^= a %= b)
        ;
    return a;
}

int cmp1(Mo a, Mo b)
{
    return be[a.l] == be[b.l] ? a.r < b.r : a.l < b.l;
}

int cmp(Mo a, Mo b)
{
    return a.id < b.id;
}

void revise(int x, int add)
{
    ans -= S(sum[col[x]]);
```

```
39.     sum[col[x]] += add;
40.     ans += S(sum[col[x]]);
41. }
42.
43. int main()
44. {
45.     scanf("%d%d", &n, &m);
46.     unit = sqrt(n);
47.     for (int i = 1; i <= n; i++)
48.     {
49.         scanf("%d", &col[i]);
50.         be[i] = i / unit + 1;
51.     }
52.     for (int i = 1; i <= m; i++)
53.     {
54.         scanf("%d%d", &q[i].l, &q[i].r);
55.         q[i].id = i;
56.     }
57.     sort(q + 1, q + m + 1, cmp1);
58.     int l = 1, r = 0;
59.     for (int i = 1; i <= m; i++)
60.     {
61.         while (l < q[i].l)
62.         {
63.             revise(l, -1);
64.             l++;
65.         }
66.         while (l > q[i].l)
67.         {
68.             revise(l - 1, 1);
69.             l--;
70.         }
71.         while (r < q[i].r)
72.         {
73.             revise(r + 1, 1);
74.             r++;
75.         }
76.         while (r > q[i].r)
77.         {
78.             revise(r, -1);
79.             r--;
80.         }
81.         if (q[i].l == q[i].r)
82.         {
```

```
            q[i].A = 0;
            q[i].B = 1;
            continue;
        }
        q[i].A = ans - (q[i].r - q[i].l + 1);
        q[i].B = 1LL * (q[i].r - q[i].l + 1) * (q[i].r - q[i].l);
        ll g = gcd(q[i].A, q[i].B);
        q[i].A /= g;
        q[i].B /= g;
    }
    sort(q + 1, q + m + 1, cmp);
    for (int i = 1; i <= m; i++)
    {
        printf("%lld/%lld\n", q[i].A, q[i].B);
    }
    return 0;
}
```

6.5.2 树上分块

例题　Count on a tree II/[模板]树分块（luoguP6177）

给定一个n个节点的树，每个节点上有一个整数，i号点的整数为val_i。

有m次询问，每次给出u',v，您需要将其解密得到u,v，并查询u到v的路径上有多少个不同的整数。

解密方式：$u = u' \ xor \ lastans$。$lastans$为上一次询问的答案，若无询问则为0。

【输入格式】

第一行有两个整数n和m。

第二行有n个整数。第i个整数表示val_i。

在接下来的$n-1$行中，每行包含两个整数u,v，描述一条边。

在接下来的m行中，每行包含两个整数u',v，描述一组询问。

【输出格式】

对于每个询问，一行一个整数表示答案。

【分析】

树分块的大致思想是将树上的点每次划分进$\sqrt{n}$（或者基于题目性质的更优块大小）个块中，对与块内信息进行维护的算法。基于不同的题目，会有很多种不同的分块方法。在一般情况下，树分块在处理树的路径等具有连通性质的问题时表现较为强力。

在本题中，随机找$\sqrt{n}$个关键点，对于所有点找到它的第一个关键祖先，将它和关键祖先分为一块。可以保证块联通，在期望下块直径长度和块大小为$\sqrt{n}$。

分块以后的每个询问被拆成了两个零散块和一个整块，我们记两个零散块所构成的集合分别为A和B，整块构成的集合为C，那么答案就为$|A|+|B|+|C|-|A\cap B|-|B\cap C|-|A\cap C|+|A\cap B\cap C|$。而$|A|,|B|,|A\cap B|,|A\cap B\cap C|$ 可以在处理零散块的时候直接计算，剩下的可以 $O(n\sqrt{n})$ 预处理。

【代码】

```
struct ss
{
    int node, nxt;
} e[N];

void add(int u, int v)
{
    e[++tot].nxt = head[u];
    e[tot].node = v;
    head[u] = tot;
}

void dfs(int x, int ffa, int dp)
{
    fa[x] = ffa;
    d[x] = dp;
    sz[x] = 1;

    int maxsz = -1;
    for (int i = head[x]; i; i = e[i].nxt)
    {
        int k = e[i].node;
        if (k == ffa)
            continue;
        if (!vis[k])
            g[k] = g[x];
        else
            g[k] = k;
        dfs(k, x, dp + 1);
        sz[x] += sz[k];
        if (maxsz < sz[k])
            maxsz = sz[k], son[x] = k;
    }
}
```

```
35.
36. void ddfs(int x, int tt)
37. {
38.     top[x] = tt;
39.     l[x] = h[a[x]];
40.     if (!son[x])
41.         return;
42.     h[a[x]] = x;
43.     ddfs(son[x], tt);
44.     for (int i = head[x]; i; i = e[i].nxt)
45.     {
46.         int k = e[i].node;
47.         if (fa[x] == k || son[x] == k)
48.             continue;
49.         ddfs(k, k);
50.     }
51.     h[a[x]] = l[x];
52. }
53.
54. int lca(int x, int y) // 树剖 LCA
55. {
56.     while (top[x] != top[y])
57.     {
58.         if (d[top[x]] < d[top[y]])
59.             swap(x, y);
60.         x = fa[top[x]];
61.     }
62.     if (d[x] > d[y])
63.         swap(x, y);
64.     return x;
65. }
66.
67. void dp(int x, int fa, int *ff, int *s, int ls, int ans)
68. {
69.     if (!x)
70.         return;
71.     s[++ls] = a[x];
72.     if (t[a[x]] && !vis2[a[x]])
73.         ans++;
74.     vis2[s[ls]]++;
75.     if (vis[x])
76.     {
77.         for (int i = 1; i <= ls; i++)
78.         {
```

```
79.             if (!t[s[i]])
80.                 tt++;
81.             t[s[i]]++, vis2[s[i]]--;
82.         }
83.         ff[x] = tt;
84.     }
85.     else
86.         ff[x] = ans;
87.     for (int i = head[x]; i; i = e[i].nxt)
88.     {
89.         if (e[i].node == fa)
90.             continue;
91.         if (vis[x])
92.             dp(e[i].node, x, ff, s + ls, 0, 0);
93.         else
94.             dp(e[i].node, x, ff, s, ls, ans);
95.     }
96.     if (vis[x])
97.     {
98.         for (int i = 1; i <= ls; i++)
99.         {
100.            t[s[i]]--;
101.            vis2[s[i]]++;
102.            if (!t[s[i]])
103.                tt--;
104.        }
105.    }
106.    vis2[s[ls]]--;
107. }
108.
109. int sol(int u, int v, int typ)
110. {
111.    int lans = 0;
112.    int x = u, y = v, LCA = lca(u, v);
113.    while (x != LCA)
114.    {
115.        if (!vis1[a[x]])
116.            lans++;
117.        vis1[a[x]] = 1;
118.        x = fa[x];
119.    }
120.    while (y != LCA)
121.    {
122.        if (!typ || y != v)
```

```
123.         {
124.             if (!vis1[a[y]])
125.                 lans++;
126.             vis1[a[y]] = 1;
127.         }
128.         y = fa[y];
129.     }
130.     if (!vis1[a[LCA]] && (!typ || LCA != v))
131.         lans++;
132.     while (u != LCA)
133.     {
134.         vis1[a[u]] = 0;
135.         u = fa[u];
136.     }
137.     while (v != LCA)
138.     {
139.         vis1[a[v]] = 0;
140.         v = fa[v];
141.     }
142.     return lans;
143. }
144.
145. int b[N], cnt, q;
146.
147. void solve1(int u, int v)
148. {
149.     int x = u, y = v, LCA = lca(u, v);
150.     while (x != LCA)
151.     {
152.         if (!vis1[a[x]])
153.             lans++;
154.         vis1[a[x]] = 1;
155.         x = fa[x];
156.     }
157.     while (y != LCA)
158.     {
159.         if (!vis1[a[y]])
160.             lans++;
161.         vis1[a[y]] = 1;
162.         y = fa[y];
163.     }
164.     if (!vis1[a[LCA]])
165.         lans++;
166.     while (u != LCA)
```

```
167.     {
168.         vis1[a[u]] = 0;
169.         u = fa[u];
170.     }
171.     while (v != LCA)
172.     {
173.         vis1[a[v]] = 0;
174.         v = fa[v];
175.     }
176.     write(lans);
177.     puts("");
178. }
179.
180. int main()
181. {
182.     n = read();
183.     q = read();
184.     for (int i = 1; i <= n; i++)
185.     {
186.         b[++cnt] = a[i] = read();
187.         c[i] = i;
188.     }
189.     for (int i = 1, u, v; i < n; i++)
190.     {
191.         u = read();
192.         v = read();
193.         add(u, v);
194.         add(v, u);
195.     }
196.     srand(time(0));
197.     sort(b + 1, b + cnt + 1);
198.     cnt = unique(b + 1, b + cnt + 1) - b - 1;
199.     random_shuffle(c + 2, c + n + 1);
200.     for (int i = 1; i <= n; i++)
201.     {
202.         if (i <= BB)
203.             p[i] = c[i], vis[c[i]] = 1;
204.         a[i] = lower_bound(b + 1, b + cnt + 1, a[i]) - b;
205.     }
206.     g[1] = 1;
207.     dfs(1, 0, 1);
208.     ddfs(1, 1);
209.     for (int i = 1; i <= BB; i++)
210.         dp(p[i], 0, f[i], fs, 0, 0);
```

```
211.     lans = 0;
212.     memset(vis1, 0, sizeof(vis1));
213.     memset(vis2, 0, sizeof(vis2));
214.     for (int i = 1, u, v; i <= q; i++)
215.     {
216.         u = read() ^ lans;
217.         v = read();
218.         if (g[u] == g[v]) // 零散块
219.         {
220.             lans = 0;
221.             solve1(u, v);
222.             continue;
223.         }
224.         int LCA = lca(u, v);
225.         if (d[g[u]] < d[LCA] || d[g[v]] < d[LCA])
226.         {
227.             if (d[g[u]] < d[LCA])
228.                 swap(u, v);
229.             int x = u, y = v, z = u, A = 0, B = 0, C = 0, AB = 0, BC = 0, AC = 0, ABC
= 0, xx = u, yy = v, ty;
230.             while (!vis[x] && x != LCA)
231.             {
232.                 if (!vis1[a[x]])
233.                     A++;
234.                 vis1[a[x]] = x;
235.                 x = fa[x];
236.             }
237.             while (!vis[y] && y != LCA)
238.             {
239.                 if (!vis2[a[y]])
240.                 {
241.                     B++;
242.                     if (vis1[a[y]])
243.                     {
244.                         AB++;
245.                     }
246.                 }
247.                 vis2[a[y]] = y;
248.                 y = fa[y];
249.             }
250.             while (d[g[fa[g[z]]]] > d[LCA])
251.                 z = fa[g[z]];
252.             ty = z = g[z];
253.             z = fa[z];
```

```
254.            int zz = z;
255.            while (!vis[z] && z != LCA)
256.            {
257.                if (!vis2[a[z]])
258.                {
259.                    B++;
260.                    if (vis1[a[z]])
261.                    {
262.                        AB++;
263.                    }
264.                }
265.                vis2[a[z]] = z;
266.                z = fa[z];
267.            }
268.            if (!vis2[a[LCA]])
269.            {
270.                B++;
271.                if (vis1[a[LCA]])
272.                    AB++;
273.            }
274.            vis2[a[LCA]] = LCA;
275.            while (!vis[yy] && yy != LCA)
276.            {
277.                if (vis2[a[yy]] && vis1[a[yy]])
278.                {
279.                    if ((d[l[vis1[a[yy]]]] >= d[ty] && d[l[vis1[a[yy]]]] <= d[x]))
280.                        ABC++;
281.                }
282.                vis2[a[yy]] = 0;
283.                yy = fa[yy];
284.            }
285.            while (!vis[zz] && zz != LCA)
286.            {
287.                if (vis2[a[zz]] && vis1[a[zz]])
288.                {
289.                    if ((d[l[vis1[a[zz]]]] >= d[ty] && d[l[vis1[a[zz]]]] <= d[x]))
290.                        ABC++;
291.                }
292.                vis2[a[zz]] = 0;
293.                zz = fa[zz];
294.            }
295.            if (vis2[a[LCA]])
296.            {
297.                if (vis1[a[LCA]])
```

```
298.                     if ((d[l[vis1[a[LCA]]]] >= d[ty] && d[l[vis1[a[LCA]]]] <= d[x]))
299.                         ABC++;
300.             }
301.             vis2[a[LCA]] = 0;
302.             while (!vis[xx] && xx != LCA)
303.             {
304.                 vis1[a[xx]] = 0;
305.                 xx = fa[xx];
306.             }
307.             for (int j = 1; j <= BB; j++)
308.                 if (p[j] == x)
309.                 {
310.                     x = j;
311.                     break;
312.                 }
313.             for (int j = 1; j <= BB; j++)
314.                 if (p[j] == ty)
315.                 {
316.                     ty = j;
317.                     break;
318.                 }
319.             C = f[x][p[ty]];
320.             if (!vis[v])
321.                 BC = f[x][v];
322.             if (!vis[u])
323.                 AC = f[ty][u];
324.             lans = A + B + C - AB - BC - AC + ABC;
325.             write(lans);
326.             puts("");
327.             continue;
328.         }
329.
330.         int x = u, y = v, A = 0, B = 0, C = 0, AB = 0, BC = 0, AC = 0, ABC = 0, xx =
u, yy = v;
331.         while (!vis[x] && x != LCA)
332.         {
333.             if (!vis1[a[x]])
334.                 A++;
335.             vis1[a[x]] = x;
336.             x = fa[x];
337.         }
338.         while (!vis[y] && y != LCA)
339.         {
340.             if (!vis2[a[y]])
```

```
341.            {
342.                B++;
343.                if (vis1[a[y]])
344.                {
345.                    AB++;
346.                }
347.            }
348.            vis2[a[y]] = y;
349.            y = fa[y];
350.        }
351.        while (!vis[yy] && yy != LCA)
352.        {
353.            if (vis2[a[yy]] && vis1[a[yy]])
354.            {
355.                if (((d[l[vis1[a[yy]]]] >= d[LCA]) && (d[l[vis1[a[yy]]]] <= d[x])) ||
356.                    ((d[l[vis2[a[yy]]]] >= d[LCA]) && (d[l[vis2[a[yy]]]] <= d[y])))
357.                    ABC++;
358.            }
359.            vis2[a[yy]] = 0;
360.            yy = fa[yy];
361.        }
362.        while (!vis[xx] && xx != LCA)
363.        {
364.            vis1[a[xx]] = 0;
365.            xx = fa[xx];
366.        }
367.        for (int j = 1; j <= BB; j++)
368.            if (p[j] == x)
369.            {
370.                x = j;
371.                break;
372.            }
373.        for (int j = 1; j <= BB; j++)
374.            if (p[j] == y)
375.            {
376.                y = j;
377.                break;
378.            }
379.        C = f[x][p[y]];
380.        if (!vis[v])
381.            BC = f[x][v];
382.        if (!vis[u])
383.            AC = f[y][u];
384.        lans = A + B + C - AB - BC - AC + ABC;
```

```
385.         write(lans);
386.         puts("");
387.     }
388.     return 0;
389. }
```