17. 树状数组与线段树

17.1 树状数组

用一个数组或前缀和数组来维护区间信息,则查询的时间复杂度为$O(n)$,单点修改的时间复杂度为$O(1)$.

树状数组的常用操作:①快速求前缀和,时间复杂度$O(\log n)$;②单点修改,时间复杂度$O(\log n)$.

树状数组的常用扩展:①差分;②差分+推公式.

设$x=2^{i_k}+2^{i_{k-1}}+\cdots+2^{i_1}$,其中$i_k\geq i_{k-1}\geq\cdots\geq i_1;k\leq\log_2 x$.

计算$1\sim x$的和时,可按二的幂次划分为k部分:$(x-2^{i_1},x],(x-2^{i_1}-2^{i_2},x-2^{i_1}],\cdots,(0,x-2^{i_1}-2^{i_2}-\cdots-2^{i_k}]$,每个区间分别有$2^{i_1},2^{i_2},\cdots,2^{i_k}$个数,数的个数分别是区间右端点的二进制表示中的最后一个1,则上述区间$(L,R]$即$(R-lowbit(R)+1,R]$.

预处理出上述区间内的和,则可在$O(\log n)$的时间复杂度内计算出$1\sim x$的前缀和.

设原数组$a[x-lowbit(x)+1,\ x]$的区间和为$c[x]$,则$a[1,x]$的前缀和可分为$\lfloor\log_2 x\rfloor$个$c[x]$.

设下标从1开始,i为奇数时,$c[i]=a[i]$;i为偶数时,
$c[2]=a[1,2],c[4]=a[1,4],c[6]=a[5,6],c[8]=a[1,8],c[10]=a[9,10],c[12]=a[9,12],c[14]=a[13,14],c[16]=a[1,16],\cdots$.

考察如何最快计算出$c[16]$:①$a[16]$单独出现;②$a[15]$被$c[15]$覆盖;③$a[13,14]$被$c[14]$覆盖;④$a[9,12]$被$c[12]$覆盖;⑤$a[1,8]$被$c[8]$覆盖,故
$c[16]=c[8]+c[12]+c[14]+c[15]+a[16]$.同理
$c[8]=c[4]+c[6]+c[7]+a[8],c[4]=c[2]+c[3]+a[4],c[12]=c[10]+c[11]+a[12],c[2]=c[1]+a[2],c[6]=c[5]+a[6],c[10]=c[9]+a[10],c[14]=c[13]+a[14]$
.

通过父节点找子节点:对$x>0,x$的二进制表示中存在最后一个1,设它之后有k个0,则$c[x]$是以x为右端点,长度为2^k的区间和$a[x-2^k+1,x-1]$.$x-1$的二进制表示最后有k个1,按这k个1分为k份.下面以$k=4$为例,分为$(\cdots01110,\cdots01111],(\cdots01100,\cdots01110],(\cdots01000,\cdots01100],(\cdots00000,\cdots01000]$.则
$c[x]=a[x]+c[x-1]+c[x-1-lowbit(x-1)]+\cdots+0$.

通过子节点找父节点(对应修改操作):考察修改$a[x]$后需更新哪些$c[]$,即包含$a[x]$的父节点.设x的二进制表示为$\cdots01\cdots10\cdots0$,则直接包含$a[x]$的父节点为$c[\cdots10\cdots00\cdots0]$,即单点修改后只影响到一个$c[]$,它是$c[x+lowbit(x)]$.同理更新包含$c[x+lowbit(x)]$的父节点,直至更新完所有间接包含$a[x]$的$c[]$.注意到每更新一个$c[]$,下标末尾0的个数至少加1,而$x$最多有$\lfloor\log_2 x\rfloor$位,故单点修改至多影响$\lfloor\log_2 x\rfloor$个$c[]$.

17.1.1 楼兰图腾

题意

一幅壁画中有n $(1\leq n\leq 2e5)$个点, 它们的水平位置和竖直位置两两不同, 壁画的信息与这n个点的相对位置有关, 它们的坐标分别为$(1,y_1),(2,y_2),\cdots,(n,y_n)$, 其中$y_1,\cdots,y_n$是$1\sim n$的一个排列.

①若三个点$(i,y_i),(j,y_j),(k,y_k)$满足$1\leq i<j<k\leq n$且$y_i>y_j,y_j<y_k$, 则称这三点构成A图腾.

②若三个点$(i,y_i),(j,y_j),(k,y_k)$满足$1\leq i<j<k\leq n$且$y_i<y_j,y_j>y_k$, 则称这三点构成B图腾.

输出两个数, 分别表示该壁画上A、B图腾的数量, 数据保证答案不超过int64.

思路

对A图腾, 可按最下方的y_j是哪个点分为$1\sim n$类, 显然这样的划分是不重不漏的. 考察其中第k类的集合的元素个数, 只需分别统计出y_j左边和右边有几个大于y_j的数, 它们之积即该类集合的元素个数. 只需分别统计y_j左边和右边有多少个数在$[y_j+1,n]$的范围内. 维护一个数组$higher[k]$, 从前往后扫一遍统计$1\sim(k-1)$中有多少个数大于y_j, 从后往前扫一遍统计$(k+1)\sim n$中有多少个数大于y_j. 统计的过程可用BIT, 每查询一个数就将该数y_k加入集合, 相当于y_k++.

B图腾同理, 维护数组$lower[k]$.

统计一个数左右大于或小于它的数的数量是n^2级别的, 有n个数, 则答案是n^3级别的, n最大为2e5, 故要开long long.

代码

```
template<class T>
struct FenwickTree {
    int n;
    vector<T> BIT;  // BIT维护a[]

    FenwickTree(int _n) :n(_n) {
        BIT.resize(n + 2);
    }

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

    void add(int pos, int val) {  // a[pos] += val
        for (int i = pos; i <= n; i += lowbit(i))
            BIT[i] += val;
```

```
    }

    T sum(int pos) {  // a[1...pos]之和
        T res = 0;
        for (int i = pos; i; i -= lowbit(i))
            res += BIT[i];
        return res;
    }

    T query(int l, int r) {  // a[l...r]之和
        return sum(r) - sum(l - 1);
    }

    void clear() {
        fill(all(BIT), 0);
    }
};

void solve() {
    int n; cin >> n;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    // lower[i]、higher[i]分别表示a[i]左右两边大于、小于它的数的个数
    vector<int> lower(n + 1), higher(n + 1);

    FenwickTree<int> bit(n);
    for (int i = 1; i <= n; i++) {  // 从前往后扫预处理lower[]和higher[]
        lower[i] = bit.query(1, a[i] - 1);
        higher[i] = bit.query(a[i], n);

        bit.add(a[i], 1);
    }

    bit.clear();  // 注意清空
    ll ans1 = 0, ans2 = 0;  // A类为'V', B类为'∧'
    for (int i = n; i; i--) {  // 从后往前扫统计答案
        ans1 += (ll)higher[i] * bit.query(a[i], n);
        ans2 += (ll)lower[i] * bit.query(1, a[i] - 1);

        bit.add(a[i], 1);
    }
    cout << ans1 << ' ' << ans2 << endl;
}

int main() {
    solve();
}
```

17.1.2 树状数组模板I

题意

给定长度为n $(1 \le n \le 1e5)$的数列$a[]$(其中元素的绝对值$\le$ 1e9)和m $(1 \le m \le 1e5)$个操作, 操作命令有如下两种类型:

①$C\ l\ r\ d$, 表示将$a[l,r]$中的每个数加d $(|d| \le 1e4)$.

②$Q\ x$, 表示输出第x个数的值.

思路

给定一个原数组, 建BIT的方法:

①树状数组初始化为0, 将原数组的每个元素作插入操作, 总时间复杂度$O(n \log n)$.

②求出每个节点x的直接儿子节点, 每次只添加BIT中的树边. 原数组和BIT共$2n$个节点, 有$(2n-1)$条边, 至多添加$(2n-1)$次, 总时间复杂度$O(n)$, 但一般用不到该方法, 因为查询的时间复杂度也是$O(\log n)$. 该方法的实现:

```
for(int i = x - 1; i; i -= lowbit(i)) BIT[x] += BIT[i];
```

③预处理出原数组的前缀和数组$pre[]$, 则$BIT[x] = pre[x] - pre[x - lowbit(x)]$, 总时间复杂度$O(n)$, 一般也用不到.

①操作为区间修改, 可先预处理出原数组$a[]$的差分数组$diff[]$, 则$a[l \cdots r]$中每个数加d只需$diff[l] += c, diff[r+1] -= c$,将区间修改转化为两个单点修改, 故可直接用$diff[]$建BIT.

②操作查询某个$a[x]$时, 只需求一次差分数组的前缀和.

代码

```cpp
template<class T>
struct FenwickTree {
    int n;
    vector<T> BIT;  // BIT维护diff[]

    FenwickTree(int _n) :n(_n) {
        BIT.resize(n + 2);
    }

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

    void add(int pos, int val) {  // diff[pos] += val
        for (int i = pos; i <= n; i += lowbit(i))
            BIT[i] += val;
    }

    void modify(int l, int r, int val) {  // a[l...r] += val
        add(l, val), add(r + 1, -val);
    }

    T sum(int pos) {  // a[1...pos]之和
        T res = 0;
        for (int i = pos; i; i -= lowbit(i))
            res += BIT[i];
        return res;
    }

    T query(int l, int r) {  // a[l...r]之和
        return sum(r) - sum(l - 1);
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    FenwickTree<ll> bit(n);
    for (int i = 1; i <= n; i++)
        bit.add(i, a[i] - a[i - 1]);

    while (m--) {
        char op; cin >> op;
        if (op == 'C') {
            int l, r, d; cin >> l >> r >> d;
            bit.modify(l, r, d);
        }
        else {
            int pos; cin >> pos;
            cout << bit.query(1, pos) << endl;
        }
    }
}

int main() {
    solve();
}
```

17.1.3 树状数组模板II

题意

给定长度为n $(1 \leq n \leq 1e5)$的数列$a[]$(其中元素的绝对值$\leq$ 1e9)和m $(1 \leq m \leq 1e5)$个操作,操作命令有如下两种类型:

①C l r d,表示将$a[l,r]$中的每个数加d $|d| \leq 1e4$.

②Q l r,输出数列中第$l \sim r$个数的和.

思路

②操作为区间查询,先考察如何求$a[1,x]$的前缀和.注意到$\sum_{i=1}^{x} a[i] = \sum_{i=1}^{x} \sum_{j=1}^{i} diff[i]$,其中

i	$diff_1$	$diff_2$	$\cdots$	$diff_x$
1	$diff_1$			
2	$diff_1$	$diff_2$		
$\vdots$	$\vdots$	$\vdots$	$\ddots$	
x	$diff_1$	$diff_2$	$\cdots$	$diff_x$

考虑将该表格补全,则$\sum_{i=1}^{x}\sum_{j=1}^{i} diff[i] = (diff[1]+\cdots+diff[x])*(x+1)-(diff[1]-2*diff[2]-3*diff[3]-\cdots-x*diff[x])$,注意到RHS的第二项是$i\cdot diff_i$的前缀和,故只需用$BIT1[]$和$BIT2[]$分别维护$diff[i]$和$i*diff[i]$两个前缀和.

注意区间和爆int,故要开long long.

代码

```
const int MAXN = 1e5 + 5;
int n, m;  // 点数、操作数
int a[MAXN];  // 原数组
ll BIT1[MAXN];  // 维护diff[i]的前缀和
ll BIT2[MAXN];  // 维护i*diff[i]的前缀和

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

void add(ll BIT[], int x, ll c) {  // BIT单点修改
  for (int i = x; i <= n; i += lowbit(i)) BIT[i] += c;
}

ll sum(ll BIT[], int x) {  // BIT求[1...x]的前缀和
  ll res = 0;
  for (int i = x; i; i -= lowbit(i)) res += BIT[i];
  return res;
}

ll get_pre(int x) {  // 求前缀和
  return sum(BIT1, x) * (ll)(x + 1) - sum(BIT2, x);
}

int main() {
  for (int i = 1; i <= n; i++) cin >> a[i];

  for (int i = 1; i <= n; i++) {
    int diff = a[i] - a[i - 1];
    add(BIT1, i, diff);
    add(BIT2, i, (ll)diff * i);
  }

  while (m--) {
    char op; int l, r; cin >> op >> l >> r;
    if (op == 'Q')
      cout << get_pre(r) - get_pre(l - 1) << endl;
    else {
      int d; cin >> d;

      // a[l]+=d
      add(BIT1, l, d), add(BIT2, l, l * d);

      // a[r+1]-=d
      add(BIT1, r + 1, -d), add(BIT2, r + 1, (r + 1) * -d);
    }
  }
}
```

17.1.4 谜一样的牛

题意

有n $(1\le n\le 1e5)$头牛,它们的身高为$1\sim n$且互不相同,但不知道每头牛的具体身高.现它们站成一列,已知第i头牛前有a_i头牛比它矮$(i\ge 2)$,依次输出每头牛的身高.

思路

从后往前扫,第n头牛前有a_n头牛比它矮,则它排第(a_n+1).同理求得每头在剩下的牛中排名,即每次的操作都是从剩余的数中找出第k小的数并将其删去.这两个操作可用平衡树维护,但代码难写,考虑用BIT维护.

BIT初始化为1,表示每个身高可用一次,维护$a[1,n]$的前缀和.注意到$sum(x)$表示$1\sim x$中剩下几个数,则在剩下的数中找到一个第k小的数等价于求最小的$x\ \ s.t.\ \ sum(x)=k$.因$sum(x)$单调,故可用二分,总时间复杂度$O(n\log^2 n)$.

BIT初始化可用add(i,1),也可用$BIT[i] = lowbit(i)$,即区间长度都为1.

代码

```cpp
const int MAXN = 1e5 + 5;
int n;  // 牛数
int a[MAXN];  // 牛的高度
int BIT[MAXN];  // 树状数组
int ans[MAXN];  // 存牛的身高

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

void add(int x, int c) {  // BIT单点修改
  for (int i = x; i <= n; i += lowbit(i)) BIT[i] += c;
}

ll sum(int x) {  // BIT求a[1...x]的前缀和
  ll res = 0;
  for (int i = x; i; i -= lowbit(i)) res += BIT[i];
  return res;
}

int main() {
  for (int i = 2; i <= n; i++) cin >> a[i];

  for (int i = 1; i <= n; i++) BIT[i] = lowbit(i);  // 初始化BIT为1

  for (int i = n; i; i--) {  // 从后往前扫
    int k = a[i] + 1;  // 排第a[i]+1

    int l = 1, r = n;
    while (l < r) {
      int mid = l + r >> 1;
      if (sum(mid) >= k) r = mid;
      else l = mid + 1;
    }
    ans[i] = r;
    add(r, -1);  // 用过该身高
  }

  for (int i = 1; i <= n; i++) cout << ans[i] << endl;
}
```

17.1.5 Paimon Sorting

题意

有T组测试数据,每组测试数据给定一个长度为n $(1 \le n \le 1e5)$的整数序列a_i $(1 \le a_i \le n)$,用如下图所示的排序算法对该序列的每个非空前缀进行排序,分别输出排序过程中交换的次数.数据保证所有测试数据的n之和不超过1e6.

Algorithm 1 The Sorting Algorithm

```
function SORT(A)
    for i ← 1 to n do                ▷ n is the number of elements in A
        for j ← 1 to n do
            if a_i < a_j then        ▷ a_i is the i-th element in A
                Swap a_i and a_j
            end if
        end for
    end for
end function
```

思路

把该排序算法的过程打出来找规律,代码:

```cpp
const int MAXN = 1e5 + 5;
int n;  // 序列长度
int a[MAXN];  // 原数组

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  int ans = 0;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
```

```
      if (a[i] < a[j]) {
        printf("swaped a[%d]=%d and a[%d]=%d\n", i, a[i], j, a[j]);
        swap(a[i], a[j]);
        ans++;
      }

      printf("[%d,%d]  ", i, j);
      for (int k = 1; k <= n; k++) cout << a[k] << ' ';
      cout << endl << endl;
    }
  }
  cout << ans;
}
```

测试数据|输入:

```
5
4 3 2 1 5
```

测试数据|输出:

```
[1,1]  4 3 2 1 5

[1,2]  4 3 2 1 5

[1,3]  4 3 2 1 5

[1,4]  4 3 2 1 5

swaped a[1]=4 and a[5]=5
[1,5]  5 3 2 1 4  // a[2]=3前有1个数比它大

// 第一趟排序把5换到了序列的开头
// 把4换到了序列的末尾

swaped a[2]=3 and a[1]=5
[2,1]  3 5 2 1 4

[2,2]  3 5 2 1 4

[2,3]  3 5 2 1 4

[2,4]  3 5 2 1 4

[2,5]  3 5 2 1 4  // a[3]=2前有2个数比它大

// 第二趟排序把5后移了一位
// 把3换到了序列的开头

swaped a[3]=2 and a[1]=3
[3,1]  2 5 3 1 4

swaped a[3]=3 and a[2]=5
[3,2]  2 3 5 1 4

[3,3]  2 3 5 1 4

[3,4]  2 3 5 1 4

[3,5]  2 3 5 1 4  // a[4]=1前有3个数比它大

// 第三趟排序把5后移了一位
// 把2换到序列的开头

swaped a[4]=1 and a[1]=2
[4,1]  1 3 5 2 4

swaped a[4]=2 and a[2]=3
[4,2]  1 2 5 3 4

swaped a[4]=3 and a[3]=5
[4,3]  1 2 3 5 4

[4,4]  1 2 3 5 4

[4,5]  1 2 3 5 4  // a[5]=4前有1个数比它大

// 第四趟排序把5后移了一位
// 把1换到序列的开头

```

```
[5,1]  1 2 3 5 4

[5,2]  1 2 3 5 4

[5,3]  1 2 3 5 4

swaped a[5]=4 and a[4]=5
[5,4]  1 2 3 4 5

[5,5]  1 2 3 4 5

// 第五趟排序把5后移了一位

8
```

可以发现,每趟排序后,序列的最大、次大、第三大等依次被换到序列开头,且排好的序列逐步后移.观察知,从第二趟排序开始,第$i\ \ (2 \leq i \leq n)$趟排序所需的交换次数等于第$(i-1)$趟排序后的序列中$a[i]$前比它大的数的个数,可用BIT来维护.

上述测试数据中无重复数字,下面讨论有重复数字的情况.

测试数据II输入:

```
5
2 3 2 1 5
```

测试数据II输出:

```
[1,1]  2 3 2 1 5

swaped a[1]=2 and a[2]=3
[1,2]  3 2 2 1 5

[1,3]  3 2 2 1 5

[1,4]  3 2 2 1 5

swaped a[1]=3 and a[5]=5
[1,5]  5 2 2 1 3  // a[2]=2前有1个数比它大

swaped a[2]=2 and a[1]=5
[2,1]  2 5 2 1 3

[2,2]  2 5 2 1 3

[2,3]  2 5 2 1 3

[2,4]  2 5 2 1 3

[2,5]  2 5 2 1 3

[3,1]  2 5 2 1 3  // a[3]=2前有一个数比它大

swaped a[3]=2 and a[2]=5
[3,2]  2 2 5 1 3

[3,3]  2 2 5 1 3

[3,4]  2 2 5 1 3

[3,5]  2 2 5 1 3  // a[4]=1前有3个数比它大,但其中有两个数相等,第4躺排序只交换了2次

swaped a[4]=1 and a[1]=2
[4,1]  1 2 5 2 3

[4,2]  1 2 5 2 3

swaped a[4]=2 and a[3]=5
[4,3]  1 2 2 5 3

[4,4]  1 2 2 5 3

[4,5]  1 2 2 5 3  // a[5]=3前有1个数比它大

[5,1]  1 2 2 5 3

[5,2]  1 2 2 5 3

[5,3]  1 2 2 5 3

swaped a[5]=3 and a[4]=5
[5,4]  1 2 2 3 5
```

```
[5,5]  1 2 2 3 5

7
```

观察知,上述结论中交换次数等于前面比它大的数的个数应去重.

i从2开始从左往右扫一遍$a[i]$,遇到$a[i] > a[1]$时交换并$ans++$,每一趟排序$ans+=$BIT的前缀和,即$a[i]$前比其大的数的个数.

用这种思路模拟输入数据II:

```
原序列 2 3 2 1 5
a[2] = 3 > 2 = a[1]  // ans++  swap(a[1], a[2]) = swap(2, 3)
新序列 3 2 2 1 5  // a[2]前有1个比它大的数,ans += 1  swap(a[2], a[1]) = swap(2, 3)
                  // a[3]前有1个比它大的数,ans += 1  swap(a[1], a[3]) = swap(2, 3)
a[5] = 5 > 3 = a[1]  // ans++  swap(a[1], a[5]) = swap(3, 5)
新序列 5 2 2 1 3  // a[5]前有1个比它大的数,ans+=1  swap(a[1], a[5]) = swap(5, 3)

得到的ans = 5 < 7  // 少了swap(2, 1)和swap(2, 5)
```

从左往右扫一遍的过程中遇到$a[i] = a[1]$时,将$flag$置为$true$.当$flag = true$时,若$a[i]$后$\exists a[j] > a[1]$,则答案还需加上$a[i]$和$a[j]$间的数的个数cnt.

代码

```cpp
const int MAXN = 1e5 + 5;
int n;  // 序列长度
int a[MAXN];  // 原数组
int BIT[MAXN];  // 树状数组
bool vis[MAXN];  // 记录某个数是否已加入集合,用于去重

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

void add(int x) { for (int i = x; i <= n; i += lowbit(i)) BIT[i]++; }  // 将数加入集合(单点修改)

int sum(int x) {  // 求a[1...x]的前缀和
  int res = 0;
  for (int i = x; i; i -= lowbit(i)) res += BIT[i];
  return res;
}

int main() {
  CaseT{
    memset(BIT, 0, so(BIT));
    memset(vis, false, so(vis));

    cin >> n;
    int maxnum = 0;  // a[i]中的最大值
    for (int i = 1; i <= n; i++) {
      cin >> a[i];
      maxnum = max(maxnum, a[i]);
    }

    ll ans = 0;
    cout << ans;  // 长度为1的前缀无需交换

    bool flag = false;  // 记录a[i]后是否出现与a[1]相等的元素
    int cnt = 0;  // 在flag=1时,若a[i]后存在一个>a[1]的a[j],则cnt统计a[i]与a[j]间的数的个数
    vis[a[1]] = true, add(a[1]);  // a[1]加入集合
    for (int i = 2; i <= n; i++) {
      if (!vis[a[i]]) vis[a[i]] = true, add(a[i]);  // 去重后加入集合

      if (a[i] == a[1]) flag = true;  // 出现与a[1]相等的元素

      cnt += flag - (flag ? a[i] > a[1] : 0);  // flag=0时cnt不变,flag=1且a[i]<=a[1]时cnt++

      if (a[i] > a[1]) {  // 原a[i]==a[1]的位置后存在a[j]>a[1],需增加对答案的贡献
        swap(a[1], a[i]);
        ans += 1 + cnt;  // 交换次数为中间的数的个数+1(交换a[1]和a[j])
        cnt = flag = 0;  // 清空
      }

      ans += sum(a[1]) - sum(a[i]);  // a[i]前比其大的数的个数
      cout << ' ' << ans;
    }
    cout << endl;
  }
}
```

17.1.6 逆序对

题意

给定一个长度为n $(1 \leq n \leq 5e5)$的序列$a_1, \cdots, a_n$ $(1 \leq a_i \leq 1e9)$,求逆序对数.

代码

```
const int MAXN = 5e5 + 5;
int n;
int a[MAXN];
vi dis;

struct FenwickTree {
  int n;  // 数组长度
  ll a[MAXN];  // 原数组,下标从1开始
  ll BIT1[MAXN];  // 维护差分数组diff[]的前缀和
  ll BIT2[MAXN];  // 维护i*diff[i]的前缀和

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

  void add(ll BIT[], int x, ll c) {  // BIT[x]+=c
    for (int i = x; i <= n; i += lowbit(i)) BIT[i] += c;
  }

  ll sum(ll BIT[], int x) {  // 求[1...x]的前缀和
    ll res = 0;
    for (int i = x; i; i -= lowbit(i)) res += BIT[i];
    return res;
  }

  ll get_pre(int x) {  // 求a[1...x]的前缀和
    return sum(BIT1, x) * (x + 1) - sum(BIT2, x);
  }

  void modify(int l, int r, ll c) {  // 区间修改:a[l...r]+=c
    add(BIT1, l, c), add(BIT2, l, c * l);
    add(BIT1, r + 1, -c), add(BIT2, r + 1, -c * (r + 1));
  }

  ll query(int l, int r) {  // 区间查询:求a[l...r]的和
    return get_pre(r) - get_pre(l - 1);
  }

  void build() {  // 对原数组建BIT
    for (int i = 1; i <= n; i++) {
      ll diff = a[i] - a[i - 1];
      add(BIT1, i, diff), add(BIT2, i, diff * i);
    }
  }
}bit;

void solve() {
  cin >> n;
  bit.n = n;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
    dis.push_back(i);
  }

  sort(all(dis), [&](int x, int y) {
    return a[x] != a[y] ? a[x] > a[y] : x > y;
    });

  ll ans = 0;
  for (int i = 0; i < n; i++) {
    ans += bit.query(1, dis[i]);
    bit.modify(dis[i], dis[i], 1);
  }
  cout << ans;
}

int main() {
  solve();
}
```

17.1.6 Fenwick Tree

原题指路:https://codeforces.com/gym/103861/problem/L

题意

给定一个长度为n的序列$c_1,\cdots,c_n$,初始时$c_i=0\ \ (1\le i\le n)$.

定义函数:

```
def update(pos, val):
  while (pos <= n):
    c[pos] += val
    pos += pos & (-pos)
```

现调用若干次update(函数),其中的val可为任意实数.给定一个长度为n的$0/1$串$s_1\cdots s_n$描述操作后序列$c[]$,其中$s_i=0$表示$c_i=0$,$s_i=1$表示$c_i\neq 0$.问使得序列$c[]$满足s所需调用update()的最小次数.

有$t\ \ (1\le t\le 1e5)$组测试数据.每组测试数据第一行输入一个整数$n\ \ (1\le n\le 1e5)$.第二行输入一个长度为n的$0/1$串$s_1\cdots s_n$.数据保证所有测试数据的n之和不超过$1e6$.

思路

修改下标i的同时会修改下标$i+lowbit(i),i+2\cdot lowbit(i),\cdots$.注意到当前下标$i$的操作$+A$的影响一定会传给下标$i+lowbit(i)$,但未必会继续下传给$i+2\cdot lowbit(i)$,如下标$i+lowbit(i)$也修改操作$-A$时,下标$i$的操作的影响等价于不再下传,故下标$i$的下下个位置$i+2\cdot lowbit(i)$是否需要修改取决于下标$i+lowbit(i)$的位置的修改,故对每个$s_i=1$,只需考虑下标$i$对下标$i+lowbit(i)$的影响,

称使得$j+lowbit(j)=i$的所有下标j构成的集合为下标i的上一个位置.$cnt[i]$表示下标i的上一个位置的集合中被操作过的下标的个数,显然最小操作次数是如下两种情况的操作次数之和:

①对$s_i=0,cnt_i=1$的位置,需在下标i处操作一次,使得c_i变为0.

②对$s_i=1,cnt_i=0$的位置,需在下标i处操作一次,使得c_i非零.

代码

```
void solve() {
  int n; string s; cin >> n >> s;
  s = " " + s;

  vector<int> cnt(n + 1);  // cnt[i]表示下标i被操作的次数
  for (int i = 1; i <= n; i++) {
    if (s[i] == '1' && i + lowbit(i) <= n)
      cnt[i + lowbit(i)]++;
  }

  int ans = 0;
  for (int i = 1; i <= n; i++)
    ans += (s[i] == '0' && cnt[i] == 1) || (s[i] == '1' && cnt[i] == 0);
  cout << ans << endl;
}

int main() {
  CaseT
    solve();
}
```

17.1.7 Propagating tree

原题指路:https://codeforces.com/problemset/problem/383/C

题意 (2 s)

给定一棵包含编号$1\sim n$的n个节点的树,其中1号节点为根节点.每个节点有一个权值,初始时i号节点的权值为a_i.

现有如下两种操作:

①$1\ u\ val$,表示令节点u的权值$+=val$.特别地,若节点u的权值$+=val$,则其所有儿子节点的权值$-=val$,该过程是递归的.

②$2\ u$,表示询问节点u的权值.

第一行输入两个整数$n,m\ \ (1\le n,m\le 2e5)$,分别表示节点数、询问数.第二行输入n个整数$a_1,\cdots,a_n\ \ (1\le a_i\le 1000)$.接下来$(n-1)$行每行输入两个整数$u,v\ \ (1\le u,v\le n,u\neq v)$,表示节点$u$与节点$v$间存在无向边.接下来$m$行每行输入一个询问,格式如上,其中$1\le u\le n,1\le val\le 1000$.数据保证给定的边构成一棵树.

思路I

因只涉及到子树的修改,考虑求出该树的DFS序后用BIT或线段树维护.因转化后只涉及区间修改和单点查询,可用差分和BIT维护.

对操作①,可用两棵BIT分别维护原树奇数、偶数深度的节点.

代码I

```cpp
const int MAXN = 2e5 + 5, MAXM = MAXN << 1;
namespace DFSOrderFenwickTree {
  int n;
  int a[MAXN];
  int head[MAXN], edge[MAXM], nxt[MAXM], idx;
  int dfn[MAXN], idfn[MAXN], tim;  // 节点的DFS序、DFS序对应的节点、时间戳
  bool depth[MAXN];  // 节点的深度的奇偶性
  int siz[MAXN];  // siz[u]表示以u为根节点的子树的大小
  int BIT[2][MAXN];  // BIT[0/1]维护奇数/偶数深度的节点的点权的差分数组diff[]

  void addEdge(int a, int b) {
    edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
  }

  void dfs(int u, int fa) {
    dfn[u] = ++tim, siz[u] = 1, depth[u] = depth[fa] ^ 1;
    for (int i = head[u]; ~i; i = nxt[i]) {
      int v = edge[i];
      if (v != fa) {
        dfs(v, u);
        siz[u] += siz[v];
      }
    }
  }

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

  void add(bool op, int pos, int v) {  // 令op深度的BIT中,DFS序为pos的节点的diff[]值+=v
    for (int i = pos; i <= n; i += lowbit(i)) BIT[op][i] += v;
  }

  int query(bool op, int pos) {  // 查询op深度的BIT中,DFS序为pos的节点的权值的增量
    int res = 0;
    for (int i = pos; i; i -= lowbit(i)) res += BIT[op][i];
    return res;
  }

  void modify(int u, int v) {  // 令节点u的点权+=v
    add(depth[u], dfn[u], v), add(depth[u], dfn[u] + siz[u], -v);
  }
}
using namespace DFSOrderFenwickTree;

void solve() {
  memset(head, -1, sizeof(head));

  int m; cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    addEdge(u, v), addEdge(v, u);
  }

  dfs(1, 0);

  while (m--) {
    int op, u; cin >> op >> u;
    if (op == 1) {
      int val; cin >> val;
      modify(u, val);
    }
    else cout << a[u] + query(depth[u], dfn[u]) - query(depth[u] ^ 1, dfn[u]) << endl;
  }
}

int main() {
  solve();
}
```

思路II

对操作①,考虑将修改相同的节点分别放在连续的区间,具体地,考察每个节点在整棵树中的深度,记录两次DFS序,分别将深度为奇数、偶数的节点放在一个连续区间.

考察连续区间的起点:①对奇数深度的情况,对每个节点u,记录以u为根节点的子树中DFS序最小的奇数深度的节点son_u;②对偶数深度的情况,对每个节点u,记录以u为根节点的子树中DFS序最小的偶数深度的节点son_u.

考察连续区间的大小:对每个节点u,记录以u为根节点的子树中深度的奇偶性与u相同的节点数$siz_{0,u}$和深度的奇偶性与u不同的节点数$siz_{1,u}$.

修改以u为根节点的子树时:①对节点u和与u深度的奇偶性相同的节点,修改区间$[dfn_u, dfn_u + siz_{0,u} - 1]$;②对于节点$u$深度的奇偶性不同的节点,修改区间$[dfn_{son_u}, dfn_{son_u} + siz_{1,u} - 1]$.

用差分的BIT实现区间修改和单点查询即可.

代码II

```cpp
const int MAXN = 2e5 + 5, MAXM = MAXN << 1;
namespace DFSOrderFenwickTree {
  int n;
  int a[MAXN];
  int head[MAXN], edge[MAXM], nxt[MAXM], idx;
  int dfn[MAXN], idfn[MAXN], tim;  // 节点的DFS序、DFS序对应的节点、时间戳
  int depth[MAXN];  // 节点的深度
  int siz[2][MAXN];  // siz[0/1][u]表示以u为根节点的子树中奇数/偶数深度的节点数
  int son[MAXN];  // son[u]表示以u为根节点的子树中DFS序最小的对应奇偶性的深度的节点
  int BIT[MAXN];  // BIT维护点权的差分数组diff[]

  void addEdge(int a, int b) {
    edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
  }

  void dfs1(int u, int fa) {  // 预处理奇数深度的节点的DFS序
    depth[u] = depth[fa] + 1;
    if (depth[u] & 1) {  // 节点u的深度为奇数
      dfn[u] = ++tim, idfn[tim] = u, siz[0][u] = 1;
      for (int i = head[u]; ~i; i = nxt[i]) {
        int v = edge[i];  // 节点v的深度为偶数
        if (v != fa) {
          dfs1(v, u);
          siz[0][u] += siz[1][v];
        }
      }
    }
    else {  // 节点u的深度为偶数
      for (int i = head[u]; ~i; i = nxt[i]) {
        int v = edge[i];  // 节点v的深度为奇数
        if (v != fa) {
          dfs1(v, u);
          if (!son[u]) son[u] = v;  // 记录DFS序最小的偶数深度的节点
          siz[1][u] += siz[0][v];
        }
      }
    }
  }

  void dfs2(int u, int fa) {  // 预处理偶数深度的节点的DFS序
    depth[u] = depth[fa] + 1;
    if (depth[u] & 1) {  // 节点u的深度为奇数
      for (int i = head[u]; ~i; i = nxt[i]) {
        int v = edge[i];  // 节点v的深度为偶数
        if (v != fa) {
          dfs2(v, u);
          if (!son[u]) son[u] = v;  // 记录DFS序最小的奇数深度的节点
          siz[1][u] += siz[0][v];
        }
      }
    }
    else {  // 节点u的深度为偶数
      dfn[u] = ++tim, idfn[tim] = u, siz[0][u] = 1;
      for (int i = head[u]; ~i; i = nxt[i]) {
        int v = edge[i];  // 节点v的深度为奇数
        if (v != fa) {
          dfs2(v, u);
          siz[0][u] += siz[1][v];
        }
      }
    }
  }

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

  void add(int pos, int v) {  // 令DFS序为pos的节点的diff[]值+=v
    for (int i = pos; i <= n; i += lowbit(i)) BIT[i] += v;
  }

  int query(int pos) {  // 查询DFS序为pos的节点的权值
    int res = 0;
```

```
    for (int i = pos; i; i -= lowbit(i)) res += BIT[i];
    return res;
  }

  void init() {
    dfs1(1, 0);
    dfs2(1, 0);

    for (int i = 1; i <= n; i++)  // BIT维护点权的差分
      add(dfn[i], a[i] - a[idfn[dfn[i] - 1]]);
  }

  void modify(int l, int r, int v) {  // 令DFS序在区间[l,r]中的节点的点权+=v
    add(l, v), add(r + 1, -v);
  }
}
using namespace DFSOrderFenwickTree;

void solve() {
  memset(head, -1, sizeof(head));

  int m; cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    addEdge(u, v), addEdge(v, u);
  }

  init();

  while (m--) {
    int op, u; cin >> op >> u;
    if (op == 1) {
      int val; cin >> val;

      modify(dfn[u], dfn[u] + siz[0][u] - 1, val);  // 操作与u的深度奇偶性相同的节点
      if (son[u])  // u是树中节点
        modify(dfn[son[u]], dfn[son[u]] + siz[1][u] - 1, -val);  // 操作与u的深度奇偶性不同的节点
    }
    else cout << query(dfn[u]) << endl;
  }
}

int main() {
  solve();
}
```

思路III

用本题的树剖做法的思想,用节点在整棵树中的深度的奇偶性来统一节点权值的修改与深度的奇偶性的关系,即修改和查询时,奇数深度的节点的权值统一$+= val$,偶数深度的节点的权值统一$-= val$,这样用一个BIT即可维护.

代码III

```
const int MAXN = 2e5 + 5, MAXM = MAXN << 1;
namespace DFSOrderFenwickTree {
  int n;
  int a[MAXN];
  int head[MAXN], edge[MAXM], nxt[MAXM], idx;
  int dfn[MAXN], idfn[MAXN], tim;  // 节点的DFS序、DFS序对应的节点、时间戳
  bool depth[MAXN];  // 节点的深度的奇偶性
  int siz[MAXN];  // siz[u]表示以u为根节点的子树的大小
  int BIT[MAXN];  // BIT维护点权的差分数组diff[]

  void addEdge(int a, int b) {
    edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
  }

  void dfs(int u, int fa) {
    dfn[u] = ++tim, siz[u] = 1, depth[u] = depth[fa] ^ 1;
    for (int i = head[u]; ~i; i = nxt[i]) {
      int v = edge[i];
      if (v != fa) {
        dfs(v, u);
        siz[u] += siz[v];
      }
    }
  }

  int lowbit(int x) {
```

```
    return x & -x;
  }

  void add(int pos, int v) {  // 令DFS序为pos的节点的diff[]值+=v
    for (int i = pos; i <= n; i += lowbit(i)) BIT[i] += v;
  }

  int query(int pos) {  // 查询DFS序为pos的节点的权值的增量
    int res = 0;
    for (int i = pos; i; i -= lowbit(i)) res += BIT[i];
    return res;
  }

  void modify(int l, int r, int v) {  // 令DFS序在区间[l,r]中的节点的点权+=v
    add(l, v), add(r + 1, -v);
  }
}
using namespace DFSOrderFenwickTree;

void solve() {
  memset(head, -1, sizeof(head));

  int m; cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    addEdge(u, v), addEdge(v, u);
  }

  dfs(1, 0);

  while (m--) {
    int op, u; cin >> op >> u;
    if (op == 1) {
      int val; cin >> val;
      modify(dfn[u], dfn[u] + siz[u] - 1, depth[u] & 1 ? val : -val);
    }
    else {
      int delta = query(dfn[u]);
      cout << a[u] + (depth[u] & 1 ? delta : -delta) << endl;
    }
  }
}

int main() {
  solve();
}
```

17.1.8 Range Update Point Query

原题指路:https://codeforces.com/contest/1791/problem/F

题意 (2 s)

维护一个长度为n的序列$a_1,\cdots,a_n$,支持如下两个操作:

①1 l r,表示对$\forall i\in[l,r]$ $(1\le l\le r\le n)$,将a_i置为其数位和.

②2 x,表示输出a_x $(1\le x\le n)$.

有t $(1\le t\le 1000)$组测试数据.每组测试数据第一行输入两个整数n,q $(1\le n,q\le 2e5)$.第二行输入n个整数$a_1,\cdots,a_n$ $(1\le a_i\le 1e9)$.接下来q行每行输入一个操作,格式如上.数据保证所有测试数据的n之和、q之和都不超过2e5.

思路I

[**引理**] 将一个非负整数x一直置为其数位和后该数不增,最后会得到一个一位数,该一位数称为x的digit root.

[**证**] 注意到"不增"中等号取得当且仅当该数是一位数即证.

[**注**] 非负整数x的digit root为$(x-1)\bmod 9+1$.

[**定理**] 对一个1e9内的非负整数,至多进行3次求数位和操作后数位和不变.

[**证**] 1e9内数位和最大的数为$(1e9-1)$,数位和为$9\times 9=81$.

81以内数位和最大的数为79,数位和为16.

16以内数位和最大的数为9,数位和为9,之后数位和不变.

用树状数组维护每个下标被覆盖的次数后,查询时暴力修改欲查询的下标被覆盖的次数次,数位和不变时打上标记即可.

时间复杂度$O\left(n+q\log n+3n\right)=O(n+q\log n)$.

代码I

```cpp
template<typename T>
struct FenwickTree {
  int n;  // 元素个数
  vector<T> BIT;  // BIT维护原数组a[]的差分数组diff[]

  FenwickTree(int _n) :n(_n) {
    BIT.resize(n + 2);
  }

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

  void add(int x, int c) {  // diff[x]+=c
    for (int i = x; i <= n; i += lowbit(i))
      BIT[i] += c;
  }

  void modify(int l, int r, int c) {  // a[l...r]+=c
    add(l, c), add(r + 1, -c);
  }

  T query(int x) {  // 求a[x]
    T res = 0;
    for (int i = x; i; i -= lowbit(i)) res += BIT[i];
    return res;
  }
};

int get(int x) {  // 求数x的数位和
  int res = 0;
  for (; x; x /= 10) res += x % 10;
  return res;
}

void solve() {
  int n, q; cin >> n >> q;
  vector<int> a(n + 1);
  vector<bool> ok(n + 1);  // 记录下标的数位和是否已不变
  for (int i = 1; i <= n; i++) {
    cin >> a[i];

    if (a[i] < 10) ok[i] = true;
  }

  FenwickTree<int> bit(n);
  while (q--) {
    int op; cin >> op;
    if (op == 1) {
      int l, r; cin >> l >> r;
      bit.modify(l, r, 1);
    }
    else {
      int pos; cin >> pos;

      if (ok[pos]) cout << a[pos] << endl;
      else {
        int cnt = bit.query(pos);  // 该下标被覆盖的次数
        for (int i = 0; i < cnt; i++) {
          int tmp = get(a[pos]);
          if (tmp == a[pos]) {
            ok[pos] = 1;
            break;
          }
          else a[pos] = tmp;
        }
        cout << a[pos] << endl;
        bit.modify(pos, pos, -cnt);  // 清空
      }
    }
  }
}

int main() {
  CaseT
    solve();
```

```
}
```

思路II

因每个元素至多被修改3次,用set维护修改次数不足的下标即可.

时间复杂度$O(q+3n\log n)=O(q+n\log n)$.

代码II

```
int get(int x) {  // 求数x的数位和
  int res = 0;
  for (; x; x /= 10) res += x % 10;
  return res;
}

void solve() {
  int n, q; cin >> n >> q;
  vector<int> a(n + 1);
  set<int> s;  // 存修改次数不足的下标
  for (int i = 1; i <= n; i++) {
    cin >> a[i];

    if (a[i] >= 10) s.insert(i);
  }

  while (q--) {
    int op; cin >> op;
    if (op == 1) {
      int l, r; cin >> l >> r;

      vector<int> ids;
      for (auto it = s.lower_bound(l); it != s.end() && *it <= r; s.erase(it++))
        ids.push_back(*it);

      for (auto idx : ids)
        if ((a[idx] = get(a[idx])) >= 10) s.insert(idx);
    }
    else {
      int pos; cin >> pos;
      cout << a[pos] << endl;
    }
  }
}

int main() {
  CaseT
    solve();
}
```

17.1.9 Nested Segments

原题指路: https://codeforces.com/problemset/problem/652/D

题意 (2 s)

给定数轴上的n $(1\le n\le 2e5)$条线段$[l_i,r_i]$ $(-1e9\le l_i\le r_i\le 1e9)$. 对每条线段, 求有多少条其他线段包含于该线段.

思路I

若线段$[l_i,r_i]$包含线段$[l_j,r_j]$, 则$l_i\le l_j,r_i\ge r_j$. 将线段按r_i非降序排列后, 对每条线段i, 需求出 $s.t.\ j<i$且$l_j\ge l_i$的下标j的个数, 即求l值的非严格逆序对数, 这可用BIT实现. 坐标值域较大, 需将l离散化后再插入 BIT.

时间复杂度$O(n\log n)$.

代码I

```
#define l first
#define r second
typedef pair<pair<int, int>, int> F;  // 区间端点、下标

template<typename T>
struct FenwickTree {
    int n;
    vector<T> BIT;  // BIT维护原数组a[]

    FenwickTree(int _n) :n(_n) {
        BIT.resize(n + 2);
```

```
    }

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

    void add(int x, int c) {  // a[x]+=c
        for (int i = x; i <= n; i += lowbit(i))
            BIT[i] += c;
    }

    T query(int x) {  // 求a[x]
        T res = 0;
        for (int i = x; i; i -= lowbit(i)) res += BIT[i];
        return res;
    }
};

void solve() {
    int n; cin >> n;
    vector<F> a(n + 1);
    vector<int> dis;  // 离散化数组
    for (int i = 1; i <= n; i++) {
        int l, r; cin >> l >> r;

        a[i] = { { l, r }, i };
        dis.push_back(l);
    }
    sort(a.begin() + 1, a.end(), [&](const F& A, const F& B) {
        return A.first.r <= B.first.r;
    });

    // 离散化
    sort(all(dis));
    dis.erase(unique(all(dis)), dis.end());
    int siz = dis.size();
    for (int i = 1; i <= n; i++)
        a[i].first.l = lower_bound(all(dis), a[i].first.l) - dis.begin() + 1;

    vector<int> ans(n + 1);
    FenwickTree<int> bit(siz + 1);
    for (int i = 1; i <= n; i++) {
        int l = a[i].first.l, idx = a[i].second;
        ans[idx] = bit.query(siz) - bit.query(l - 1);
        bit.add(l, 1);
    }

    for (int i = 1; i <= n; i++)
        cout << ans[i] << endl;
}

int main() {
    solve();
}
```

思路II

同**思路I**, 但不将坐标离散化, 而用动态开点的权值线段树求逆序对.

时间复杂度$O(n \log n)$.

代码II

```
#define l first
#define r second
typedef pair<pair<int, int>, int> F;  // 区间端点、下标

const int MAXN = 2e5 + 5;
const int MAXA = 1e9 + 5;

struct WeightedSegmentTree {
    struct Node {
        int l, r;
        int cnt;
    };
    vector<Node> SegT;
    int root = 1;
    int idx = 1;

    WeightedSegmentTree(int _n) {
```

```
        SegT.resize(_n * ((int)log2(_n) + 1));
    }

    void insert(int& u, int l, int r, int x) {
        if (!u) {
            u = ++idx;
            SegT[u].l = SegT[u].r = SegT[u].cnt = 0;
        }
        SegT[u].cnt++;

        if (l == r) return;

        int mid = l + r >> 1;
        if (x <= mid) insert(SegT[u].l, l, mid, x);
        else insert(SegT[u].r, mid + 1, r, x);
    }

    void insert(int x) {
        insert(root, -MAXA, MAXA, x);
    }

    int query(int u, int l, int r, int L, int R) {
        if (!u) return 0;
        if (L <= l && r <= R) return SegT[u].cnt;

        int mid = l + r >> 1;
        int res = 0;
        if (L <= mid) res += query(SegT[u].l, l, mid, L, R);
        if (R > mid) res += query(SegT[u].r, mid + 1, r, L, R);
        return res;
    }

    int query(int L, int R) {
        return query(root, -MAXA, MAXA, L, R);
    }
};

void solve() {
    int n; cin >> n;
    vector<F> a(n + 1);
    vector<int> dis;  // 离散化数组
    for (int i = 1; i <= n; i++) {
        int l, r; cin >> l >> r;

        a[i] = { { l, r }, i };
        dis.push_back(l);
    }
    sort(a.begin() + 1, a.end(), [&](const F& A, const F& B) {
        return A.first.r <= B.first.r;
    });

    vector<int> ans(n + 1);
    WeightedSegmentTree st(MAXN);
    for (int i = 1; i <= n; i++) {
        int l = a[i].first.l, idx = a[i].second;
        ans[idx] = st.query(l, MAXA);
        st.insert(l);
    }

    for (int i = 1; i <= n; i++)
        cout << ans[i] << endl;
}

int main() {
    solve();
}
```

17.1.10 一个简单的整数问题

题意

给定长度为n $(1 \le n \le 1e5)$的数列$a[]$(其中元素的绝对值$\le$ 1e9)和m $(1 \le m \le 1e5)$个操作, 操作命令有如下两种类型:

①$C\ l\ r\ d$, 表示将$a[l, r]$中的每个数加d $(|d| \le 1e4)$.

②$Q\ x$, 表示输出第x个数的值.

思路

①操作为区间修改, 可先预处理出原数组$a[]$的差分数组$diff[]$, 则$a[l\cdots r]$中每个数加d只需$diff[l]+=c, diff[r+1]-=c$,将区间修改转化为两个单点修改, 故可直接用$diff[]$建BIT.

②操作查询某个$a[x]$时, 只需求一次差分数组的前缀和.

代码

```cpp
template<class T>
struct FenwickTree {
    int n;
    vector<T> BIT;  // BIT维护diff[]

    FenwickTree(int _n) :n(_n) {
        BIT.resize(n + 2);
    }

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

    void add(int pos, int val) {  // diff[pos] += val
        for (int i = pos; i <= n; i += lowbit(i))
            BIT[i] += val;
    }

    void modify(int l, int r, int val) {  // a[l...r] += val
        add(l, val), add(r + 1, -val);
    }

    T sum(int pos) {  // diff[1...pos]之和, 即a[pos]
        T res = 0;
        for (int i = pos; i; i -= lowbit(i))
            res += BIT[i];
        return res;
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    FenwickTree<ll> bit(n);
    for (int i = 1; i <= n; i++)
        bit.add(i, a[i] - a[i - 1]);

    while (m--) {
        char op; cin >> op;
        if (op == 'C') {
            int l, r, d; cin >> l >> r >> d;
            bit.modify(l, r, d);
        }
        else {
            int pos; cin >> pos;
            cout << bit.sum(pos) << endl;
        }
    }
}

int main() {
    solve();
}
```

17.2 线段树

线段树主要有两个操作:

①push_up:由父节点求子节点的信息.

②push_down(lazy tag):将父节点的修改信息下传到子节点.

③build:将一段区间建线段树.

④modify:单点修改或区间修改.

⑤query:查询一段区间的信息.

线段树除最后一层节点外是满二叉树.以维护区间$[l,r]$的信息为例:根节点维护整个区间$[l,r]$的信息,其两个儿子节点分别维护$[l,mid]$和$[mid+1,r]$的信息,其中$mid=\left\lfloor\frac{l+r}{2}\right\rfloor$,同理继续递归直至每个区间长度都为1.线段树一般用类似于堆的存储方式,即用一维数组存树,即对节点x,其父亲节点为$\left\lfloor\frac{x}{2}\right\rfloor$,左儿子为$2x$,右儿子为$2x+1$.

对一个长度为n的区间,叶子节点有n个,则整棵树除最后一层外约$(2n-1)$个节点,最后一层最坏有$2n$个节点,则最坏有$(4n-1)$个节点,故线段树一般开区间长度4倍的空间.

线段树每个节点一般是一个结构体,除了记录区间的左右端点外,还需记录询问的信息和辅助信息,即需保证某区间的属性能用其两个子区间的属性更新,亦即保证维护的信息具有完备性.

查询区间$[l,r]$的信息时,设涉及的树中结点的范围为$[T_l,T_r]$.

①若$[T_l,T_r]\subset[l,r]$,则直接返回$[T_l,T_r]$.

②若$[T_l,T_r]$与$[l,r]$相交,若左边相交则递归到左边,若右边相交则递归到右边.

③不存在$[T_l,T_r]\bigcap[l,r]=\varnothing$的情况,因为查询的区间必在根节点维护的区间内,且每次递归都有交集.

下证进行查询操作时至多访问的区间数正比于$\log_2 n$:

1.若为上述①的情况,则不用往下递归.

2.若为上述②的情况,记$[T_l,T_r]$的中点为m:

(1)$T_l\le l\le T_r\le r$时,若$l>m$,则只需递归右边;若$l\le m$,则需递归两边.但递归右边的情况的下一层必为①的情况,即只需递归1次,不会走到底部,

(2)$l\le T_l\le r\le T_r$时与(1)同理.

(3)$[l,r]\subset[T_l,T_r]$时,①若$r\le m$,则只递归左边;②若$l>m$,则只递归右边;③$l\le m<r$,则递归两边.但③的情况递归左边时,$[l,r]$有一部分不在$[T_l,m]$中,则递归的下一次就不是该情况.

综上,一棵树至多分裂出两条链,每条链长度与树的高度成正比,至多会访问$\lfloor 4\log_2 n\rfloor$个区间,故时间复杂度$O(\log n)$,但常数比BIT大.

若只涉及到单点修改,则无需添加懒标记也可保证线段树$O(\log n)$的时间复杂度,因为只有$\lfloor\log_2 n\rfloor$个只包含一个点的区间,最坏的情况下将包含该点所有区间都修改,至多修改$O(n)$个区间;若涉及区间修改,不添加懒标记可能使线段树时间复杂度退化,因为最坏的情况是修改整个序列,最多需要修改$(4*n-1)$个区间,时间复杂度$O(n)$.

添加lazy tag后线段树可实现$O(\log n)$区间修改,其思路来源于区间查询:与欲查询区间相关的区间有$O(n)$个,但若线段树的某个区间包含在欲查询区间中则直接返回,则最多会访问$O(4\log n)$个区间.类似地,区间修改中若线段树的某个区间包含在欲修改区间中则不继续往下传递,而是在该区间打上标记.同区间查询的证明知:这样至多修改$O(4\log n)$个区间,故时间复杂度$O(\log n)$.

以区间中所有数加一个数为例,线段树的节点记录区间左右端点l,r、只考虑当前节点及其子节点的懒标记(即不考虑当前节点的祖宗节点的标记)的当前区间和sum和懒标记$lazy$,其中$lazy$表示给以当前区间为根节点的子树的每个子区间都加一个数(不包含根节点的区间).

添加懒标记后进行区间查询时,需将欲查询区间相关的父区间的$lazy$值下传到相关子区间并清空父区间的$lazy$值,即push_down操作.更新方式:$ls.lazy+=lazy, ls.sum+=(ls.r-ls.l+1)*lazy; rs.lazy+=lazy, rs.sum+=(rs.r-rs.l+1)*lazy; lazy=0.$

添加懒标记后进行区间修改时也需下传$lazy$,即保证区间的$lazy$适用于整个区间.

17.2.1 区间最大值

题意

维护一个正整数序列$a[]$, 每个数都在$0\sim(p-1)$ $(1\le p\le 2e9)$间. 现进行m $(1\le m\le 2e5)$次操作, 操作命令有如下两种:

①A t, 表示向序列末尾添加一个数$(t+a)\%p$, 其中t $(0\le t<p)$为输入的参数, a是该添加操作前最后一个询问的答案, 初始时$a=0$.

②Q l, 输出序列中最后l个数的最大值.

初始时$a[]$为空.

数据保证第一个操作是添加操作; $L>0$且不超过当前序列的长度.

思路

先开m个位置, 再开一个变量idx记录当前已用了几个位置.

①操作, 在序列末尾添加一个数等价于将第$(idx+1)$个数修改为要添加的数.

②操作, 询问区间$[n-l+1,n]$的最大值.

代码

```
int MOD;

struct SegmentTree {
    int n;
    struct Node {
        int maxv;  // 区间最大值

        Node() :maxv(-INF) {}
    };
    vector<Node> SegT;
```

```cpp

    friend Node operator+(const Node A, const Node B) {
        Node res;
        res.maxv = max(A.maxv, B.maxv);
        return res;
    }

    SegmentTree(int _n) :n(_n) {
        SegT.resize(n + 1 << 2);
        build(1, 1, n);
    }

    void pushUp(int u) {
        SegT[u] = SegT[u << 1] + SegT[u << 1 | 1];
    }

    void build(int u, int l, int r) {
        if (l == r) return;

        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        // pushUp(u);  // 初始时序列各元素都为0, 无需pushUp
    }

    Node query(int u, int l, int r, int L, int R) {  // 求max a[L...R]
        if (R < l || L > r) return Node();
        if (L <= l && r <= R) return SegT[u];

        int mid = l + r >> 1;
        return query(u << 1, l, mid, L, R) + query(u << 1 | 1, mid + 1, r, L, R);
    }

    void modify(int u, int l, int r, int pos, int val) {  // a[pos] = val
        if (l == r && l == pos) {
            SegT[u].maxv = val;
            return;
        }

        int mid = l + r >> 1;
        if (pos <= mid) modify(u << 1, l, mid, pos, val);
        else modify(u << 1 | 1, mid + 1, r, pos, val);
        pushUp(u);
    }
};

void solve() {
    int m; cin >> m >> MOD;

    int idx = 0;  // 当前的序列长度
    int last = 0;  // 上个询问的答案

    SegmentTree st(m);

    for (int i = 0; i < m; i++) {
        char op; int x; cin >> op >> x;

        if (op == 'Q')
            cout << (last = st.query(1, 1, m, idx - x + 1, idx).maxv) << endl;
        else {
            st.modify(1, 1, m, idx + 1, ((ll)last + x) % MOD);
            idx++;
        }
    }
}

int main() {
    solve();
}
```

17.2.2 线段树模板I

题意

给定长度为n $(1 \le n \le 5e5)$的数组$a[]$和m $(1 \le m \le 1e5)$个操作,操作命令有如下两种:

①1 x y,输出区间$[x, y]$中最大连续子段和,即 $\max\limits_{x \le l \le r \le y} \left\{ \sum\limits_{i=l}^{r} a[i] \right\}$.

②2 x y,将$a[x]$修改为y.

数据保证$|a[x]| \leq 1000$.数据不保证①操作中$x \leq y$.

思路

线段树的节点需记录区间左右端点l, r、最大连续子段和$tmax$,但这样还不够,因为某区间的最大连续子段可能横跨左右儿子,故还需记录以左子区间的最大后缀、右子区间的最大前缀,这样横跨左右区间的最大连续字段和=左子区间的最大后缀+右子区间的最大前缀.节点再记录最大前缀和$lmax$、最大后缀和$rmax$,则由左右子区间更新父区间的方式为$tmax = \max\{ls.tmax, rs.tmax, ls.rmax + rs.lmax\}$.

但这还不够.以由左右子区间的最大前缀更新父区间的最大前缀为例,有两种情况:

①若父区间的最大前缀没超过左子区间,则$lmax = ls.lmax$.

②若父区间的最大前缀超过左子区间,$lmax = ls.sum + rs.lmax$,故节点还需记录区间和sum,区间和的更新方式为$sum = ls.sum + rs.sum$.

查询有三种情况:

①区间包含查询区间,则直接返回.

②查询区间只包含在左子区间或右子区间,则直接返回左子区间或右子区间.

③查询区间横跨左子区间或右子区间,则需合并,

代码

```cpp
const int MAXN = 5e5 + 5;
int n, m;  // 点数、操作数
int a[MAXN];  // 原数组
struct Node {
  int l, r;  // 区间左右端点
  int sum;  // 区间和
  int lmax;  // 最大前缀和
  int rmax;  // 最大后缀和
  int tmax;  // 最大连续子段和
}SegT[MAXN * 4];

void push_up(Node& u, Node& l, Node& r) {  // u为左右儿子节点的父亲节点
  u.sum = l.sum + r.sum;
  u.lmax = max(l.lmax, l.sum + r.lmax);
  u.rmax = max(r.rmax, r.sum + l.rmax);
  u.tmax = max(max(l.tmax, r.tmax), l.rmax + r.lmax);
}

void push_up(int u) {  // 更新节点u的左右儿子节点
  push_up(SegT[u], SegT[u << 1], SegT[u << 1 | 1]);
}

void build(int u, int l, int r) {  // 以u为根节点,将区间[l,r]建SegT
  if (l == r) SegT[u] = { l,r,a[r],a[r],a[r],a[r] };  // 叶子节点
  else {  // 树中节点
    SegT[u] = { l,r };
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    push_up(u);
  }
}

void modify(int u, int x, int v) {  // 从节点u开始,将第x个数修改为v
  if (SegT[u].l == x && SegT[u].r == x) SegT[u] = { x,x,v,v,v,v };  // 叶子节点
  else {
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (x <= mid) modify(u << 1, x, v);
    else modify(u << 1 | 1, x, v);
    push_up(u);
  }
}

Node query(int u, int l, int r) {  // 从节点u开始查询[l,r]的信息
  if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u];  // 区间包含[l,r]
  else {
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (r <= mid) return query(u << 1, l, r);  // 左子区间包含[l,r]
    else if (l > mid) return query(u << 1 | 1, l, r);  // 右子区间包含[l,r]
    else {  // [l,r]横跨左右子区间
      auto left = query(u << 1, l, r), right = query(u << 1 | 1, l, r);
      Node res;  // 存答案
      push_up(res, left, right);
      return res;
    }
  }
}

int main() {
```

```
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);  // 建SegT

  while (m--) {
    int k, x, y; cin >> k >> x >> y;
    if (k == 1) {
      if (x > y) swap(x, y);
      cout << query(1, x, y).tmax << endl;
    }
    else modify(1, x, y);
  }
}
```

17.2.3 区间最大公约数

题意

给定一个长度为n $(1 \le n \le 5e5)$的数列$a[]$.现有m $(1 \le m \le 1e5)$个操作,操作命令有如下两种:

①$C\ l\ r\ d$,表示将$a[l], \cdots, a[r]$的数都加d $(|d| \le 1e18)$.

②$Q\ l\ r$,输出$\gcd\{a[l], \cdots, a[r]\}$.

数据保证$1 \le |a[i]| \le 1e18$.

思路

本题只有区间整体加一个数,可用差分数组,不用lazy tag.

节点记录区间左右端点l, r,区间最大公约数d.但注意到无法从$\gcd\{a, b, c\}$推出$\gcd\{a+x, b+x, c+x\}$.注意到只修改一个数时易更新d,考虑将区间修改转化为单点修改.同样考虑差分,注意到$\gcd\{a_1, a_2, \cdots, a_n\} = \gcd\{a_1, a_2 - a_1, \cdots, a_n - a_{n-1}\}$,则线段树维护差分数组$diff[]$,则
$\gcd\{a_l, \cdots, a_r\} = \gcd\{a_l, \gcd\{diff[l+1], \cdots, diff[r]\}\}$,即将区间修改转化为对差分数组的单点修改.

$\gcd\{a_1, a_2, \cdots, a_n\} = \gcd\{a_1, a_2 - a_1, \cdots, a_n - a_{n-1}\}$的证明:

①先证$\text{LHS} \le \text{RHS}$.令$d = \gcd\{a_1, a_2, \cdots, a_n\}$.因gcd表示所有公约数中的最大值,只需证明$d$是RHS的每一项的约数.因$d = \gcd\{a_1, a_2, \cdots, a_n\}$,则$d \mid a_1, d \mid a_2$,进而$d \mid (a_2 - a_1)$,同理即证.

②再证$\text{LHS} \ge \text{RHS}$.令$d = \gcd\{a_1, a_2 - a_1, \cdots, a_n - a_{n-1}\}$,只需证$d$是LHS的每一项的约数.因$d \mid a_1, d \mid (a_2 - a_1)$,则$d \mid [(a_2 - a_1) + a_1]$,同理即证.

代码

```
const int MAXN = 5e5 + 5;
int n, m;  // 点数、操作数
ll a[MAXN];  // 原数组
struct Node {
  int l, r;  // 区间左右端点
  ll sum;  // 前缀和
  ll d;  // 区间gcd
}SegT[MAXN * 4];

ll gcd(ll a, ll b) {
  return b ? gcd(b, a % b) : a;
}

void push_up(Node& u, Node& l, Node& r) {  // u为左右儿子节点的父亲节点
  u.sum = l.sum + r.sum;
  u.d = gcd(l.d, r.d);
}

void push_up(int u) {  // 更新节点u的左右儿子节点
  push_up(SegT[u], SegT[u << 1], SegT[u << 1 | 1]);
}

void build(int u, int l, int r) {  // 以u为根节点,将区间[l,r]建SegT
  if (l == r) {  // 叶子节点
    ll tmp = a[r] - a[r - 1];
    SegT[u] = { l,r,tmp,tmp };
  }
  else {
    SegT[u].l = l, SegT[u].r = r;
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    push_up(u);
  }
}
```

```
35
36  void modify(int u, int x, ll v) {  // 从节点u开始,将第x个数修改为v
37    if (SegT[u].l == x && SegT[u].r == x) {  // 叶子节点
38      ll tmp = SegT[u].sum + v;
39      SegT[u] = { x,x,tmp,tmp };
40    }
41    else {
42      int mid = SegT[u].l + SegT[u].r >> 1;
43      if (x <= mid) modify(u << 1, x, v);
44      else modify(u << 1 | 1, x, v);
45      push_up(u);
46    }
47  }
48
49  Node query(int u, int l, int r) {  // 从节点u开始查询[l,r]的信息
50    if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u];
51    else {
52      int mid = SegT[u].l + SegT[u].r >> 1;
53      if (r <= mid) return query(u << 1, l, r);
54      else if (l > mid) return query(u << 1 | 1, l, r);
55      else {
56        auto left = query(u << 1, l, r), right = query(u << 1 | 1, l, r);
57        Node res;
58        push_up(res, left, right);
59        return res;
60      }
61    }
62  }
63
64  int main() {
65    for (int i = 1; i <= n; i++) cin >> a[i];
66
67    build(1, 1, n);  // 建SegT
68
69    while (m--) {
70      int l, r; char op; cin >> op >> l >> r;
71      if (op == 'Q') {
72        auto left = query(1, 1, l);
73        Node right({ 0,0,0,0 });
74        if (l + 1 <= r) right = query(1, l + 1, r);  // 防止l+1越界
75        cout << abs(gcd(left.sum, right.d)) << endl;  // 输出正的gcd
76      }
77      else {
78        ll d; cin >> d;
79        modify(1, l, d);
80        if (r + 1 <= n) modify(1, r + 1, -d);  // 防止r+1越界
81      }
82    }
83  }
```

17.2.4 线段树模板II

题意

给定一个长度为n $(1 \le n \le 1e5)$的数列$a = [a_1, \cdots, a_n]$ $(1 \le a_i \le 1e9)$. 现有m $(1 \le m \le 1e5)$个操作,操作命令有如下两种:

①$C\ l\ r\ d$,表示将$a[l \cdots r]$的数都加d $(|d| \le 1e4)$.

②$Q\ l\ r$,输出$\sum_{i=l}^{r} a_i$.

思路

线段树的节点记录区间左右端点l, r、只考虑当前节点及其子节点的懒标记(即不考虑当前节点的祖宗节点的标记)的当前区间和sum和懒标记$lazy$.

注意结果可能爆int.

代码

```
1  struct LazySegmentTree {
2      int n;
3      vector<int> a;
4      struct Lazy {
5          ll add;  // 区间加懒标记
6
7          Lazy(int _add = 0) :add(_add) {}
8
9          friend Lazy operator+(const Lazy A, const Lazy B) {
```

```
        Lazy res;
        res.add = A.add + B.add;
        return res;
    }
};
struct Node {
    ll sum;
    Lazy lazy;

    Node(int _sum = 0) :sum(_sum) {}

    friend Node operator+(const Node A, const Node B) {
        Node res;
        res.sum = A.sum + B.sum;
        return res;
    }
};
vector<Node> SegT;

LazySegmentTree(int _n, const vector<int>& _a) :n(_n), a(_a) {
    SegT.resize(n + 1 << 2);
    build(1, 1, n);
}

void pushUp(int u) {
    SegT[u] = SegT[u << 1] + SegT[u << 1 | 1];
}

void build(int u, int l, int r) {
    if (l == r) {
        SegT[u] = Node(a[l]);
        return;
    }

    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    pushUp(u);
}

void apply(int u, int l, int r, const Lazy lazy) {
    SegT[u].lazy = SegT[u].lazy + lazy;
    SegT[u].sum += (ll)(r - l + 1) * lazy.add;
}

void pushDown(int u, int l, int r) {
    auto& lazy = SegT[u].lazy;
    if (lazy.add) {
        int mid = l + r >> 1;
        apply(u << 1, l, mid, lazy), apply(u << 1 | 1, mid + 1, r, lazy);
        lazy = Lazy();
    }
}

void modify(int u, int l, int r, int L, int R, int val) {  // a[L...R] = val
    if (L <= l && r <= R) {
        SegT[u].sum += (ll)(r - l + 1) * val;
        SegT[u].lazy.add += val;
        return;
    }

    pushDown(u, l, r);
    int mid = l + r >> 1;
    if (L <= mid) modify(u << 1, l, mid, L, R, val);
    if (R > mid) modify(u << 1 | 1, mid + 1, r, L, R, val);
    pushUp(u);
}

void modify(int L, int R, int val) {
    modify(1, 1, n, L, R, val);
}

Node query(int u, int l, int r, int L, int R) {  // 求max a[L...R]
    if (R < l || L > r) return Node();
    if (L <= l && r <= R) return SegT[u];

    pushDown(u, l, r);
    int mid = l + r >> 1;
    return query(u << 1, l, mid, L, R) + query(u << 1 | 1, mid + 1, r, L, R);
}

ll query(int L, int R) {
    return query(1, 1, n, L, R).sum;
```

```
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    LazySegmentTree st(n, a);
    while (m--) {
        char op; int l, r; cin >> op >> l >> r;

        if (op == 'C') {
            int d; cin >> d;
            st.modify(l, r, d);
        }
        else cout << st.query(l, r) << endl;
    }
}

int main() {
    solve();
}
```

17.2.5 扫描线

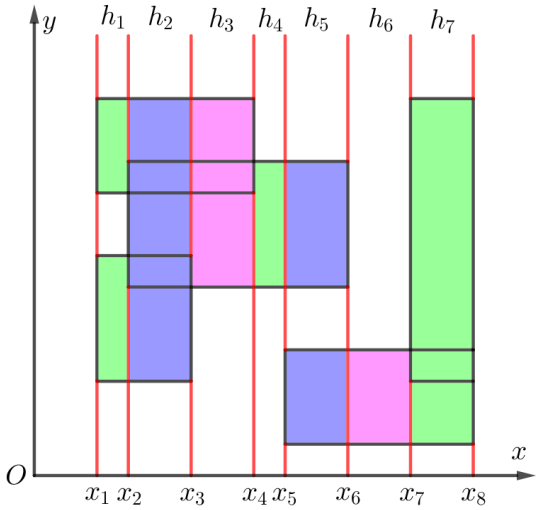
题意

有多组测试样例,每组测试样例第1行包含一个整数n $(1 \leq 1e4)$表示矩形的数量,接下来n行每行包含四个实数x_1, y_1, x_2, y_2 $(0 \leq x_1 < x_2 \leq 1e5, 0 \leq y_1 < y_2 \leq 1e5)$,表示长方形的左上角、右下角的坐标分别为$(x_1, y_1)$、$(x_2, y_2)$,坐标轴$x$轴从上到下沿伸,$y$轴从左向右沿伸.输入$n = 0$时表示终止,该用例无需处理.

每组测试样例输出两行,第一行输出"Test case #k",其中k为测试样例的编号,从1开始;第二行输出"Total explored area: a",其中a是矩形的面积并,若一个区域被多个矩形覆盖,则只计一次),精确到小数点后两位.每组测试样例后输出一个空行.

思路

如下图,将矩形的所有纵边延长成直线,将面积分割为每一竖条内的小矩形面积之和,即矩形纵边长度之和乘竖条宽度.(下图的坐标轴和本题的坐标轴不同,但这不影响答案)



现考虑用线段树维护y轴,将矩形的纵边变为带权的有向线段,每个矩形左边的竖边权值为$+1$,右边的竖边权值为-1.沿x轴从左往右扫,扫到对应的竖边即在y轴的对应区间上加上或减去1,更新后y轴的对应区间的权值即当前区间被几个矩形覆盖.从扫到第二条纵边起,每次计算y轴上至少被覆盖一次的区间总长.线段树的节点记录当前区间被覆盖的次数cnt和当前区间在不考虑祖先节点的cnt的前提下被至少覆盖了一次的区间总长len.

因涉及到区间修改,可加入懒标记来优化,但本题有利用扫描线的性质的优化:

(1)只查询整个区间,即考虑根节点的信息,故某区间包含于欲查询区间时直接return,无需push_down.

(2)所有操作成对出现,即某区间前面$+1$后面-1,且先加后减.

①减操作:能进行减操作的前提是之前有加操作,则减完后区间的值至少为0,且减操作涉及的区间依然是之前加操作涉及到的$O(4 \log n)$个区间,不会再分裂出新的区间,故减操作无需push_down.

②加操作:因查询时只用到根节点的信息,与其儿子节点的信息正确与否无关,有两种情况:

1)当前区间$cnt > 0$,则当前区间至少被cnt个矩形覆盖,此时$len = r - l + 1$.若再将该区间的一部分加上某个数,显然len不变,故无需push_down.

2)当前区间$cnt = 0$,则无需向下传递,即无需push_down.

综上,modify操作无需push_down.

push_up操作中若该节点的$cnt=0$,则其len等于其两儿子节点的len之和,否则$len=r-l+1$.

因本题的坐标可能有小数,故需对y轴进行离散化,最多有$2n$个y坐标,故线段树开$2n$个节点,故需开8倍空间.注意线段树维护的是区间而不是点,即离散化后的点为$y_1,y_2,\cdots$这$2n$个点,要维护的是$[y_1,y_2],[y_2,y_3],\cdots$这$(2n-1)$个区间.用区间的左端点代表该区间,则对区间$[l,r]$进行操作时用到的是$y_l$和$y_{r-1}$.

代码

```
const int MAXN = 1e5 + 5;
int n;  // 矩形数
struct Segment {  // 矩形的纵边
  double x, y1, y2;
  int k;  // 竖边的权值,左+1,右-1

  bool operator< (const Segment& t)const {  // 按x升序排序
    return x < t.x;
  }
}segs[MAXN * 2];
struct Node {
  int l, r;
  int cnt;  // 当前区间被覆盖的次数
  double len;  // 区间长度
}SegT[MAXN * 8];
vector<double> dis;  // 存y坐标离散化后的值

int find(double y) {  // 求y离散化后的在数组中的位置,下标从0开始
  return lower_bound(dis.begin(), dis.end(), y) - dis.begin();
}

void push_up(int u) {  // 用子节点更新父节点
  if (SegT[u].cnt) SegT[u].len = dis[SegT[u].r + 1] - dis[SegT[u].l];  // len=r-l+1
  else if (SegT[u].l != SegT[u].r)  // 不是叶子节点
    SegT[u].len = SegT[u << 1].len + SegT[u << 1 | 1].len;
  else SegT[u].len = 0;
}

void build(int u, int l, int r) {  // 从节点u开始将区间[l,r]建线段树
  SegT[u] = { l,r,0,0 };
  if (l != r) {  // 不是叶子节点
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  }
}

void modify(int u, int l, int r, int k) {  // 从节点u开始,将区间[l,r]内的数加k
  if (SegT[u].l >= l && SegT[u].r <= r) {  // 该区间包含于[l,r]中
    SegT[u].cnt += k;
    push_up(u);
  }
  else {
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r, k);
    if (r > mid) modify(u << 1 | 1, l, r, k);
    push_up(u);
  }
}

int main() {
  int T = 1;  // 测试样例编号
  while (cin >> n, n) {
    dis.clear();  // 每组测试样例清空离散化数组

    for (int i = 0, j = 0; i < n; i++) {
      double x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
      segs[j++] = { x1,y1,y2,1 };  // 左纵边
      segs[j++] = { x2,y1,y2,-1 };  // 右纵边
      dis.pb(y1), dis.pb(y2);
    }

    // 离散化
    sort(dis.begin(), dis.end());
    dis.erase(unique(dis.begin(), dis.end()), dis.end());

    build(1, 0, dis.size() - 2);  // n个节点对应(n-1)个区间,下标从0开始,故为(n-2)

    sort(segs, segs + n * 2);  // 纵边按x升序排序

    double ans = 0;
    for (int i = 0; i < n * 2; i++) {
```

```
72        if (i > 0)  // 从第2条纵边开始累加
73          ans += SegT[1].len * (segs[i].x - segs[i - 1].x);  // 小矩形的面积
74        modify(1, find(segs[i].y1), find(segs[i].y2) - 1, segs[i].k);  // 注意-1
75      }
76      cout << "Test case #" << T++ << endl;
77      cout << "Total explored area: " << fixed << setprecision(2) << ans << endl << endl;
78    }
79  }
```

17.2.6 维护序列

题意

给定一个长度为n $(1 \le n \le 1e5)$的序列$a[]$和模数p $(1 \le p \le 1e9)$,数列下标从1开始.现有m $(1 \le m \le 1e5)$个操作,操作命令有如下两种:

①$1\ t\ g\ c$,表示将所有a_i $(t \le i \le g)$修改为$a_i \times c$.

②$2\ t\ g\ c$,表示将所有a_i $(t \le i \le g)$修改为$a_i + c$.

③$3\ t\ g$,输出所有a_i $(t \le i \le g)$之和模p的值.

数据保证$1 \le t \le g \le n$.

思路

线段树的节点要记录区间的左右端点l, r、区间和sum、乘法懒标记mul、加法懒标记add.

考虑加法和乘法的先后问题:

①若先加再乘,对$(s + a) \times b$,乘法$(s + a) \times b \times c$易更新,但加法$(s + a) \times b + c$难更新为$(s+?)\times$?的形式.

②若先乘再加,则加法$s \times mul + add_1 + add_2 = s \times mul + add'$易更新,$(s \times mul_1 + add) \times mul_2 = s \times mul_1 \times mul_2 + add \times mul_2$易更新.

综上,维护时先乘后加.

优化:将乘法和加法同一为$s \times c + d$操作,则$(s \times a + b) \times c + d = s \times (ac) + (bc + d)$,即懒标记更新为$mul = ac, add = bc + d$.

初始时懒标记$add = 0, mul = 1$.

代码

```
1   const int MAXN = 1e5 + 5;
2   int n, p, m;  // 点数、模数、操作数
3   int a[MAXN];  // 原数组
4   struct Node {
5     int l, r;
6     int sum;  // 区间和
7     int add;  // 加法懒标记
8     int mul;  // 乘法懒标记
9   }SegT[MAXN * 4];
10
11  void push_up(int u) {
12    SegT[u].sum = (SegT[u << 1].sum + SegT[u << 1 | 1].sum) % p;
13  }
14
15  void cal(Node& u, int add, int mul) {  // 先乘mul再加add
16    u.sum = ((ll)u.sum * mul + (ll)(u.r - u.l + 1) * add) % p;
17    u.mul = (ll)u.mul * mul % p;
18    u.add = ((ll)u.add * mul + add) % p;
19  }
20
21  void push_down(int u) {
22    cal(SegT[u << 1], SegT[u].add, SegT[u].mul);
23    cal(SegT[u << 1 | 1], SegT[u].add, SegT[u].mul);
24    SegT[u].add = 0, SegT[u].mul = 1;  // 清空懒标记
25  }
26
27  void build(int u, int l, int r) {
28    if (l == r) SegT[u] = { l,r,a[r],0,1 };  // 叶子节点
29    else {
30      SegT[u] = { l,r,0,0,1 };  // add初始化为0,mul初始化为1
31      int mid = l + r >> 1;
32      build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
33      push_up(u);
34    }
35  }
36
37  void modify(int u, int l, int r, int add, int mul) {
38    if (SegT[u].l >= l && SegT[u].r <= r) cal(SegT[u], add, mul);
39    else {
```

```
    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r, add, mul);
    if (r > mid) modify(u << 1 | 1, l, r, add, mul);
    push_up(u);
  }
}

int query(int u, int l, int r) {
  if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u].sum;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  int res = 0;
  if (l <= mid) res = query(u << 1, l, r);
  if (r > mid) res = (res + query(u << 1 | 1, l, r)) % p;
  return res;
}

int main() {
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);

  cin >> m;
  while (m--) {
    int op, l, r; cin >> op >> l >> r;
    if (op == 1) {
      int d; cin >> d;
      modify(1, l, r, 0, d);
    }
    else if (op == 2) {
      int d; cin >> d;
      modify(1, l, r, d, 1);
    }
    else cout << query(1, l, r) << endl;
  }
}
```

17.2.7 Chess Strikes Back

原题指路:https://codeforces.com/contest/1379/problem/F2

题意

有一个$2n \times 2m$的棋盘.对格子(i, j),若$i + j$为偶数,则它染为白色,否则染为黑色.每个白色格子有两种状态:可用状态或禁用状态.初始时,所有白色格子都是可用状态.每轮A选择一个白色格子,反转其状态.B需在剩下的可用的白色格子上放$n \times m$个国王,使得任意以两国王为中心的3×3范围不相交,问B是否能做到.

第一行输入三个整数n, m, q $(1 \le n, m, q \le 2e5)$,分别表示棋盘大小和询问数.接下来q行每行输入两个整数i, j $(1 \le i \le 2n, 1 \le j \le 2m, i + j$为偶数$)$,表示反转白色格子$(i, j)$的状态.

对每个询问,若B能放国王,输出"YES",否则输出"NO".

思路

将2×2格子视为一个大格子,将棋盘划分为$n \times m$的棋盘,则每个大格子中只有左上和右下两个白色格子,且每个大格子放有且只有一个国王.下面的坐标(x, y)是对$n \times m$的棋盘称的.

对每个大格子,称左上角被禁用的大格子为L型,右下角被禁用的大格子为R型,一个大格子可同时为L和R.

①若存在L型格子(x_1, y_1)和R型格子(x_2, y_2),使得$x_1 \le y_2, y_1 \le y_2$,则它们相连的地方无法放国王,答案为NO.

②显然其余情况答案为YES.

显然可用map<pii,bool>来记录每个格子的状态.对每个x,维护两个变量:①a_x表示使得(x, y)是L型格子的最小的y;②b_x表示使得$(x - 1, y)$是R型格子的最大的y,它们可用set维护.

对每个操作,检查对所有$i > j$,是否都有$a_i > b_j$,这可用线段树实现:线段树节点记录三个变量:①区间a_i的最小值$mina$;②区间b_i的最大值$maxb$;③$flag$,表示区间是否对所有$i > j$,是否都有$a_i > b_j$.push_up时,只需检查是否有$mina_{left} > maxb_{right}$.因map默认初始化为false,则此处节点$flag$初始化为false,根节点的$flag$为true表示无解.

每个询问的时间复杂度为$O(\log n + \log q)$,总时间复杂度$O((n + q)(\log n + \log q))$.

代码

```
const int MAXN = 2e5 + 5;
int n, m, q;  // 棋盘大小、询问数
map<pii, bool> ban;  // 记录每个格子是否被禁用
set<int> L[MAXN], R[MAXN];  // 维护每行的L型、R型格子
struct Node {
```

```
  int l, r;
  int mina;  // 区间a_i最小值
  int maxb;  // 区间b_i最大值
  bool flag;  // 记录区间是否对所有i>j,都有a_i>b_j
}SegT[MAXN << 2];

void push_up(int u) {
  SegT[u].mina = min(SegT[u << 1].mina, SegT[u << 1 | 1].mina);
  SegT[u].maxb = max(SegT[u << 1].maxb, SegT[u << 1 | 1].maxb);
  SegT[u].flag = SegT[u << 1].flag || SegT[u << 1 | 1].flag || (SegT[u << 1].mina <= SegT[u << 1 | 1].maxb);  // 注意取或
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r, SegT[u].flag = false;
  if (l == r) {
    SegT[u].mina = m + 1, SegT[u].maxb = 0;
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void modify(int u, int l, int r, int pos) {  // pos为当前修改的格子的行号
  if (l == r) {
    SegT[u].mina = L[l].size() ? *L[l].begin() : m + 1;
    SegT[u].maxb = R[r].size() ? *R[r].rbegin() : 0;
    SegT[u].flag = SegT[u].mina <= SegT[u].maxb;
    return;
  }

  int mid = l + r >> 1;
  if (pos <= mid) modify(u << 1, l, mid, pos);
  else modify(u << 1 | 1, mid + 1, r, pos);
  push_up(u);
}

void solve() {
  cin >> n >> m >> q;

  build(1, 1, n);

  while (q--) {
    int x, y; cin >> x >> y;
    x++, y++;  // 方便下面除以2上取整,注意+1后奇偶性改变

    // 反转格子状态
    if (ban[{x, y}]) {
      if (y & 1) R[x >> 1].erase(y >> 1);
      else L[x >> 1].erase(y >> 1);
    }
    else {
      if (y & 1) R[x >> 1].insert(y >> 1);
      else L[x >> 1].insert(y >> 1);
    }
    ban[{x, y}] ^= 1;

    modify(1, 1, n, x >> 1);
    cout << (SegT[1].flag ? "NO" : "YES") << endl;
  }
}

int main() {
  solve();
}
```

17.2.8 Chtholly and the Broken Chronograph

题意

维护一个长度为n的序列$a_1,\cdots,a_n$,其中每个元素a_i有一个状态s_i:①$s_i=0$表示第i个元素禁用;②$s_i=1$表示第i个元素启用.

现有如下四种操作:

①1 x,表示将s_x置为0,数据保证操作前$s_x=1,1\le x\le n$

②2 x,表示将s_x置为1,数据保证操作前$s_x=0,1\le x\le n$.

③3 l r x,表示将区间$[l,r]$中所有$s_i=1$的$a_i+=x$,数据保证$1\le l\le r\le n,1\le x\le 1e8$.

④4 l r,表示查询区间$[l,r]$中所有a_i之和,无论a_i的状态.数据保证$1 \le l \le r \le n$.

第一行输入两个整数n,q $(1 \le n,q \le 1e5)$.第二行输入n个整数$a_1,\cdots,a_n$ $(1 \le a_i \le 1e8)$.第三行输入n个整数$s_1,\cdots,s_n$ $(s_i \in \{0,1\})$.接下来q行每行输入一个操作,格式如上.

思路

线段树节点维护一个变量cnt表示区间内$s_i = 1$的a_i的个数.

代码

```
const int MAXN = 1e5 + 5;
namespace SegmentTree {
  int n;
  int a[MAXN];
  int s[MAXN];  // 每个元素的状态
  struct Node {
    int l, r;
    int s;  // 元素的状态,s=0表示禁用,s=1表示启用
    int cnt;  // 区间s=1的元素个数
    ll sum;  // 区间和
    ll lazy;  // 加法懒标记
  }SegT[MAXN << 2];

  void push_up(int u) {
    SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
    SegT[u].cnt = SegT[u << 1].cnt + SegT[u << 1 | 1].cnt;
  }

  void build(int u, int l, int r) {
    SegT[u].l = l, SegT[u].r = r;
    if (l == r) {
      SegT[u].sum = a[l];
      SegT[u].s = s[l];
      SegT[u].cnt = SegT[u].s;
      return;
    }

    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    push_up(u);
  }

  void push_down(int u) {
    SegT[u << 1].sum += SegT[u].lazy * SegT[u << 1].cnt;
    SegT[u << 1].lazy += SegT[u].lazy;
    SegT[u << 1 | 1].sum += SegT[u].lazy * SegT[u << 1 | 1].cnt;
    SegT[u << 1 | 1].lazy += SegT[u].lazy;
    SegT[u].lazy = 0;
  }

  void modify_state(int u, int x) {  // 反转a[x].s
    if (SegT[u].l == SegT[u].r) {  // 暴力修改叶子节点
      SegT[u].s ^= 1;
      SegT[u].cnt = SegT[u].s;
      return;
    }

    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (x <= mid) modify_state(u << 1, x);
    else modify_state(u << 1 | 1, x);
    push_up(u);
  }

  void modify_add(int u, int l, int r, int x) {
    if (l <= SegT[u].l && SegT[u].r <= r) {
      SegT[u].sum += (ll)SegT[u].cnt * x;
      SegT[u].lazy += x;
      return;
    }

    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify_add(u << 1, l, r, x);
    if (r > mid) modify_add(u << 1 | 1, l, r, x);
    push_up(u);
  }

  ll query(int u, int l, int r) {
```

```
    if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].sum;

    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    ll res = 0;
    if (l <= mid) res += query(u << 1, l, r);
    if (r > mid) res += query(u << 1 | 1, l, r);
    return res;
  }
};
using namespace SegmentTree;

void solve() {
  int q; cin >> n >> q;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i <= n; i++) cin >> s[i];

  build(1, 1, n);

  while (q--) {
    int op; cin >> op;
    if (op == 1 || op == 2) {
      int x; cin >> x;
      modify_state(1, x);
    }
    else if (op == 3) {
      int l, r, x; cin >> l >> r >> x;
      modify_add(1, l, r, x);
    }
    else {
      int l, r; cin >> l >> r;
      cout << query(1, l, r) << endl;
    }
  }
}

int main() {
  solve();
}
```

17.2.9 XOR on Segment

原题指路:https://codeforces.com/problemset/problem/242/E

题意 (2 s)

维护一个长度为n的序列$a_1,\cdots,a_n$,支持如下两种操作:①1 l r,表示询问$[l,r]$的区间和;②2 l r x,表示令所有$a_i^{\wedge}=x\ \ (l\le i\le r)$.

第一行输入一个整数$n\ \ (1\le n\le 1e5)$.第二行输入n个整数$a_1,\cdots,a_n\ \ (0\le a_i\le 1e6)$.第三行输入一个整数$m\ \ (1\le m\le 5e4)$,表示操作次数.接下来$m$行每行输入一个操作,其中操作②中$1\le x\le 1e6$.数据保证操作合法.

思路

显然异或操作对二进制的每个数位的影响独立,而用二进制的数位可计算区间和,线段树节点维护区间异或的懒标记$lazy$和每个数位上的1的个数$cnt[31]$.

对操作②,枚举x的二进制表示每个数位,若当前数位是1,则反转区间对应数位,即将对应数位上1的个数cnt更新为$len-cnt$,其中len是区间长度.

代码

```
const int MAXN = 1e5 + 5;
namespace SegmentTree {
  int n;
  int a[MAXN];
  struct Node {
    int l, r;
    int lazy;  // 区间异或的懒标记
    int cnt[31];  // 每个数位上1的个数
  }SegT[MAXN << 2];

  int get_length(const Node& u) { return u.r - u.l + 1; }

  void push_up(int u) {
    for (int i = 0; i < 31; i++)
      SegT[u].cnt[i] = SegT[u << 1].cnt[i] + SegT[u << 1 | 1].cnt[i];
  }

  void build(int u, int l, int r) {
    SegT[u] = { l,r };
    if (l == r) {
      for (int i = 0; i < 31; i++)
```

```
        if (a[l] >> i & 1) SegT[u].cnt[i]++;
      return;
    }

    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    push_up(u);
  }

  void push_down(int u) {
    if (SegT[u].lazy) {
      SegT[u << 1].lazy ^= SegT[u].lazy, SegT[u << 1 | 1].lazy ^= SegT[u].lazy;
      for (int i = 0; i < 31; i++) {
        if (SegT[u].lazy >> i & 1) {
          // 区间0-1翻转
          SegT[u << 1].cnt[i] = get_length(SegT[u << 1]) - SegT[u << 1].cnt[i];
          SegT[u << 1 | 1].cnt[i] = get_length(SegT[u << 1 | 1]) - SegT[u << 1 | 1].cnt[i];
        }
      }
      SegT[u].lazy = 0;
    }
  }

  void modify(int u, int l, int r, int x) {  // [l,r]^=x
    if (l <= SegT[u].l && SegT[u].r <= r) {
      for (int i = 0; i < 31; i++)
        if (x >> i & 1) SegT[u].cnt[i] = get_length(SegT[u]) - SegT[u].cnt[i];
      SegT[u].lazy ^= x;
      return;
    }

    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r, x);
    if (r > mid) modify(u << 1 | 1, l, r, x);
    push_up(u);
  }

  ll query(int u, int l, int r) {  // 查询区间和
    if (l <= SegT[u].l && SegT[u].r <= r) {
      ll res = 0;
      for (int i = 0; i < 31; i++)
        res += (ll)SegT[u].cnt[i] * (1 << i);
      return res;
    }

    push_down(u);
    ll res = 0;
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) res = query(u << 1, l, r);
    if (r > mid) res += query(u << 1 | 1, l, r);
    return res;
  }
}
using namespace SegmentTree;

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);

  CaseT{
    int op, l, r; cin >> op >> l >> r;
    if (op == 1) cout << query(1, l, r) << endl;
    else {
      int x; cin >> x;
      modify(1, l, r, x);
    }
  }
}

int main() {
  solve();
}
```

17.2.10 Yash And Trees

原题指路:https://codeforces.com/problemset/problem/633/G

题意 (4 s)

给定一棵包含编号$1\sim n$的n个节点的树,其中1号节点为根节点,第i $(1\le i\le n)$个节点的权值为a_i.

给定一个整数m.现有操作:

①$1\ v\ x$ $(1\le v\le n,0\le x\le 1e9)$,表示令节点$v$的子树上的所有节点的权值$+=x$.

②$2\ v$ $(1\le v\le n)$,表示询问节点v的子树上的所有节点中有多少个节点的权值模m的余数为素数.

第一行输入两个整数n,m $(1\le n\le 1e5,1\le m\le 1000)$.第二行输入$n$个整数$a_1,\cdots,a_n$ $(0\le a_i\le 1e9)$.接下来$(n-1)$行每行输入两个整数u,v $(1\le u,v\le n,u\ne v)$,表示节点u与v间存在无向边.接下来一行输入一个整数q $(1\le q\le 1e5)$,表示操作次数.接下来q行每行输入一个操作,格式如上.

思路

容易想到对树DFS序建线段树,将操作①转化为区间加法.

注意到m很小,线段树节点维护一个bitset,每一位表示区间中是否存在模m的相应余数,若存在则对应位为1;否则对应位为0,此时区间加法转化为bitset的循环移位,这可通过bitset左移的结果或上右移的结果实现.

预处理出表示m以内的数是否为素数的bitset,则操作②的结果即区间的bitset与素数的bitset相与的结果中1的个数.

代码

```cpp
const int MAXN = 1e5 + 5;
int n, m;
int a[MAXN];
vi edges[MAXN];
int dfn[MAXN], idfn[MAXN], tim;  // 节点的DFS序
int siz[MAXN];  // siz[u]表示以u为根节点的子树的大小
bitset<1005> full;  // [0,m-1]位为1,表示余数的取值范围
bitset<1005> primes;  // 记录m以内的数是否为素数
struct Node {
  int l, r;
  bitset<1005> rest;  // 记录区间模m的余数是否存在
  int lazy;  // 加法的懒标记
}SegT[MAXN << 2];

void init() {  // 预处理full,primes
  for (int i = 0; i < m; i++) full.set(i);

  primes.set();
  primes.reset(0), primes.reset(1);
  for (int i = 2; i <= m; i++) {
    if (primes[i])
      for (int j = 2 * i; j <= m; j += i) primes.reset(j);
  }
}

void dfs(int u, int fa) {  // 预处理出DFS序:当前节点、前驱节点
  dfn[u] = ++tim, idfn[tim] = u;
  siz[u] = 1;
  for (auto v : edges[u]) {
    if (v != fa) {
      dfs(v, u);
      siz[u] += siz[v];
    }
  }
}

void push_up(int u) {
  SegT[u].rest = SegT[u << 1].rest | SegT[u << 1 | 1].rest;
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r;
  if (l == r) {
    SegT[u].rest.set(a[idfn[l]] % m);
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void shift(int u, int v) {  // 区间bitset循环左移v位
  SegT[u].lazy = (SegT[u].lazy + v) % m;
  SegT[u].rest = ((SegT[u].rest << v) & full) | (SegT[u].rest >> (m - v));  // 注意&full
```

```cpp
}

void push_down(int u) {
  if (SegT[u].lazy) {
    shift(u << 1, SegT[u].lazy), shift(u << 1 | 1, SegT[u].lazy);
    SegT[u].lazy = 0;
  }
}

void modify(int u, int l, int r, int v) {  // [l,r]+=v
  if (l <= SegT[u].l && SegT[u].r <= r) {
    shift(u, v);
    return;
  }

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify(u << 1, l, r, v);
  if (r > mid) modify(u << 1 | 1, l, r, v);
  push_up(u);
}

bitset<1005> query(int u, int l, int r) {  // 查询区间模m的余数
  if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].rest;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  bitset<1005> res;
  if (l <= mid) res |= query(u << 1, l, r);
  if (r > mid) res |= query(u << 1 | 1, l, r);
  return res;
}

void solve() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    edges[u].push_back(v), edges[v].push_back(u);
  }

  init();
  dfs(1, 0);  // 从根节点开始搜,根节点无前驱节点
  build(1, 1, n);

  CaseT{
    int op, v; cin >> op >> v;
    if (op == 1) {
      int x; cin >> x;
      modify(1, dfn[v], dfn[v] + siz[v] - 1, x % m);  // 注意取模
    }
    else cout << (query(1, dfn[v], dfn[v] + siz[v] - 1) & primes).count() << endl;
  }
}

int main() {
  solve();
}
```

17.2.11 Danil and a Part-time Job

原题指路:https://codeforces.com/problemset/problem/877/E

题意 (2 s)

维护一棵以1号节点为根节点的树,树上每个节点有一个数字0或1,支持操作:①$pow\ u$,表示将以u为根节点的子树中的所有节点的数字反转;②$get\ u$,表示询问以u为根节点的子树中1的个数.

第一行输入一个整数n $(1 \le n \le 2e5)$,表示树的节点数.第二行输入$(n-1)$个整数,其中第i $(1 \le i \le n-1)$个整数表示节点$(i+1)$的父亲节点.第三行输入n个整数$a_1, \cdots, a_n$ $(a_i \in \{0, 1\})$,表示初始时树上每个节点的数字.第四行输入一个整数q $(1 \le q \le 2e5)$,表示操作个数.接下来q行每行输入一个操作,格式如上,数据保证操作合法.

思路

用DFS序线段树维护即可.

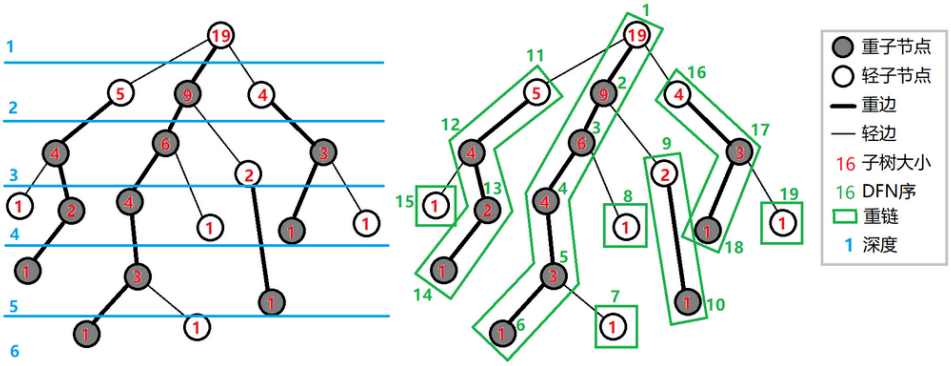
代码

```
const int MAXN = 2e5 + 5;
namespace DFS_SegmentTree {
  int n;
  bool a[MAXN];  // 每个节点的权值
  vi edges[MAXN];
  int dfn[MAXN], idfn[MAXN], tim;  // 节点的DFS序
  int siz[MAXN];  // siz[u]表示以u为根节点的子树的大小
  struct Node {
    int l, r;
    int cnt;  // 区间内1的个数
    bool lazy;  // 区间反转懒标记
  }SegT[MAXN << 2];

  int getLength(int u) { return SegT[u].r - SegT[u].l + 1; }

  void dfs(int u) {  // 预处理出DFS序:当前节点
    dfn[u] = ++tim, idfn[tim] = u;
    siz[u] = 1;
    for (auto v : edges[u]) {
      dfs(v);
      siz[u] += siz[v];
    }
  }

  void push_up(int u) {
    SegT[u].cnt = SegT[u << 1].cnt + SegT[u << 1 | 1].cnt;
  }

  void build(int u, int l, int r) {
    SegT[u].l = l, SegT[u].r = r;
    if (l == r) {
      SegT[u].cnt = a[idfn[l]];
      return;
    }

    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    push_up(u);
  }

  void push_down(int u) {
    if (SegT[u].lazy) {
      SegT[u << 1].cnt = getLength(u << 1) - SegT[u << 1].cnt;
      SegT[u << 1 | 1].cnt = getLength(u << 1 | 1) - SegT[u << 1 | 1].cnt;
      SegT[u << 1].lazy ^= 1, SegT[u << 1 | 1].lazy ^= 1;
      SegT[u].lazy = 0;
    }
  }

  void modify(int u, int l, int r) {  // 反转区间[l,r]
    if (l <= SegT[u].l && SegT[u].r <= r) {
      SegT[u].cnt = getLength(u) - SegT[u].cnt;
      SegT[u].lazy ^= 1;
      return;
    }

    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r);
    if (r > mid) modify(u << 1 | 1, l, r);
    push_up(u);
  }

  int query(int u, int l, int r) {  // 询问区间[l,r]中1的个数
    if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].cnt;

    push_down(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    int res = 0;
    if (l <= mid) res += query(u << 1, l, r);
    if (r > mid) res += query(u << 1 | 1, l, r);
    return res;
  }
}
using namespace DFS_SegmentTree;

void solve() {
  cin >> n;
  for (int i = 2; i <= n; i++) {
```

```
    int p; cin >> p;
    edges[p].push_back(i);
  }
  for (int i = 1; i <= n; i++) cin >> a[i];

  dfs(1);
  build(1, 1, n);

  CaseT{
    string op; int u; cin >> op >> u;
    if (op == "pow") modify(1, dfn[u], dfn[u] + siz[u] - 1);
    else cout << query(1, dfn[u], dfn[u] + siz[u] - 1) << endl;
  }
}

int main() {
  solve();
}
```

17.3 树链剖分

树链剖分将树上的节点重新编号, 将树转化为一个序列, 将树上的任一条路径都能拆分为$O(\log n)$个连续区间, 进而用线段树维护.



树中节点的所有儿子节点的子树中, 含节点数最多的子树的根节点称为该树中结点的重儿子, 其他子树的根节点称为该树中节点的轻儿子. 若一个树中结点有多个含节点数最多的子树, 则任选一个为重儿子. 重儿子与其父亲节点相连的边称为重边, 其他边称为轻边. 极大的由重边构成的路径称为重链. 树上的每个节点都在一条重链中, 其中轻儿子在其开始往下的重链中, 重儿子在其父亲节点开始的重链中. 重儿子可在DFS时记录每棵子树的大小, 遍历完一个节点的所有儿子节点后, 取子树所含节点最多的儿子节点为重儿子.

按树的DFS序将其转化为一个序列, 其中DFS优先遍历每个节点的重儿子, 这样每条重链所含的节点编号连续. 另外, DFS也保证了一棵子树所含的节点编号连续.

树上的任一条路径都能拆分为$O(\log n)$条重链, 而每条重链所含的节点编号连续, 故树上的任一条路径都能拆分为$O(\log n)$个连续区间. 第二次DFS时一边求出DFS序, 一边找到重链(记录每个节点所在重链的顶点).

将一条路径转化拆分为$O(\log n)$条重链的方法: 对树上的两节点u和v, 每次走到它们所在的重链的顶点u'和v', 取出较矮的顶点(不妨设为u')到u的区间, 再从u'和v'走到它们所在重链的顶点. 重复上述操作, 直至两路径会在u和v的LCA所在的重链上会和.

以树剖后用线段树维护为例. 每个询问的路径会被拆分为$O(\log n)$个连续区间, 线段树对区间操作的时间复杂度为$O(n \log n)$, 故树剖中每个询问的时间复杂度为$O(n \log^2 n)$, 常数低.

17.3.1 树链剖分

题意

给定一棵树,包含编号为$1 \sim n$的n个节点,其中节点i的权值为a_i.初始时1号节点为树的根节点.

现有m个操作,有如下四种类型:

①$1\ u\ v\ k$,修改路径上节点的权值,表示将节点u和v间的路径上的所有节点(含这两个节点)的权值$+k$.

②$2\ u\ k$,修改子树所含节点的权值,表示将以u为根节点的子树上的所有节点的权值$+k$.

③$3\ u\ v$,询问路径,表示询问节点u和v间的路径上所有节点(含这两个节点)的权值$+k$.

④$4\ u$,询问子树,表示询问以节点u为根节点的子树所含节点的权值之和.

第一行输入整数n $(1 \le n \le 1e5)$,表示树上的节点数.第二行输入n个整数$a_1, \cdots, a_n$ $(0 \le a_i \le 1e5)$,表示初始时节点的权值.接下来$(n-1)$行每行输入两个整数x, y $(1 \le x, y \le n)$,表示节点x和y间存在一条边.接下来一行输入一个整数m $(1 \le m \le 1e5)$,表示操作次数.接下来m行每行输入一个操作,数据保证操作合法.

代码

```
const int MAXN = 1e5 + 5, MAXM = MAXN << 1;
int n;  // 节点数
int head[MAXN], edge[MAXM], w[MAXN], nxt[MAXM], idx;
int dfn[MAXN], cnt;  // 节点的DFS序
int neww[MAXN];  // neww[i]表示DFS序为i的节点的权值
int depth[MAXN];  // 节点的深度
int siz[MAXN];  // siz[u]表示以节点u为根节点的子树的大小
int top[MAXN];  // top[u]表示节点u所在的重链的顶点
int father[MAXN];  // 每个节点的父亲节点
int son[MAXN];  // 每个树中节点的重儿子
struct Node {
  int l, r;
  ll lazy;
  ll sum;  // 区间和
}SegT[MAXN << 2];

void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

void dfs1(int u, int fa, int dep) {  // 求树中节点的重儿子:当前节点为u,其父亲节点为fa,u的深度为dep
  depth[u] = dep, father[u] = fa, siz[u] = 1;
  for (int i = head[u]; ~i; i = nxt[i]) {
    int j = edge[i];
    if (j == fa) continue;

    dfs1(j, u, dep + 1);
    siz[u] += siz[j];
    if (siz[son[u]] < siz[j]) son[u] = j;
  }
}

void dfs2(int u, int to) {  // 找到每条重链:当前节点u所在的重链顶点为to
  dfn[u] = ++cnt, neww[cnt] = w[u], top[u] = to;
  if (!son[u]) return;  // 叶子节点

  dfs2(son[u], to);  // 优先搜重儿子,重儿子与u在同一条重链上

  for (int i = head[u]; ~i; i = nxt[i]) {  // 搜轻儿子
    int j = edge[i];
    if (j == father[u] || j == son[u]) continue;

    dfs2(j, j);  // 轻儿子是自己所在的重链的顶点
  }
}

void push_up(int u) {
  SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
}

void build(int u, int l, int r) {
  SegT[u] = { l,r,0,neww[r] };
  if (l == r) return;

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void push_down(int u) {
  auto &root = SegT[u], &left = SegT[u << 1], &right = SegT[u << 1 | 1];
  if (root.lazy) {
    left.lazy += root.lazy, left.sum += root.lazy * (left.r - left.l + 1);
    right.lazy += root.lazy, right.sum += root.lazy * (right.r - right.l + 1);
    root.lazy = 0;
  }
}

void modify(int u, int l, int r, int k) {  // [l,r]+=k
  if (l <= SegT[u].l && r >= SegT[u].r) {
    SegT[u].lazy += k;
    SegT[u].sum += (ll)k * (SegT[u].r - SegT[u].l + 1);
    return;
  }

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify(u << 1, l, r, k);
  if (r > mid) modify(u << 1 | 1, l, r, k);
```

```
  push_up(u);
}

ll query(int u, int l, int r) {
  if (l <= SegT[u].l && r >= SegT[u].r) return SegT[u].sum;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  ll res = 0;
  if (l <= mid) res += query(u << 1, l, r);
  if (r > mid) res += query(u << 1 | 1, l, r);
  return res;
}

void modify_path(int u, int v, int k) {  // 节点u到v的路径上的节点权值+=k
  while (top[u] != top[v]) {  // 节点u和v不在一条重链上
    if (depth[top[u]] < depth[top[v]]) swap(u, v);  // 保证节点u比v更深
    modify(1, dfn[top[u]], dfn[u], k);
    u = father[top[u]];
  }

  // 节点u和v在同一条重链上
  if (depth[u] < depth[v]) swap(u, v);  // 保证节点u比节点v更深
  modify(1, dfn[v], dfn[u], k);
}

ll query_path(int u, int v) {  // 询问节点u到v的路径上节点的权值和
  ll res = 0;
  while (top[u] != top[v]) {  // 节点u和v不在一条重链上
    if (depth[top[u]] < depth[top[v]]) swap(u, v);  // 保证节点u比v更深
    res += query(1, dfn[top[u]], dfn[u]);
    u = father[top[u]];
  }

  // 节点u和v在同一条重链上
  if (depth[u] < depth[v]) swap(u, v);  // 保证节点u比节点v更深
  res += query(1, dfn[v], dfn[u]);
  return res;
}

void modify_subtree(int u, int k) {  // 以节点u为根节点的子树所含节点的权值+=k
  modify(1, dfn[u], dfn[u] + siz[u] - 1, k);
}

ll query_subtree(int u) {  // 询问以节点u为根节点的子树所含节点的权值和
  return query(1, dfn[u], dfn[u] + siz[u] - 1);
}

int main() {
  memset(head, -1, so(head));

  cin >> n;
  for (int i = 1; i <= n; i++) cin >> w[i];
  for (int i = 0; i < n - 1; i++) {
    int a, b; cin >> a >> b;
    add(a, b), add(b, a);
  }

  dfs1(1, -1, 1);  // 从根节点开始搜,根节点无父亲节点,根节点的深度为1
  dfs2(1, 1);  // 从根节点开始搜,根节点所在的重链的顶点是根节点
  build(1, 1, n);

  CaseT{
    int op, u, v, k; cin >> op;
    switch (op) {
      case 1:
        cin >> u >> v >> k;
        modify_path(u, v, k);
        break;
      case 2:
        cin >> u >> k;
        modify_subtree(u, k);
        break;
      case 3:
        cin >> u >> v;
        cout << query_path(u, v) << endl;
        break;
      case 4:
        cin >> u;
        cout << query_subtree(u) << endl;
        break;
    }
```

```
  }
}
```

17.3.2 软件包管理器

题意

若软件包A依赖软件包B,则安装A前要安装B,卸载B前要卸载A.

现已知所有软件包间的依赖关系.除0号软件包外,其他软件包都依赖且仅依赖一个软件包,0号软件包不依赖其他软件包.依赖关系不存在环(若有m $(m \geq 2)$个软件包$A_1, \cdots, A_m$,其中A_1依赖A_2, A_2依赖$A_3, \cdots, A_{m-1}$依赖A_m, A_m依赖A_1,则称这m个软件包的依赖关系成环),也不会有软件包依赖自己.

现希望在安装和卸载某软件包时快速知道该操作实际会改变多少个软件包的安装状态,即安装操作会安装多少个未安装的软件包,或卸载操作会卸载多少个已安装的软件包.注意,安装一个已安装的软件包或卸载一个未安装的软件包不影响任何软件包的安装状态,此时改变安装状态的软件包数为0.

第一行输入一个整数n $(1 \leq n \leq 1e5)$,表示软件包的总数,软件包编号从0开始.第二行包含$(n-1)$个整数,分别表示$1, 2, \cdots, n-1$号软件包依赖的软件包的编号.第三行输入一个整数q $(1 \leq q \leq 1e5)$,表示询问个数.接下来q行每行输入一个询问,询问有两种:①$install\ x$,表示安装软件包x;②$uninstall\ x$,表示卸载软件包x.设一开始所有软件包都未安装,你需维护所有软件包的安装状态.对每个询问,先输出进行此操作会改变多少个软件包的安装状态,随后应用该操作(即改变维护的安装状态).

思路

除0号软件包外,其他软件包都依赖且仅依赖一个软件包,这表明软件包的依赖关系是一棵树.每个软件包有安装(1)和未安装(0)两种状态.安装软件包x等价于将x到根节点的路径上的软件包状态都变为1,改变的软件包安装状态数即将多少个0变为1.卸载软件包x等价于将以x为根节点的子树的软件包状态都变为0,改变的软件包安装状态数即将多少个1变为0.改变的软件包安装状态数可用线段树求路径或子树的权值和来实现.对安装操作,设x到根节点的路径上的权值和为a,安装x后x到根节点的路径上的权值和为b,则改变的软件包安装状态数即$(b-a)$.

线段树需实现将区间修改为0或1,懒标记$lazy$有三种取值:①-1,表示无操作;②0,表示将区间修改为0;③1,表示将区间修改为1.

因线段树下标从1开始,软件包编号从0开始,故将软件包编号映射为从1开始.

代码

```
const int MAXN = 1e5 + 5;
int n;  // 安装包数
vi edges[MAXN];
int dfn[MAXN], cnt;  // DFS序
int depth[MAXN];  // 节点的深度
int siz[MAXN];  // 子树的大小
int top[MAXN];  // 重链的顶点
int father[MAXN];  // 每个节点的父亲节点
int son[MAXN];  // 树中节点的重儿子
struct Node {
  int l, r;
  int lazy;  // -1表示无操作;0表示将区间修改为0;1表示将区间修改为1
  int sum;  // 区间和
}SegT[MAXN << 2];

void dfs1(int u, int dep) {  // 求树中节点的重儿子
  depth[u] = dep, siz[u] = 1;
  for (auto v : edges[u]) {
    dfs1(v, dep + 1);
    siz[u] += siz[v];
    if (siz[son[u]] < siz[v]) son[u] = v;
  }
}

void dfs2(int u, int to) {  // 找到每条重链
  dfn[u] = ++cnt, top[u] = to;
  if (!son[u]) return;

  dfs2(son[u], to);
  for (auto v : edges[u]) {
    if (v == son[u]) continue;

    dfs2(v, v);
  }
}

void push_up(int u) {
  SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
}

void build(int u, int l, int r) {
  SegT[u] = { l,r,-1,0 };  // lazy=-1表示无操作
  if (l == r) return;

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
```

```
  // 因初始时每个安装包的状态都为0,故无需push_up
}

void push_down(int u) {
  auto &root = SegT[u], &left = SegT[u << 1], &right = SegT[u << 1 | 1];
  if (~root.lazy) {
    left.sum = root.lazy * (left.r - left.l + 1);
    right.sum = root.lazy * (right.r - right.l + 1);
    left.lazy = right.lazy = root.lazy;
    root.lazy = -1;  // 注意清空为-1
  }
}

void modify(int u, int l, int r, int k) {  // [l,r]修改为k
  if (l <= SegT[u].l && r >= SegT[u].r) {
    SegT[u].lazy = k;
    SegT[u].sum = k * (SegT[u].r - SegT[u].l + 1);
    return;
  }

  int mid = SegT[u].l + SegT[u].r >> 1;
  push_down(u);
  if (l <= mid) modify(u << 1, l, r, k);
  if (r > mid) modify(u << 1 | 1, l, r, k);
  push_up(u);
}

void modify_path(int u, int v, int k) {  // 将u到v路径上的节点的状态修改为k
  while (top[u] != top[v]) {
    if (depth[top[u]] < depth[top[v]]) swap(u, v);
    modify(1, dfn[top[u]], dfn[u], k);
    u = father[top[u]];
  }

  if (depth[u] < depth[v]) swap(u, v);
  modify(1, dfn[v], dfn[u], k);
}

void modify_subtree(int u, int k) {  // 将以u为根节点的子树所含节点的状态修改为k
  modify(1, dfn[u], dfn[u] + siz[u] - 1, k);
}

int main() {
  cin >> n;
  for (int i = 2; i <= n; i++) {
    int x; cin >> x;
    x++;  // 下标从1开始
    edges[x].push_back(i);
    father[i] = x;
  }

  dfs1(1, 1);
  dfs2(1, 1);
  build(1, 1, n);

  CaseT{
    string op; int u; cin >> op >> u;
    u++;  // 下标从1开始
    int tmp = SegT[1].sum;
    if (op == "install") modify_path(1, u, 1);
    else modify_subtree(u, 0);
    cout << abs(tmp - SegT[1].sum) << endl;
  }
}
```

17.3.3 Eezie and Pie

原题指路:https://ac.nowcoder.com/acm/contest/33191/B

题意 (3 s)

给定一棵包含编号$1 \sim n$的n个节点的树,其中根节点为1号,每个节点处有外卖站和点外卖的人.节点i $(1 \leq i \leq n)$处的外卖站只能送节点i到根节点的简单路径上的节点.此外,节点i $(1 \leq i \leq n)$处的外卖站有一个最长距离d_i,表示该外卖站只能送与其距离(经过的边数)不超过d_i的节点.对每个节点i $(1 \leq i \leq n)$,求有多少个外卖站可以给它送外卖.

第一行输入整数n $(1 \leq n \leq 2e6)$.接下来$(n-1)$行每行输入两个整数u, v $(1 \leq u, v \leq n)$,表示节点u与v间存在无向边.数据保证输入的信息构成一棵树.最后一行输入n个整数$d_1, \cdots, d_n$ $(0 \leq d_i \leq n)$.

输出n个整数,其中第i $(1 \leq i \leq n)$个整数表示对节点i,求有多少个外卖站可以给它送外卖.

思路

对每个节点u,考察它到根节点的简单路径上与其相距d_u的节点v(若u到根节点的距离$\leq d_u$,则取v为根节点),则u处的外卖站能给u到v的简单路径上的节点(含端点)送外卖,只需给这段路径上的节点的权值都$+1$即可.考虑树剖,它可快速实现给两节点间的路径上的节点权值$+1$,而线段树可快速实现查询每个节点处的权值,故只需解决对每个节点u,如何快速找到对应的v.

朴素做法,$u = father[u]$回跳d_u次(中途跳到根节点可break),最坏的情况是一条链,需回跳$O(n^2)$次,显然TLE.

考虑优化,类似于Manacher算法的优化思路,能跳尽量跳,不能跳再暴力扩展.注意到树剖中$top[u]$表示节点u所在重链的顶点,则若它们的深度之差$\leq d[u]$,则u直接跳到其所在重链的顶点$top[u]$处,并更新剩余的步数.重复该过程,直至剩下的步数不足以跳到所在重链的顶点或已到达根节点.若此时还有剩下的步数,则暴力$u = father[u]$回跳,直至找到v.过程:设$1, a, b, c$是一条重链,c的儿子节点是d,而d, e, f, g是一条重链,且$d[g] = 6$.首先g花费3的步数跳到所在重链的顶点d处,此时剩下3个步数.d花费一个步数跳到其父亲节点c处,此时剩下2个步数.注意到剩下的步数不足以跳到c所在的重链的顶点1处,开始暴力扩展.c先跳到其父亲节点b处,b再跳到其父亲节点a处,此时步数用完,故节点g处的外卖站能给g到a的路径上的节点送外卖.最坏的情况是每个节点剩下的步数都是跳到其所在重链的顶点的步数-1,则都需暴力扩展,时间复杂度为$O(n^2)$,显然TLE.

考虑进一步优化.设节点u的DFS序为$dfn[u] = v$,设$idfn[v] = u$.注意到重链上的节点的DFS序连续,则对节点u剩下的步数tmp不足以跳到所在重链的顶点$top[u]$的情况,其能跳到的最高节点即DFS序为$dfn[u] - tmp$的节点,即节点$v = idfn[\max\{1, dfn[u] - tmp\}]$,其中与1取max是为了防止$dfn[u] - tmp$减成非正数.

总时间复杂度$O(n \log^2 n)$,最坏约$2e6 \times 20^2 = 8e8$,树剖常数小,牛客的评测机最快$1e9/s$,可过.

代码

```cpp
const int MAXN = 2e6 + 6, MAXM = MAXN << 1;
int n;  // 节点数
int d[MAXN];  // 最长距离
int head[MAXN], edge[MAXM], nxt[MAXM], idx;
ll w[MAXN];
int dfn[MAXN], cnt, idfn[MAXN];  // 节点的DFS序,dfn[u]=v,idfn[v]=u
ll neww[MAXN];  // neww[i]表示DFS序为i的节点的权值
int depth[MAXN];  // 节点的深度
int siz[MAXN];  // siz[u]表示以节点u为根节点的子树的大小
int top[MAXN];  // top[u]表示节点u所在的重链的顶点
int father[MAXN];  // 每个节点的父亲节点
int son[MAXN];  // 每个树中节点的重儿子
struct Node {
  int l, r;
  ll lazy;
  ll sum;  // 区间和
}SegT[MAXN << 2];

void add(int a, int b) {
  edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
}

void dfs1(int u, int fa, int dep) {  // 求树中节点的重儿子:当前节点为u,其父亲节点为fa,u的深度为dep
  depth[u] = dep, father[u] = fa, siz[u] = 1;
  for (int i = head[u]; ~i; i = nxt[i]) {
    int j = edge[i];
    if (j == fa) continue;

    dfs1(j, u, dep + 1);
    siz[u] += siz[j];
    if (siz[son[u]] < siz[j]) son[u] = j;
  }
}

void dfs2(int u, int to) {  // 找到每条重链:当前节点u所在的重链顶点为to
  dfn[u] = ++cnt, idfn[cnt] = u, neww[cnt] = w[u], top[u] = to;
  if (!son[u]) return;  // 叶子节点

  dfs2(son[u], to);  // 优先搜重儿子,重儿子与u在同一条重链上

  for (int i = head[u]; ~i; i = nxt[i]) {  // 搜轻儿子
    int j = edge[i];
    if (j == father[u] || j == son[u]) continue;

    dfs2(j, j);  // 轻儿子是自己所在的重链的顶点
  }
}

void push_up(int u) {
  SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
}

void build(int u, int l, int r) {
  SegT[u] = { l,r,0,neww[r] };
  if (l == r) return;

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

```

```cpp
void push_down(int u) {
  auto& root = SegT[u], & left = SegT[u << 1], & right = SegT[u << 1 | 1];
  if (root.lazy) {
    left.lazy += root.lazy, left.sum += root.lazy * (left.r - left.l + 1);
    right.lazy += root.lazy, right.sum += root.lazy * (right.r - right.l + 1);
    root.lazy = 0;
  }
}

void modify(int u, int l, int r, int k) {  // [l,r]+=k
  if (l <= SegT[u].l && r >= SegT[u].r) {
    SegT[u].lazy += k;
    SegT[u].sum += (ll)k * (SegT[u].r - SegT[u].l + 1);
    return;
  }

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify(u << 1, l, r, k);
  if (r > mid) modify(u << 1 | 1, l, r, k);
  push_up(u);
}

void modify_path(int u, int v, int k) {  // 节点u到v的路径上的节点权值+=k
  while (top[u] != top[v]) {  // 节点u和v不在一条重链上
    if (depth[top[u]] < depth[top[v]]) swap(u, v);  // 保证节点u比v更深
    modify(1, dfn[top[u]], dfn[u], k);
    u = father[top[u]];
  }

  // 节点u和v在同一条重链上
  if (depth[u] < depth[v]) swap(u, v);  // 保证节点u比节点v更深
  modify(1, dfn[v], dfn[u], k);
}

ll query(int u, int l, int r) {
  if (l <= SegT[u].l && r >= SegT[u].r) return SegT[u].sum;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  ll res = 0;
  if (l <= mid) res += query(u << 1, l, r);
  if (r > mid) res += query(u << 1 | 1, l, r);
  return res;
}

void solve() {
  memset(head, -1, so(head));

  cin >> n;
  for (int i = 0; i < n - 1; i++) {
    int a, b; cin >> a >> b;
    add(a, b), add(b, a);
  }

  dfs1(1, -1, 1);  // 从根节点开始搜,根节点无父亲节点,根节点的深度为1
  dfs2(1, 1);  // 从根节点开始搜,根节点所在的重链的顶点是根节点
  build(1, 1, n);

  for (int i = 1; i <= n; i++) cin >> d[i];

  for (int i = 1; i <= n; i++) {
    int u = i;
    int tmp = d[u];
    while (tmp && depth[u] - depth[top[u]] <= tmp) {
      tmp -= depth[u] - depth[top[u]];
      u = top[u];
      if (u == 1 || u == 0) break;
      else if (tmp) {  // 注意防止tmp减成负数
        u = father[u];
        tmp--;
      }
    }
    u = max(1, u);  // 防止u变为0
    if (tmp) u = idfn[max(1, dfn[u] - tmp)];
    modify_path(i, u, 1);
  }

  for (int i = 1; i <= n; i++) cout << query(1, dfn[i], dfn[i]) << ' ';
}

int main() {
```

```
144   solve();
145 }
```

17.3.4 Propagating tree

原题指路:https://codeforces.com/problemset/problem/383/C

题意 (2 s)

给定一棵包含编号$1 \sim n$的n个节点的树,其中1号节点为根节点.每个节点有一个权值,初始时i号节点的权值为a_i.

现有如下两种操作:

①$1\ u\ val$,表示令节点u的权值$+= val$.特别地,若节点u的权值$+= val$,则其所有儿子节点的权值$-= val$,该过程是递归的.

②$2\ u$,表示询问节点u的权值.

第一行输入两个整数$n, m\ \ (1 \le n, m \le 2e5)$,分别表示节点数、询问数.第二行输入$n$个整数$a_1, \cdots, a_n\ \ (1 \le a_i \le 1000)$.接下来$(n-1)$行每行输入两个整数$u, v\ \ (1 \le u, v \le n, u \ne v)$,表示节点$u$与节点$v$间存在无向边.接下来$m$行每行输入一个询问,格式如上,其中$1 \le u \le n, 1 \le val \le 1000$.数据保证给定的边构成一棵树.

思路

考虑树剖后用线段树维护.对操作①,显然以节点u为根节点的子树中除u外的其他节点的权值应$+= val$还是$-= val$与该节点的深度与u的深度的奇偶性有关.考虑如何维护懒标记,朴素的想法是修改节点u的权值时,与u的深度的奇偶性相同的子孙节点的权值$+= val$,与u的深度的奇偶性不同的子孙节点的权值$-= val$.但注意到操作不同节点时,同一子孙节点与子树的根节点的深度的奇偶性关系可能不同,故懒标记无法合并.

注意到节点在整棵树中的深度的奇偶性确定,考虑用节点在整棵树中的深度的奇偶性来统一节点权值的修改与深度的奇偶性的关系.线段树的叶子节点记录树的节点的深度,在操作①和下传懒标记时,统一奇数深度的节点权值$+= val$,偶数深度的节点的权值$-= val$.

代码

```cpp
const int MAXN = 2e5 + 5, MAXM = MAXN << 1;
namespace HeavyPathDecomposition {
  int head[MAXN], edge[MAXM], nxt[MAXM], idx;
  int dfn[MAXN], idfn[MAXN], tim;  // 节点的DFS序、DFS序对应的节点、时间戳
  int siz[MAXN];  // siz[u]表示以u为根节点的子树的大小
  int depth[MAXN];  // 节点的深度
  int father[MAXN];  // 节点的父亲节点
  int top[MAXN];  // 节点所在重链的顶点
  int son[MAXN];  // son[u]表示以u为根节点的子树中的重儿子节点

  void add(int a, int b) {
    edge[idx] = b, nxt[idx] = head[a], head[a] = idx++;
  }

  void dfs1(int u, int fa) {  // 求树中节点的重儿子:当前节点、前驱节点
    siz[u] = 1, father[u] = fa, depth[u] = depth[fa] + 1;
    for (int i = head[u]; ~i; i = nxt[i]) {
      int v = edge[i];
      if (v == fa) continue;

      dfs1(v, u);
      siz[u] += siz[v];
      if (siz[son[u]] < siz[v]) son[u] = v;  // 更新重儿子节点
    }
  }

  void dfs2(int u, int tp) {  // 求出每条重链:当前节点u所在的重链的顶点为tp
    dfn[u] = ++tim, idfn[tim] = u, top[u] = tp;
    if (!son[u]) return;  // u是叶子节点

    dfs2(son[u], tp);  // 优先搜重儿子节点,重儿子与u在同一条重链上

    for (int i = head[u]; ~i; i = nxt[i]) {  // 搜轻儿子节点
      int v = edge[i];
      if (v != father[u] && v != son[u])  // 不是往回搜或重儿子
        dfs2(v, v);  // 轻儿子是自身所在重链的顶点
    }
  }
}
using namespace HeavyPathDecomposition;

namespace SegmentTree {
  int n;
  int a[MAXN];
  struct Node {
    int l, r;
    int w;  // 线段树的叶子节点所维护的树的节点的权值
    int depth;  // 线段树的叶子节点所维护的树的节点的深度
    int lazy;  // 区间加懒标记
```

```
  }SegT[MAXN << 2];

  void build(int u, int l, int r) {
    SegT[u] = { l,r,0,0,0 };
    if (l == r) {
      SegT[u].w = a[idfn[l]], SegT[u].depth = depth[idfn[l]];
      return;
    }

    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  }

  void init() {
    dfs1(1, 0);  // 从根节点开始搜,根节点无前驱节点
    dfs2(1, 1);  // 从根节点开始搜,根节点是自身所在重链的顶点
    build(1, 1, n);
  }

  void pushDown(int u) {
    if (SegT[u].lazy) {
      SegT[u << 1].lazy += SegT[u].lazy, SegT[u << 1 | 1].lazy += SegT[u].lazy;
      if (SegT[u << 1].depth) {  // 左儿子节点是线段树的叶子节点
        if (SegT[u << 1].depth & 1) SegT[u << 1].w += SegT[u].lazy;
        else SegT[u << 1].w -= SegT[u].lazy;
      }
      if (SegT[u << 1 | 1].depth) {  // 右儿子节点是线段树的叶子节点
        if (SegT[u << 1 | 1].depth & 1) SegT[u << 1 | 1].w += SegT[u].lazy;
        else SegT[u << 1 | 1].w -= SegT[u].lazy;
      }
      SegT[u].lazy = 0;
    }
  }

  void modify(int u, int l, int r, int val) {  // 区间[l,r]±val
    if (l <= SegT[u].l && SegT[u].r <= r) {
      SegT[u].lazy += val;
      if (SegT[u].depth) {  // 该节点是线段树的叶子节点
        if (SegT[u].depth & 1) SegT[u].w += val;
        else SegT[u].w -= val;
      }
      return;
    }

    pushDown(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r, val);
    if (r > mid) modify(u << 1 | 1, l, r, val);
  }

  int query(int u, int pos) {  // 查询DFS序为pos的节点当前的权值
    if (SegT[u].l == SegT[u].r) return SegT[u].w;

    pushDown(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (pos <= mid) return query(u << 1, pos);
    else return query(u << 1 | 1, pos);
  }
}
using namespace SegmentTree;

void solve() {
  memset(head, -1, sizeof(head));

  int m; cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i < n; i++) {
    int u, v; cin >> u >> v;
    add(u, v), add(v, u);
  }

  init();

  while (m--) {
    int op, u; cin >> op >> u;
    if (op == 1) {
      int val; cin >> val;

      if (depth[u] & 1) modify(1, dfn[u], dfn[u] + siz[u] - 1, val);
      else modify(1, dfn[u], dfn[u] + siz[u] - 1, -val);
    }
    else cout << query(1, dfn[u]) << endl;
```

```
  }
}

int main() {
  solve();
}
```

17.4 线段树和树状数组优化DP

17.4.1 清理班次1

题意

有n头奶牛在工作.一天分为$1\sim t$共t个班次,每头奶牛只能在一天中的某个时间段内不间断地工作.求一个合适的奶牛班次使得每个班次都有奶牛在工作,且所需的奶牛总数尽可能少.

第一行输入整数$n,t\ \ (1\le n\le 2.5e4,1\le t\le 1e6)$.接下来$n$行每行输入两个整数,表示每头奶牛可进行工作的开始和结束时间.

若能做到每个班次都有奶牛工作,则输出所需的奶牛总数的最小值;否则输出-1.

思路

即给定若干个区间,选出最少的区间覆盖$[1,t]$.可贪心.

下面描述DP做法.$dp[i]$表示所有只用左端点在i的左边的区间且能覆盖$[1,i]$的所有方案中所选区间数的最小值.以最后一个选择的区间$[l,r]$分类,则之前选择的区间至少应覆盖$[1,l-1]$,则倒数第二个区间的右端点$k\in[l-1,r)$,进而$dp[i]=\min\limits_{l-1\le k<r}dp[k]+1$.显然有些整点$x$处无以它结尾的区间,初始化$dp[x]=INF$.

转移需枚举区间长度,每个区间的长度最坏为n,则总时间复杂度$O(nt)$,会T.考虑优化,注意到上述过程有两个操作:①查询区间最小值;②在区间右端点插入一个值,显然可用线段树优化,两操作的时间复杂度都变为$O(\log t)$.

代码

```
const int MAXN = 2.5e4 + 5, MAXT = 1e6 + 5;
int n, t;  // 区间数、要覆盖的区间[1,t]
struct Range {
  int l, r;

  bool operator<(const Range& t)const { return r < t.r; }  // 按区间右端点升序排列
}ranges[MAXN];

struct Node {
  int l, r;
  int minnum;
}SegT[MAXT << 2];

void build(int u, int l, int r) {
  SegT[u] = { l,r,INF };  // 等价于将原数组的每个节点的dp值都初始化为INF
  if (l == r) return;

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}

void push_up(int u) {
  SegT[u].minnum = min(SegT[u << 1].minnum, SegT[u << 1 | 1].minnum);
}

void modify(int u, int pos, int v) {
  if (SegT[u].l == SegT[u].r) {
    SegT[u].minnum = min(SegT[u].minnum, v);
    return;
  }

  int mid = SegT[u].l + SegT[u].r >> 1;
  if (pos <= mid) modify(u << 1, pos, v);
  else modify(u << 1 | 1, pos, v);

  push_up(u);
}

int query(int u, int l, int r) {  // 查询[l,r]中的最小值
  if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u].minnum;

  int res = INF;
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) res = query(u << 1, l, r);
```

```
  if (r > mid) res = min(res, query(u << 1 | 1, l, r));
  return res;
}

int main() {
  cin >> n >> t;

  build(1, 0, t);  // 对要覆盖的线段建线段树

  for (int i = 0; i < n; i++) cin >> ranges[i].l >> ranges[i].r;
  sort(ranges, ranges + n);

  modify(1, 0, 0);  // 初始化dp[0]=0
  for (int i = 0; i < n; i++) {
    int l = ranges[i].l, r = ranges[i].r;
    int tmp = query(1, l - 1, r - 1) + 1;  // dp[r]
    modify(1, r, tmp);
  }

  int ans = query(1, t, t);
  cout << (ans == INF ? -1 : ans);
}
```

17.4.2 清理班次2

题意

有编号$1\sim n$的n头奶牛在工作.一天分为若干个班次,其中第m到第e个班次(包含这两个班次)间必须有奶牛工作.第i头奶牛可从第a_i个班次工作到第b_i个班次,需要c_i的佣金.求一个合适的奶牛班次使得$[m,e]$中的每个班次都有奶牛在工作,且所需的佣金最少.

第一行输入三个整数n,m,e $(1\le n\le 1e4, 0\le m,e\le 86399)$.接下来$n$行每行输入三个整数$a_i,b_i,c_i$ $(m\le a_i\le b_i\le e, 0\le c_i\le 5e5)$.

若能做到$[m,e]$中的每个班次都有奶牛工作,则输出所需的佣金的最小值;否则输出-1.

思路

区间多了权重后无法用贪心求解,但依旧能用DP求解.

$dp[i]$表示所有只用左端点在i的左边的区间且能覆盖$[m,i]$的所有方案中花费的最小值.$dp[i]=\min\limits_{l_t-1\le k<r_t} dp[k]+w_t$.

代码

```
const int MAXN = 1e4 + 5, MAXM = 9e4 + 5;
int n, m, e;  // 区间数、工作时间段[m,e]
struct Range {
  int l, r;
  int w;

  bool operator<(const Range& t)const { return r < t.r; }
}ranges[MAXN];
struct Node {
  int l, r;
  ll mincost;  // 最小花费
}SegT[MAXM << 2];

void push_up(int u) {
  SegT[u].mincost = min(SegT[u << 1].mincost, SegT[u << 1 | 1].mincost);
}

void build(int u, int l, int r) {
  SegT[u] = { l,r,INFF };
  if (l == r) return;

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}

void modify(int u, int pos, ll v) {
  if (SegT[u].l == SegT[u].r) {
    SegT[u].mincost = min(SegT[u].mincost, v);
    return;
  }

  int mid = SegT[u].l + SegT[u].r >> 1;
  if (pos <= mid) modify(u << 1, pos, v);
  else modify(u << 1 | 1, pos, v);

  push_up(u);
}
```

```
ll query(int u, int l, int r) {  // 查询[l,r]的最小值
  if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u].mincost;

  ll res = INFF;
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) res = query(u << 1, l, r);
  if (r > mid) res = min(res, query(u << 1 | 1, l, r));
  return res;
}

int main() {
  cin >> n >> m >> e;

  build(1, m - 1, e);  // 只对需要覆盖的线段建线段树

  for (int i = 0; i < n; i++) {
    int l, r, w; cin >> l >> r >> w;
    ranges[i] = { l,r,w };
  }
  sort(ranges, ranges + n);

  modify(1, m - 1, 0);  // 初始化dp[m-1]=0
  for (int i = 0; i < n; i++) {
    int l = ranges[i].l, r = ranges[i].r, w = ranges[i].w;
    ll tmp = query(1, l - 1, r - 1) + w;  // dp[r]
    modify(1, r, tmp);
  }

  ll ans = query(1, e, e);
  cout << (ans == INFF ? -1 : ans);
}
```

17.4.3 严格上升子列

题意

有t $(1 \le t \le 100)$组测试数据.每组测试数据给定一个长度为n的元素的绝对值不超过1e9的序列a,求a有多少个长度为m $(1 \le m \le n \le 1000)$的严格上升子列.数据保证所有测试数据的$n \times m$之和不超过1e7.

对每组测试数据x,输出"Case #x: ans",答案对$1e9+7$取模.

思路

$dp[i][j]$表示$[1,i]$中以a_i结尾且长度为j的严格上升子列的数量.以倒数第二个数为$a_1,\cdots,a_{i-1}$分类,但每一类未必存在.设倒数第二个数为a_k,则a_i前面的集合是以a_k结尾且长度为$(j-1)$的严格上升子列的数量,即$dp[k][j-1]$.故$dp[i][j]=\sum_k dp[k][j-1]$ $(1 \le k < i, a_k < a_i)$.初始条件$dp[i][1]=1$ $(1 \le i \le n)$.

```
for (int i = 1; i <= n; i++) {
  for (int j = 2; j <= m; j++) {
    for (int k = 1; k < i; i++)
      if (a[k] < a[i]) dp[i][j] += dp[k][j - 1];
  }
}
```

状态数$1e3 \times 1e3$,每次计算需枚举k,时间复杂度1e9,会TLE.考虑优化,注意到上述代码的第三重循环的功能是求$dp[1][j-1] \sim dp[i-1][j-1]$中所有满足$a_k < a_i$的$dp[k][j-1]$之和,所需的操作:①动态求区间前缀和;②插入一个数,显然可用Splay或离散化+BIT优化.以后者为例,时间复杂度$O(n^2 \log n)$.

注意计算$dp[][j]$时需用到$dp[][j-1]$,若用上述代码的循环顺序,则需同时维护m个BIT.不妨先枚举j,这样这样同时只需维护一个BIT.

代码

```
const int MAXN = 1005;
const int MOD = 1e9 + 7;
int n, m;  // 序列长度、严格上升序列的长度
int a[MAXN];
int nums[MAXN], cnt = 0;  // 离散化后的数组
int BIT[MAXN];
int dp[MAXN][MAXN];  // 表示[1,i]中以a_i结尾且长度为j的严格上升子列的数量

void add(int pos, int v) {
  for (int i = pos; i <= cnt; i += lowbit(i)) BIT[i] = (BIT[i] + v) % MOD;
}

int query(int pos) {  // 询问[1...pos]的前缀和
  int res = 0;
  for (int i = pos; i; i -= lowbit(i)) res = ((ll)res + BIT[i]) % MOD;
  return res;
}
```

```
int main() {
  int t; cin >> t;
  for (int C = 1; C <= t; C++) {
    cin >> n >> m;

    cnt = 0;
    for (int i = 1; i <= n; i++) {
      cin >> a[i];
      nums[cnt++] = a[i];
    }

    // 离散化
    sort(nums, nums + cnt);
    cnt = unique(nums, nums + cnt) - nums;
    for (int i = 1; i <= n; i++) a[i] = lower_bound(nums, nums + cnt, a[i]) - nums + 1;  // 下标从1开始

    for (int i = 1; i <= n; i++) dp[i][1] = 1;  // 初始化

    for (int j = 2; j <= m; j++) {  // 先枚举长度
      for (int i = 1; i <= cnt; i++) BIT[i] = 0;  // 清空BIT

      for (int i = 1; i <= n; i++) {
        dp[i][j] = query(a[i] - 1);
        add(a[i], dp[i][j - 1]);
      }
    }

    int ans = 0;
    for (int i = 1; i <= n; i++) ans = ((ll)ans + dp[i][m]) % MOD;
    printf("Case #%d: %d\n", C, ans);
  }
}
```

17.4.4 The Bakery

原题指路:https://codeforces.com/contest/833/problem/B

题意 (2.5 s)

给定一个长度为n $(1 \leq n \leq 35000)$的序列$a_1, \cdots, a_n$ $(1 \leq a_i \leq n)$.定义一个区间的价值为其中不同的数的个数.将序列划分为k $(1 \leq k \leq \min\{n, 50\})$个不相交的区间,求所有区间的价值之和的最大值.

思路

$cnt[l][r]$表示区间$[l, r]$中不同的数的个数,$dp[i][j]$表示将区间$[1, i]$划分为j段的最大价值之和.状态转移方程$dp[i][j] = \max\limits_{1 \leq k \leq i}\{dp[k][j-1] + cnt[k+1][i]\}$.

考虑如何快速求$cnt[l][r]$.注意到一种颜色对答案的贡献区间是它上一次出现的位置(初始时所有颜色上一次出现的位置都为下标0)到本次出现位置的前一位,这一步可用线段树区间加加速.

取max的部分显然可用线段树优化,在算每一层$dp[][j]$时将线段树清空,再对上一层的结果$dp[][j-1]$建线段树即可.

代码

```
const int MAXN = 35005;
namespace SegmentTree {
  struct Node {
    int l, r;
    int maxnum;  // 区间max
    int add;
  }SegT[MAXN << 2];

  void pushUp(int u) {
    SegT[u].maxnum = max(SegT[u << 1].maxnum, SegT[u << 1 | 1].maxnum);
  }

  void build(int u, int l, int r, vector<int>& dp) {
    SegT[u] = { l,r,0,0 };
    if (l == r) {
      SegT[u].maxnum = dp[l];
      return;
    }

    int mid = l + r >> 1;
    build(u << 1, l, mid, dp), build(u << 1 | 1, mid + 1, r, dp);
    pushUp(u);
  }

  void pushDown(int u) {
```

```
    if (SegT[u].add) {
      SegT[u << 1].maxnum += SegT[u].add, SegT[u << 1].add += SegT[u].add;
      SegT[u << 1 | 1].maxnum += SegT[u].add, SegT[u << 1 | 1].add += SegT[u].add;
      SegT[u].add = 0;
    }
  }

  void modify(int u, int l, int r, int x) {  // [l,r]+=x
    if (l <= SegT[u].l && SegT[u].r <= r) {
      SegT[u].maxnum += x, SegT[u].add += x;
      return;
    }

    pushDown(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r, x);
    if (r > mid) modify(u << 1 | 1, l, r, x);
    pushUp(u);
  }

  int query(int u, int l, int r) {  // 查询[l,r]的区间max
    if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].maxnum;

    pushDown(u);
    int mid = SegT[u].l + SegT[u].r >> 1;
    int res = 0;
    if (l <= mid) res = max(res, query(u << 1, l, r));
    if (r > mid) res = max(res, query(u << 1 | 1, l, r));
    return res;
  }
}
using namespace SegmentTree;

void solve() {
  int n, k; cin >> n >> k;
  vector<int> a(n + 1);
  for (int i = 1; i <= n; i++) cin >> a[i];

  vector<int> dp(n + 1);  // dp[i]表示只考虑前i个物品时的最大收益
  for (int j = 1; j <= k; j++) {  // 枚举划分的段数
    build(1, 0, n, dp);

    vector<int> ndp(n + 1), last(n + 1);  // last[x]表示值x最后一次出现的下标
    for (int i = 1; i <= n; i++) {
      modify(1, last[a[i]], i - 1, 1);
      last[a[i]] = i;
      ndp[i] = query(1, 0, i - 1);
    }
    dp = ndp;
  }
  cout << dp[n] << endl;
}

int main() {
  solve();
}
```

17.4.6 LIS

原题指路: https://atcoder.jp/contests/chokudai_S001/tasks/chokudai_S001_h

题意

给定一个长度为 n $(1 \leq n \leq 1e5)$ 的整数序列 $a = [a_1, \cdots, a_n]$ $(1 \leq a_i \leq 1e9)$, 求其最长严格上升子列的长度.

代码

```
struct FenwickTree {
    int n;
    vector<int> BIT;

    FenwickTree(int _n) :n(_n), BIT(n + 5) {}

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

    void modify(int pos, int val) {  // a[pos] = val
        for (int i = pos; i <= n; i += lowbit(i))
```

```
            BIT[i] = max(BIT[i], val);
    }

    int query(int pos) {  // 求前缀a[1...pos]的最大值
        int res = 0;
        for (int i = pos; i; i -= lowbit(i))
            res = max(res, BIT[i]);
        return res;
    }
};

void solve() {
    int n; cin >> n;
    vector<int> a(n + 5), dis;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        dis.push_back(a[i]);
    }

    sort(all(dis));
    dis.erase(unique(all(dis)), dis.end());
    for (int i = 1; i <= n; i++)
        a[i] = lower_bound(all(dis), a[i]) - dis.begin() + 1;

    vector<int> dp(n + 5);
    FenwickTree bit(n);
    for (int i = 1; i <= n; i++)
        bit.modify(a[i], dp[i] = bit.query(a[i]) + 1);

    cout << *max_element(all(dp)) << endl;
}

int main() {
    solve();
}
```

17.4.5 最长上升子序列

原题指路: https://www.luogu.com.cn/problem/P4309

题意

给定一个序列, 初始时为空. 给定一个整数 n $(1 \leq n \leq 1e5)$, 现将整数 $1 \sim n$ 依次插入序列中, 第 k $(1 \leq k \leq n)$ 个数插入到给定的位置 pos $(0 \leq pos \leq k-1)$. 每插入一个数后, 求序列的LIS的长度.

思路

用vector a 记录每个下标插入的数, 其中在下标 pos 插入数 val 表示为 $a[pos] = val$.

$dp[i]$ 表示以下标 i 结尾的LIS的长度. 因序列元素互异, 则插入数 $a[pos] = i$ 后的答案为 $\max\limits_{1 \leq j \leq i} dp[j]$.

插入 $a[pos] = i$ 时, 状态转移方程 $dp[i] = \max\limits_{1 \leq j \leq i} dp[j] + 1$. 状态数 $O(n)$, 转移 $O(n)$, 会TLE.

考虑优化, 注意到转移时需求前缀max, 可用BIT维护 $dp[]$, 时间复杂度 $O(n \log n)$.

代码

```
struct FenwickTree {
    int n;
    vector<int> BIT;

    FenwickTree(int _n) :n(_n), BIT(n + 5) {}

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

    void modify(int pos, int val) {  // a[pos] = val
        for (int i = pos; i <= n; i += lowbit(i))
            BIT[i] = max(BIT[i], val);
    }

    int query(int pos) {  // 求前缀a[1...pos]的最大值
        int res = 0;
        for (int i = pos; i; i -= lowbit(i))
            res = max(res, BIT[i]);
        return res;
    }
};
```

```
23
24  void solve() {
25      vector<int> a;
26      int n; cin >> n;
27      for (int i = 1; i <= n; i++) {
28          int pos; cin >> pos;
29          a.insert(a.begin() + pos, i);
30      }
31
32      FenwickTree bit(n);
33      vector<int> dp(n + 5);
34      for (int i = 0; i < n; i++)
35          bit.modify(a[i], dp[a[i]] = bit.query(a[i]) + 1);
36
37      for (int i = 1; i <= n; i++)
38          cout << (dp[i] = max(dp[i], dp[i - 1])) << endl;
39  }
40
41  int main() {
42      solve();
43  }
```

17.5 线段树维护矩阵

17.5.1 New Year and Old Subsequence

原题指路:https://codeforces.com/contest/750/problem/E

题意 (3 s)

称一个字符串是好的,如果"2017"是它的子串,"2016"不是它的子串.给定一个字符串s,下标从1开始,询问将子串$s[l...r]$变为好的至少需要移除多少个字符,若无法通过移除字符使得它变为好的,输出-1.

第一行输入两个整数n, q $(4 \le n \le 2e5, 1 \le q \le 2e5)$,分别表示字符串长度和询问数.第二行输入一个长度为n且只包含字符'0'~'9'的字符串s.接下来q行每行输入两个整数l, r $(1 \le l \le r \le n)$,表示询问将子串$s[l...r]$变为好的至少需要移除多少个字符.

思路

将2017拆分为五种状态:$\varnothing, 2, 20, 201, 2017$,分别用$0, 1, 2, 3, 4$表示.

对字符串中的每个字符,用一个矩阵表示该字符的影响.具体地,对一个5×5矩阵M,M_{ij}表示从字符串$s[1..(i-1)]$从状态i变到字符串$s[1..i]$的状态j所需的最小步数,若不能变到则步数为∞.

以字符'2'为例.设当前遍历到字符$s[i]$,则$M = \begin{bmatrix} 1 & 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$,其中$M_{00} = 1$表示从$s[1..(i-1)]$的状态0(即空串)变到$s[1..i]$的状态0(即空串)需删除一个字符'2';$M_{01} = 0$表示从$s[1..(i-1)]$的状态0(即空串)变到$s[1..i]$的状态1(即"2")需保留字符'2';$M_{02} = \infty$示从$s[1..(i-1)]$的状态0(即空串)变到$s[1..i]$的状态2(即"20")是不可能的,因为只有一个字符'2'.同理可写出字符'0'、'1'、'7'的矩阵.

对字符'6',因不能出现子串"2016",则从$s[1..(i-1)]$的状态3(即"201")变到$s[1..i]$的状态4需删除一个字符'6',故$M = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 1 & \infty \\ \infty & \infty & \infty & \infty & 1 \end{bmatrix}$.

显然对字符'3'、'4'、'5'、'8'、'9',转移矩阵都是$M = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$.

每个询问的答案即将区间内所有矩阵相加.

代码

```
1   const int MAXN = 2e5 + 5;
2   int n, q;  // 字符串长度、操作数
3   char str[MAXN];
4
5   struct Matrix {
6     static const int MAXSIZE = 5;
7     int a[MAXSIZE][MAXSIZE];
8
9     Matrix() { memset(a, INF, so(a)); }
10
11    void init(int x) {  // 初始化数字x对应的转移矩阵
```

```
    for (int i = 0; i < 5; i++) a[i][i] = 0;

    switch (x) {
      case 2: a[0][0] = 1, a[0][1] = 0; break;
      case 0: a[1][1] = 1, a[1][2] = 0; break;
      case 1: a[2][2] = 1, a[2][3] = 0; break;
      case 7: a[3][3] = 1, a[3][4] = 0; break;
      case 6: a[3][3] = 1, a[4][4] = 1; break;
    }
  }

  Matrix operator+(const Matrix& B)const {
    Matrix res;
    for (int k = 0; k < MAXSIZE; k++) {  // 枚举分界点
      for (int i = 0; i < MAXSIZE; i++) {
        for (int j = 0; j < MAXSIZE; j++)
          res.a[i][j] = min(res.a[i][j], a[i][k] + B.a[k][j]);
      }
    }
    return res;
  }
};

struct Node {
  int l, r;
  Matrix matrix;
}SegT[MAXN << 2];

void push_up(int u) {
  SegT[u].matrix = SegT[u << 1].matrix + SegT[u << 1 | 1].matrix;
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r;
  if (l == r) {
    SegT[u].matrix.init(str[l] - '0');
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

Matrix query(int u, int l, int r) {  // 求子串s[l...r]从状态0到状态4的最小步数
  if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].matrix;

  if (r <= SegT[u << 1].r) return query(u << 1, l, r);
  else if (l >= SegT[u << 1 | 1].l) return query(u << 1 | 1, l, r);
  else return query(u << 1, l, r) + query(u << 1 | 1, l, r);
}

void solve() {
  cin >> n >> q >> str + 1;

  build(1, 1, n);

  while (q--) {
    int l, r; cin >> l >> r;
    auto ans = query(1, l, r);
    cout << (ans.a[0][4] <= n ? ans.a[0][4] : -1) << endl;
  }
}

int main() {
  solve();
}
```

17.5.2 Sasha and Array

原题指路:https://codeforces.com/problemset/problem/719/E

题意 (3 s)

维护一个序列$a_1, \cdots, a_n$,支持如下两种操作:①1 l r x,表示令所有$i \in [l, r]$的$a_i += x$;②2 l r,表示询问$\sum_{i=l}^{r} f(a_i)$,答案对1e9 + 7取模,其中$f(x)$表示Fibonacci数列的第x项,$f(1) = 1, f(2) = 1, f(x) = f(x-1) + f(x-2)\ \ (x > 2)$.

第一行输入两个整数n, m $(1 \leq n, m \leq 1e5)$,分别表示序列长度和操作数.第二行输入n个整数$a_1, \cdots, a_n$ $(1 \leq a_i \leq 1e9)$.接下来m行每行输入一个操作,格式如上.数据保证$1 \leq l \leq r \leq n, 1 \leq x \leq 1e9$.

思路

注意到操作①加的数总是正的,即等价于将a_i $(l \leq i \leq r)$变为Fibonacci数列后面的项,可用线段树直接维护$f(a_i)$ $(1 \leq i \leq n)$.

注意到$\begin{bmatrix} f_n & f_{n+1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} f_{n+1} & f_{n+2} \\ 0 & 0 \end{bmatrix}$,取初始矩阵$origin = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$,转移矩阵$trans = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$,可用矩阵快速幂求出Fibonacci数列对应的项.

线段树节点维护懒标记、区间和即可,注意矩阵初始化为单位矩阵.

代码

```cpp
const int MAXN = 1e5 + 5;
const int MOD = 1e9 + 7;
int n, m;  // 字符串长度、操作数
int a[MAXN];

struct Matrix {
  static const int MAXSIZE = 2;
  int a[MAXSIZE][MAXSIZE];

  Matrix() { memset(a, 0, so(a)); }

  void init_identity() {  // 初始化为单位矩阵
    memset(a, 0, so(a));
    for (int i = 0; i < MAXSIZE; i++) a[i][i] = 1;
  }

  Matrix operator+(const Matrix& B)const {
    Matrix res;
    for (int i = 0; i < MAXSIZE; i++) {
      for (int j = 0; j < MAXSIZE; j++)
        res.a[i][j] = ((ll)a[i][j] + B.a[i][j]) % MOD;
    }
    return res;
  }

  Matrix operator*(const Matrix& B)const {
    Matrix res;
    for (int i = 0; i < MAXSIZE; i++) {
      for (int j = 0; j < MAXSIZE; j++) {
        for (int k = 0; k < MAXSIZE; k++)
          res.a[i][j] = ((ll)res.a[i][j] + (ll)a[i][k] * B.a[k][j]) % MOD;
      }
    }
    return res;
  }

  Matrix operator^(int k)const {  // 快速幂
    Matrix res;
    res.init_identity();  // 单位矩阵
    Matrix tmpa;  // 存放矩阵a[][]的乘方
    memcpy(tmpa.a, a, so(a));

    while (k) {
      if (k & 1) res = res * tmpa;
      k >>= 1;
      tmpa = tmpa * tmpa;
    }
    return res;
  }
}origin, trans;  // 初始矩阵、转移矩阵

struct Node {
  int l, r;
  Matrix sum;  // 区间和
  Matrix cnt;  // 记录该区间乘了几次转移矩阵
  int lazy;  // 加法懒标记
}SegT[MAXN << 2];

void push_up(int u) {
  SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r, SegT[u].lazy = 0, SegT[u].cnt.init_identity();
  if (l == r) {
    SegT[u].sum = origin * (trans ^ (a[l] - 1));
    return;
  }
```

```
69
70    int mid = l + r >> 1;
71    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
72    push_up(u);
73  }
74
75  void push_down(int u) {
76    if (SegT[u].lazy) {
77      SegT[u << 1].sum = SegT[u << 1].sum * SegT[u].cnt;
78      SegT[u << 1].cnt = SegT[u << 1].cnt * SegT[u].cnt;
79      SegT[u << 1 | 1].sum = SegT[u << 1 | 1].sum * SegT[u].cnt;
80      SegT[u << 1 | 1].cnt = SegT[u << 1 | 1].cnt * SegT[u].cnt;
81      SegT[u << 1].lazy = SegT[u << 1 | 1].lazy = 1;
82      SegT[u].lazy = 0, SegT[u].cnt.init_identity();  // 注意cnt清空为单位矩阵
83    }
84  }
85
86  void modify(int u, int l, int r, Matrix mul) {  // 区间乘上矩阵mul
87    if (l <= SegT[u].l && SegT[u].r <= r) {
88      SegT[u].sum = SegT[u].sum * mul;
89      SegT[u].cnt = SegT[u].cnt * mul;
90      SegT[u].lazy = 1;
91      return;
92    }
93
94    push_down(u);
95    int mid = SegT[u].l + SegT[u].r >> 1;
96    if (l <= mid) modify(u << 1, l, r, mul);
97    if (r > mid) modify(u << 1 | 1, l, r, mul);
98    push_up(u);
99  }
100
101 int query(int u, int l, int r) {
102   if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].sum.a[0][1];  // 注意返回第一行第二个元素
103
104   push_down(u);
105   int mid = SegT[u].l + SegT[u].r >> 1;
106   int res = 0;
107   if (l <= mid) res = ((ll)res + query(u << 1, l, r)) % MOD;
108   if (r > mid) res = ((ll)res + query(u << 1 | 1, l, r)) % MOD;
109   return res;
110 }
111
112 void init() {  // 初始化origin[][]和trans[][]
113   origin.a[0][1] = 1;
114   trans.a[0][1] = trans.a[1][0] = trans.a[1][1] = 1;
115 }
116
117 void solve() {
118   init();
119
120   cin >> n >> m;
121   for (int i = 1; i <= n; i++) cin >> a[i];
122
123   build(1, 1, n);
124
125   while (m--) {
126     int op, l, r; cin >> op >> l >> r;
127     if (op == 1) {
128       int x; cin >> x;
129       modify(1, l, r, trans ^ x);
130     }
131     else cout << query(1, l, r) << endl;
132   }
133 }
134
135 int main() {
136   solve();
137 }
```

17.5.3 对线

原题指路:https://codeforces.com/gym/103186/problem/I

题意 (12 s)

有三排长度为n的兵线,每排兵线从左往右编号$1 \sim n$,同排同编号位置对齐.现有如下四个操作:

①$0\ x\ l\ r\ \ (x \in \{1,2,3\}, 1 \le l \le r \le n)$,表示询问第$x$排从$l$位置到$r$位置间的士兵数,答案对998244353取模.

②$1\ x\ l\ r\ y\ \ (x \in \{1,2,3\}, 1 \le l \le r \le n, 1 \le y \le 1e9)$,表示第$x$排从$l$位置到$r$位置都增加$y$个士兵.

③$2\ x\ y\ l\ r\ \ (x,y \in \{1,2,3\}, 1 \le l \le r \le n)$,表示交换第$x$排和第$y$排区间$[l,r]$上的士兵.

④$3\ x\ y\ l\ r\ \ (x,y \in \{1,2,3\}, 1 \le l \le r \le n)$,表示将第$x$排区间$[l,r]$上的士兵复制一份加到第$y$排对应位置.

思路

线段树节点维护:①$1 \times 4$的矩阵$sum[][]$,其中$sum[1][x]$表示第x排当前区间的区间和;②$4 \times 4$的矩阵$lazy[][]$,表示矩阵乘法的懒标记,初始化为单位矩阵.整体思路:每个4×4的矩阵左上角的3×3矩阵的每一列维护兵线的每一排,4×4的矩阵的第4行维护操作.

①操作,答案为$[l,r]$的区间和的$a[1][x]$元素.

②操作,根据Gauss消元解线性方程组中的"一行的若干倍加到另一行上",不妨取矩阵的第4行都为1,则该操作等价于将第4行的y倍加到第x行.如$x=1$时,转移矩阵$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\y&0&0&1\end{bmatrix}$;$x=2$时,转移矩阵$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&y&0&1\end{bmatrix}$;$x=3$时,转移矩阵.$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&y&1\end{bmatrix}$.

③操作,根据Gauss消元解线性方程组中的"交换两行",易知转移矩阵即单位矩阵交换第x行和第y行的结果.如$x=1,y=3$时,转移矩阵$\begin{bmatrix}0&0&1&0\\0&1&0&0\\1&0&0&0\\0&0&0&1\end{bmatrix}$.

④操作,根据Gauss消元解线性方程组中的"一行的若干倍加到另一行上",易知转移矩阵即单位矩阵的$a[x][y]$元素$+1$.如$x=1,y=3$时,转移矩阵$\begin{bmatrix}1&0&1&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$.

注意输入输出量大.

代码

```cpp
namespace FastIO {
#define gc() (p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, 1 << 21, stdin), p1 == p2) ? EOF : *p1++)  // 重写getchar()
#define pc(ch) (p - buf2 == SIZE ? fwrite(buf2, 1, SIZE, stdout), p = buf2, *p++ = ch : *p++ = ch)  // 重写putchar()
  char buf[1 << 23], * p1 = buf, * p2 = buf;

  template<typename T>
  void read(T& x) {  // 数字快读
    x = 0;
    T sgn = 1;
    char ch = gc();
    while (ch < '0' || ch > '9') {
      if (ch == '-') sgn = -1;
      ch = gc();
    }
    while (ch >= '0' && ch <= '9') {
      x = (((x << 2) + x) << 1) + (ch & 15);
      ch = gc();
    }
    x *= sgn;
  }

  const int SIZE = 1 << 21;
  int stk[40], top;
  char buf1[SIZE], buf2[SIZE], * p = buf2, * s = buf1, * t = buf1;

  template<typename T>
  void print_number(T x) {
    p = buf2;  // 复位指针p

    if (!x) {
      pc('0');
      return;
    }

    top = 0;  // 栈顶指针
    if (x < 0) {
      pc('-');
      x = ~x + 1;  // 取相反数
    }

    do {
      stk[top++] = x % 10;
```

```
      x /= 10;
    } while (x);
    while (top) pc(stk[--top] + 48);
  }

  template<typename T>
  void write(T x) {  // 数字快写
    print_number(x);
    fwrite(buf2, 1, p - buf2, stdout);
  }
};
using namespace FastIO;

const int MAXN = 3e5 + 5;
const int MOD = 998244353;

template<typename T>
struct Matrix {
  static const int MAXSIZE = 5;
  int n, m;  // 行数、列数
  T a[MAXSIZE][MAXSIZE];  // 下标从1开始

  Matrix() :n(0), m(0) { memset(a, 0, so(a)); }
  Matrix(int _n, int _m) :n(_n), m(_m) { memset(a, 0, so(a)); }

  void init_identity() {  // 初始化为单位矩阵
    assert(n == m);

    memset(a, 0, so(a));
    for (int i = 1; i <= n; i++) a[i][i] = 1;
  }

  Matrix<T> operator+(const Matrix<T>& B)const {
    assert(n == B.n), assert(m == B.m);

    Matrix<T> res(n, n);
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= m; j++)
        res.a[i][j] = ((ll)a[i][j] + B.a[i][j]) % MOD;
    }
    return res;
  }

  Matrix<T> operator-(const Matrix<T>& B)const {
    assert(n == B.n), assert(m == B.m);

    Matrix<T> res(n, n);
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= m; j++)
        res.a[i][j] = ((a[i][j] - B.a[i][j]) % MOD + MOD) % MOD;
    }
    return res;
  }

  Matrix<T> operator*(const Matrix<T>& B)const {
    assert(m == B.n);

    Matrix<T> res(n, B.m);
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= B.m; j++) {
        for (int k = 1; k <= m; k++)
          res.a[i][j] = ((ll)res.a[i][j] + (ll)a[i][k] * B.a[k][j]) % MOD;
      }
    }
    return res;
  }

  Matrix<T> operator^(int k)const {  // 快速幂
    assert(n == m);

    Matrix<T> res(n, n);
    res.init_identity();  // 单位矩阵
    Matrix<T> tmpa(n, n);  // 存放矩阵a[][]的乘方
    memcpy(tmpa.a, a, so(a));

    while (k) {
      if (k & 1) res = res * tmpa;
      k >>= 1;
      tmpa = tmpa * tmpa;
    }
    return res;
  }
```

```
  Matrix<T>& operator=(const Matrix<T>& B) {
    memset(a, 0, so(a));
    n = B.n, m = B.m;
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= m; j++) a[i][j] = B.a[i][j];
    return *this;
  }

  bool operator==(const Matrix<T>& B)const {
    if (n != B.n || m != B.m) return false;

    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= m; j++)
        if (a[i][j] != B.a[i][j]) return false;
    }
    return true;
  }

  void print() {
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= m; j++) cout << a[i][j] << " \n"[j == m];
  }
};

Matrix<ll> Imatrix = Matrix<ll>(4, 4);  // 单位矩阵

struct Node {
  int l, r;
  Matrix<ll> sum;  // 区间和
  Matrix<ll> lazy;  // 矩阵乘法标记
}SegT[MAXN << 2];

void push_up(int u) {
  SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r;
  SegT[u].sum = Matrix<ll>(4, 4);
  SegT[u].lazy = Matrix<ll>(4, 4);
  SegT[u].lazy.init_identity();  // 初始化为单位矩阵,表示无修改
  if (l == r) {
    SegT[u].sum.a[1][4] = 1;
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void push_down(int u) {
  if (SegT[u].lazy == Imatrix) return;  // 无懒标记

  SegT[u << 1].sum = SegT[u << 1].sum * SegT[u].lazy;
  SegT[u << 1].lazy = SegT[u << 1].lazy * SegT[u].lazy;
  SegT[u << 1 | 1].sum = SegT[u << 1 | 1].sum * SegT[u].lazy;
  SegT[u << 1 | 1].lazy = SegT[u << 1 | 1].lazy * SegT[u].lazy;
  SegT[u].lazy.init_identity();  // 清空为单位矩阵
}

void modify(int u, int l, int r, Matrix<ll>& mul) {
  if (l <= SegT[u].l && SegT[u].r <= r) {
    SegT[u].sum = SegT[u].sum * mul;
    SegT[u].lazy = SegT[u].lazy * mul;
    return;
  }

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify(u << 1, l, r, mul);
  if (r > mid) modify(u << 1 | 1, l, r, mul);
  push_up(u);
}

Matrix<ll> query(int u, int l, int r) {
  if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].sum;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  Matrix<ll> res(4, 4);
```

```
  if (l <= mid) res = res + query(u << 1, l, r);
  if (r > mid) res = res + query(u << 1 | 1, l, r);
  return res;
}

void solve() {
  Imatrix.init_identity();

  int n, q; read(n), read(q);
  build(1, 1, n);

  while (q--) {
    int op; read(op);
    if (op == 0) {  // 询问第x排[l,r]的区间和
      int x, l, r; read(x), read(l), read(r);
      write(query(1, l, r).a[1][x]), putchar('\n');
    }
    else if (op == 1) {  // 第x排[l,r]+=y
      int x, l, r, y; read(x), read(l), read(r), read(y);
      Matrix<ll> mul(4, 4);
      mul.init_identity();
      mul.a[4][x] = y;
      modify(1, l, r, mul);
    }
    else if (op == 2) {  // 交换第x排和第y排的[l,r]
      int x, y, l, r; read(x), read(y), read(l), read(r);
      Matrix<ll> mul(4, 4);
      mul.init_identity();
      mul.a[x][x] = mul.a[y][y] = 0;
      mul.a[x][y] = mul.a[y][x] = 1;
      modify(1, l, r, mul);
    }
    else {  // 第y排[l,r]+=第x排[l,r]
      int x, y, l, r; read(x), read(y), read(l), read(r);
      Matrix<ll> mul(4, 4);
      mul.init_identity();
      mul.a[x][y]++;
      modify(1, l, r, mul);
    }
  }
}

int main() {
  solve();
}
```

17.6 势能线段树

传统的支持区间修改的线段树通过lazy标记规避频繁且大量的单点修改,以保证查询$O(\log n)$的时间复杂度.

能用lazy标记的区间操作需满足:①区间节点的值可通过计算当前节点的lazy标记来更新;②不同的lazy标记可实现就地快速合并.如区间加、区间乘修改,当前节点的值可通过加上lazy标记乘区间长度、乘lazy来更新,而lazy标记也可通过加、乘来更新.

对区间开根号、区间与运算、区间历史最值等操作,其对每个节点的修改一定程度上取决于叶子节点当前的值,则难通过lazy标记的合并来实现对区间的快速更新.注意到区间开根号操作,每个节点至多被开$O(\log n)$次,且当某节点值变为1时,后续操作不会对该节点产生影响.注意到区间与运算操作不会增加节点值的二进制表示中1的个数,且当某节点值变为0时,后续操作不会对该节点产生影响.注意到这些操作对节点的操作次数有上限,称为"势能",超过该上限后操作退化失效,即势能减为0.当区间中所有的节点势能减为0时可直接返回,不再向下递归.

势能线段树的构建和操作:

①每个节点添加一个势能函数,记录对应区间的势能情况.

②每次做区间修改时,若当前区间内的所有节点的势能都减为0,则返回,不再向下递归;若当前区间存在节点势能非零,则向下递归,暴力修改区间内每个势能非零的节点.

可以证明每次修改的时间复杂度为$O(\log^2 n)$.

17.6.1 吉司机线段树

题意 (3 s)

给定一个长度为n的序列a和一个辅助序列b,其中b初始时与a相同.

现有m个操作,操作有如下五种类型:

①1 l r k,表示对$\forall i \in [l,r], a_i \leftarrow a_i + k$($k$可为负).

②2 l r v,表示对$\forall i \in [l,r], a_i \leftarrow \min\{a_i, v\}$.

③$3\ l\ r$,表示求$\sum_{i=l}^{r} a_i$.

④$4\ l\ r$,表示求$\max_{i\in[l,r]} a_i$.

⑤$5\ l\ r$,表示求$\max_{i\in[l,r]} b_i$.

每次操作后都更新$b_i \leftarrow \max\{b_i, a_i\}$.

第一行输入两整数n, m $(1 \le n, m \le 5e5)$,表示序列a的长度和操作次数.第二行输入n个整数$a_1, \cdots, a_n$ $(-5e8 \le a_i \le 5e8)$,表示初始序列a.接下来m行每行输入一个操作op,输入格式如上.数据保证操作合法,且$-2000 \le k \le 2000, -5e8 \le v \le 5e8$.

对$op \in \{3, 4, 5\}$的操作,输出答案.

思路

定义:

①历史最大(小)值:a_i存放的最大(小)数.

②区间最大(小)值操作:对$\forall i \in [l, r], a_i \leftarrow \max\{a_i, v\}(a_i \leftarrow \min\{a_i, v\})$.

③严格次大值:比最大值小的数的最大值,即最大值的前驱.

操作②为区间最小值操作,操作⑤求区间最大历史最大值.

瓶颈在于操作②后无法快速更新区间和.线段树不能直接区间取min,但可将区间内$> v$的数都减去一个数,使得这些数变为v.但不同的数要减去不同的数才能得到v,显然不能维护一个很复杂的懒标记来表示区间内不同的数要减去的数.注意到若区间内只有一种$> v$的数,即所有$> v$的数都相等,则只需维护一个懒标记即可.

为使得区间内$> v$的数只有一种,在线段树的节点多开三个变量:

①$maxnum$,表示区间最大值.

②cnt,表示区间最大值的个数.

③$prev$,表示区间最大值的前驱,即严格次大值.

因只有在区间内只有一种数$> v$时才能快速更新,则只能在满足$prev < v$且$maxnum > v$的节点上更新.

具体地,进行操作②的push_up操作时,有如下三种情况:

①若$maxnum \le v$,则当前区间最大值$\le v$,无需更新,直接返回.

②若$prev < v < maxnum$,则当前区间的最大值会被全部修改为v,且最大值的个数不变,修改后打上懒标记返回.

③若$v \le prev$,则无法更新,继续递归.

线段树的节点记录的信息:

①l, r,表示区间的左右端点.

②sum,表示区间和,注意开ll.

③$maxnum$,表示区间最大值.

④cnt,表示区间最大值的个数.

⑤$prev$,表示区间最大值的前驱,即严格次大值.若区间内只有一种数,则无严格次大值,令$prev = -INF$.

⑥$maxhis$,表示区间历史最大值.

⑦$lazy_maxnum$,表示区间最大值的懒标记.

⑧$lazy_others$,表示区间非最大值的懒标记.

⑨max_lazy_maxnum,表示不考虑父节信息时,区间$lazy_maxnum$的最大值.

⑩max_lazy_others,表示不考虑父节信息时,区间$lazy_others$的最大值.

push_up、build、query_sum、query_maxnum、query_maxhis、modify_add、modify_min都是线段树常规操作.

为实现push_down操作,先实现一个update函数:

```
// add_maxnum,表示区间最大值要加的数
// add_others,表示区间非最大值要加的数
// max_add_maxnum,表示区间add_maxnum的最大值
// max_add_others,表示区间add_others的最大值
void update(int u, int add_maxnum, int add_others, int max_add_maxnum, int max_add_others) {
  SegT[u].sum += (ll)add_maxnum * SegT[u].cnt + (ll)add_others * (SegT[u].r - SegT[u].l + 1 - SegT[u].cnt);

  // 更新区间历史最大值、最大懒标记
```

```
  SegT[u].maxhis = max(SegT[u].maxhis, SegT[u].maxnum + max_add_maxnum);
  SegT[u].max_lazy_maxnum = max(SegT[u].max_lazy_maxnum, SegT[u].lazy_maxnum + max_add_maxnum);
  SegT[u].max_lazy_others = max(SegT[u].max_lazy_others, SegT[u].lazy_others + max_add_others);

  // 更新区间最大值、区间次大值和懒标记
  SegT[u].maxnum += add_maxnum;
  SegT[u].lazy_maxnum += add_maxnum;
  SegT[u].lazy_others += add_others;
  if (SegT[u].prev != -INF) SegT[u].prev += add_others;
}
```

push_down操作:

```
void push_down(int u) {
  int curmax = max(SegT[u << 1].maxnum, SegT[u << 1 | 1].maxnum);

  if (SegT[u << 1].maxnum == curmax)  // 左子树
    update(u << 1, SegT[u].lazy_maxnum, SegT[u].lazy_others, SegT[u].max_lazy_maxnum, SegT[u].max_lazy_others);
  else
    update(u << 1, SegT[u].lazy_others, SegT[u].lazy_others, SegT[u].max_lazy_others, SegT[u].max_lazy_others);

  if (SegT[u << 1 | 1].maxnum == curmax)  // 右子树
    update(u << 1 | 1, SegT[u].lazy_maxnum, SegT[u].lazy_others, SegT[u].max_lazy_maxnum, SegT[u].max_lazy_others);
  else
    update(u << 1 | 1, SegT[u].lazy_others, SegT[u].lazy_others, SegT[u].max_lazy_others, SegT[u].max_lazy_others);

  SegT[u].lazy_maxnum = SegT[u].lazy_others = SegT[u].max_lazy_maxnum = SegT[u].max_lazy_others = 0;  // 清空懒标记
}
```

代码

```
const int MAXN = 5e5 + 5;
int n, m;  // 序列长度、操作数
int a[MAXN];  // 初始序列
struct Node {
  int l, r;
  ll sum;  // 区间和
  int maxnum;  // 区间最大值
  int cnt;  // 区间最大值的个数
  int prev;  // 区间最大值的前驱,即严格次大值
  int maxhis;  // 区间历史最大值
  int lazy_maxnum;  // 区间最大值的懒标记
  int lazy_others;  // 区间非最大值的懒标记
  int max_lazy_maxnum;  // 不考虑父节信息时,区间lazy_maxnum的最大值
  int max_lazy_others;  // 不考虑父节信息时,区间lazy_others的最大值
}SegT[MAXN << 2];

void push_up(int u) {
  SegT[u].sum = SegT[u << 1].sum + SegT[u << 1 | 1].sum;
  SegT[u].maxhis = max(SegT[u << 1].maxhis, SegT[u << 1 | 1].maxhis);

  // 更新区间最大值、最大值的个数、严格次大值
  SegT[u].maxnum = max(SegT[u << 1].maxnum, SegT[u << 1 | 1].maxnum);
  if (SegT[u << 1].maxnum == SegT[u << 1 | 1].maxnum) {
    SegT[u].prev = max(SegT[u << 1].prev, SegT[u << 1 | 1].prev);
    SegT[u].cnt = SegT[u << 1].cnt + SegT[u << 1 | 1].cnt;
  }
  else if (SegT[u << 1].maxnum > SegT[u << 1 | 1].maxnum) {
    SegT[u].prev = max(SegT[u << 1].prev, SegT[u << 1 | 1].maxnum);
    SegT[u].cnt = SegT[u << 1].cnt;
  }
  else {
    SegT[u].prev = max(SegT[u << 1].maxnum, SegT[u << 1 | 1].prev);
    SegT[u].cnt = SegT[u << 1 | 1].cnt;
  }
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r;
  if (l == r) {
    SegT[u].sum = SegT[u].maxnum = SegT[u].maxhis = a[l];
    SegT[u].cnt = 1;
    SegT[u].prev = -INF;  // 区间无严格次大值
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}
```

```
50
51  // add_maxnum,表示区间最大值要加的数
52  // add_others,表示区间非最大值要加的数
53  // max_add_maxnum,表示区间add_maxnum的最大值
54  // max_add_others,表示区间add_others的最大值
55  void update(int u, int add_maxnum, int add_others, int max_add_maxnum, int max_add_others) {  // 供push_down调用
56    SegT[u].sum += (ll)add_maxnum * SegT[u].cnt + (ll)add_others * (SegT[u].r - SegT[u].l + 1 - SegT[u].cnt);
57
58    // 更新区间历史最大值、最大懒标记
59    SegT[u].maxhis = max(SegT[u].maxhis, SegT[u].maxnum + max_add_maxnum);
60    SegT[u].max_lazy_maxnum = max(SegT[u].max_lazy_maxnum, SegT[u].lazy_maxnum + max_add_maxnum);
61    SegT[u].max_lazy_others = max(SegT[u].max_lazy_others, SegT[u].lazy_others + max_add_others);
62
63    // 更新区间最大值、区间次大值和懒标记
64    SegT[u].maxnum += add_maxnum;
65    SegT[u].lazy_maxnum += add_maxnum;
66    SegT[u].lazy_others += add_others;
67    if (SegT[u].prev != -INF) SegT[u].prev += add_others;
68  }
69
70  void push_down(int u) {
71    int curmax = max(SegT[u << 1].maxnum, SegT[u << 1 | 1].maxnum);
72
73    if (SegT[u << 1].maxnum == curmax)  // 左子树
74      update(u << 1, SegT[u].lazy_maxnum, SegT[u].lazy_others, SegT[u].max_lazy_maxnum, SegT[u].max_lazy_others);
75    else
76      update(u << 1, SegT[u].lazy_others, SegT[u].lazy_others, SegT[u].max_lazy_others, SegT[u].max_lazy_others);
77
78    if (SegT[u << 1 | 1].maxnum == curmax)  // 右子树
79      update(u << 1 | 1, SegT[u].lazy_maxnum, SegT[u].lazy_others, SegT[u].max_lazy_maxnum, SegT[u].max_lazy_others);
80    else
81      update(u << 1 | 1, SegT[u].lazy_others, SegT[u].lazy_others, SegT[u].max_lazy_others, SegT[u].max_lazy_others);
82
83    SegT[u].lazy_maxnum = SegT[u].lazy_others = SegT[u].max_lazy_maxnum = SegT[u].max_lazy_others = 0;  // 清空懒标记
84  }
85
86  ll query_sum(int u, int l, int r) {
87    if (SegT[u].l > r || SegT[u].r < l) return 0;
88    if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u].sum;
89
90    push_down(u);
91    int mid = SegT[u].l + SegT[u].r >> 1;
92    ll res = 0;
93    if (l <= mid) res += query_sum(u << 1, l, r);
94    if (r > mid) res += query_sum(u << 1 | 1, l, r);
95    return res;
96  }
97
98  int query_maxnum(int u, int l, int r) {
99    if (SegT[u].l > r || SegT[u].r < l) return -INF;  // 注意返回-INF
100   if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u].maxnum;
101
102   push_down(u);
103   int mid = SegT[u].l + SegT[u].r >> 1;
104   int res = -INF;
105   if (l <= mid) res = max(res, query_maxnum(u << 1, l, r));
106   if (r > mid) res = max(res, query_maxnum(u << 1 | 1, l, r));
107   return res;
108 }
109
110 int query_maxhis(int u, int l, int r) {
111   if (SegT[u].l > r || SegT[u].r < l) return -INF;  // 注意返回-INF
112   if (SegT[u].l >= l && SegT[u].r <= r) return SegT[u].maxhis;
113
114   push_down(u);
115   int mid = SegT[u].l + SegT[u].r >> 1;
116   int res = -INF;
117   if (l <= mid) res = max(res, query_maxhis(u << 1, l, r));
118   if (r > mid) res = max(res, query_maxhis(u << 1 | 1, l, r));
119   return res;
120 }
121
122 void modify_add(int u, int l, int r, int k) {
123   if (SegT[u].l > r || SegT[u].r < l) return;
124   if (SegT[u].l >= l && SegT[u].r <= r) return update(u, k, k, k, k);
125
126   push_down(u);
127   int mid = SegT[u].l + SegT[u].r >> 1;
128   if (l <= mid) modify_add(u << 1, l, r, k);
129   if (r > mid) modify_add(u << 1 | 1, l, r, k);
130   push_up(u);
131 }
```

```

void modify_min(int u, int l, int r, int v) {
  if (SegT[u].l > r || SegT[u].r < l || v >= SegT[u].maxnum) return;
  if (SegT[u].l >= l && SegT[u].r <= r && v > SegT[u].prev)  // 只在区间严格次大值<v<区间最大值时更新
    return update(u, v - SegT[u].maxnum, 0, v - SegT[u].maxnum, 0);

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify_min(u << 1, l, r, v);
  if (r > mid) modify_min(u << 1 | 1, l, r, v);
  push_up(u);
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);

  while (m--) {
    int op, l, r, k, v; cin >> op >> l >> r;
    switch (op) {
      case 1:
        cin >> k;
        modify_add(1, l, r, k);
        break;
      case 2:
        cin >> v;
        modify_min(1, l, r, v);
        break;
      case 3:
        cout << query_sum(1, l, r) << endl;
        break;
      case 4:
        cout << query_maxnum(1, l, r) << endl;
        break;
      case 5:
        cout << query_maxhis(1, l, r) << endl;
        break;
    }
  }
}
```

17.6.2 And RMQ

原题指路:https://codeforces.com/gym/103107/problem/A

题意 (3 s)

维护一个序列$\{a_n\}$,支持如下操作:

①$AND\ l\ r\ v$,表示将所有$a_i\ \ (l \le i \le r)$变为$a_i\ \&\ v\ \ (1 \le v \le 2^{30}-1)$.

②$UPD\ x\ v$,表示将$a_x\ \ (1 \le x \le n)$变为$v\ \ (1 \le v \le 2^{30}-1)$.

③$QUE\ l\ r$,表示询问区间$[l,r]\ \ (1 \le l,r \le n)$中的最大值.

第一行输入两个整数$n,m\ \ (1 \le n,m \le 4e5)$,表示序列长度和操作数.第二行输入$n$个整数$a_1,\cdots,a_n\ \ (1 \le a_i \le 2^{30}-1)$.接下来$m$行每行输入一个操作,格式如上.

思路

在每个节点处维护一个31位的二进制数作为势能函数,记录每个节点所代表的区间的元素值的二进制表示中0的位置,其中叶子节点的势能函数等于其对应元素的值,父节点的势能函数等于其两个子节点的势能函数值相或.如左右区间的势能函数分别为0110001和0011001,则它们的父节点的势能函数为0111001.

对区间与操作,若对v的二进制表示中为0的位置,当前区间的势能函数值的二进制表示对应数位也为0,则该操作对该区间无影响,直接返回;否则递归修改到左右两区间.如对上述父节点所表示的区间与$v=(0011011)_2$时,先将v按位取反得$v'=(1100100)_2$.因$(0111001)\ \&\ v' \ne 0$,递归到左右区间.对左区间,因$(0110001)_2\ \&\ v' \ne 0$,继续递归;对右区间,因$(0011001)_2\ \&\ v'=0$,直接返回.

代码

```
const int MAXN = 4e5 + 5;
int n, m;  // 序列长度、操作数
int a[MAXN];
struct Node {
  int l, r;  // 区间左右端点
  int maxnum;  // 区间最大值
  bitset<31> lazy;  // 势能函数
}SegT[MAXN << 2];

void push_up(int u) {
```

```
  SegT[u].maxnum = max(SegT[u << 1].maxnum, SegT[u << 1 | 1].maxnum);
  SegT[u].lazy = SegT[u << 1].lazy | SegT[u << 1 | 1].lazy;
}

void build(int u, int l, int r) {
  SegT[u].l = l, SegT[u].r = r;
  if (l == r) {
    SegT[u].maxnum = a[l];
    SegT[u].lazy = bitset<31>(a[l]);
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void modify_and(int u, int l, int r, int x) {  // [l,r] &= x
  bitset<31> tmpx(x);
  bitset<31> rtmpx(tmpx);
  rtmpx.flip();

  if ((rtmpx & SegT[u].lazy).none()) return;  // 本操作不会对该区间产生影响

  if (SegT[u].l == SegT[u].r) {  // 暴力修改叶子节点
    SegT[u].maxnum &= x;
    SegT[u].lazy &= tmpx;
    return;
  }

  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify_and(u << 1, l, r, x);
  if (r > mid) modify_and(u << 1 | 1, l, r, x);

  push_up(u);
}

void modify(int u, int x, int v) {  // a[x]=v
  if (SegT[u].l == SegT[u].r) {  // 叶子节点
    SegT[u].maxnum = v;
    SegT[u].lazy = bitset<31>(v);
    return;
  }

  int mid = SegT[u].l + SegT[u].r >> 1;
  if (x <= mid) modify(u << 1, x, v);
  else modify(u << 1 | 1, x, v);

  push_up(u);
}

int query(int u, int l, int r) {  // 询问[l,r]的最大值
  if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].maxnum;

  int mid = SegT[u].l + SegT[u].r >> 1;
  int res = -INF;
  if (l <= mid) res = max(res, query(u << 1, l, r));
  if (r > mid) res = max(res, query(u << 1 | 1, l, r));
  return res;
}

void solve() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);

  while (m--) {
    string op; cin >> op;
    if (op == "AND") {
      int l, r, v; cin >> l >> r >> v;
      modify_and(1, l, r, v);
    }
    else if (op == "UPD") {
      int x, v; cin >> x >> v;
      modify(1, x, v);
    }
    else {
      int l, r; cin >> l >> r;
      cout << query(1, l, r) << endl;
    }
  }
```

```
}

int main() {
  solve();
}
```

17.6.3 Lowbit

原题指路:https://codeforces.com/gym/103145/problem/D

题意 (7 s)

维护一个序列$a_1,\cdots,a_n$,支持如下两种操作:

①1 l r,表示所有$a_i+=lowbit(a_i)$ $(1\le l\le i\le r\le n)$.

②2 l r,表示询问$\sum_{i=l}^{r}a_i$,答案对998244353取模.

有t $(1\le t\le 20)$组测试数据.每组测试数据第一行输入一个整数n $(1\le n\le 1e5)$,表示序列长度.第二行输入n个整数$a_1,\cdots,a_n$ $(1\le a_i<998244353)$.第三行输入一个整数m $(1\le m\le 1e5)$,表示操作个数.接下来m行每行输入一个操作,格式如上.

思路

注意到二进制数为$100\cdots$的形式时,$+=lowbit$操作退化为$*=2$.当二进制数为$1\cdots100\cdots$的形式时,$+=lowbit$操作会使得低位的1向高位移动,若干次操作后低位的1与高位的1相邻,此时再$+=lowbit$会发生进位,变为$100\cdots$的形式.综上,$lowbit$操作在不超过$O(\log a)$次后都退化为$*=2$操作,只需在节点上开一个变量$flag$记录当前区间内是否所有数的二进制表示都是$100\cdots$的形式,若是则按区间乘操作处理,打上lazy标记后返回;否则暴力修改叶子节点,每次修改后更新$flag$.

注意$+=lowbit$操作未退化前更新区间和不能取模,否则会影响对操作是否已退化的判断.

代码

```
const int MAXN = 1e5 + 5;
const int MOD = 998244353;
int n;  // 序列长度
int a[MAXN];
struct Node {
  int l, r;  // 区间端点
  ll sum;  // 区间和
  bool flag;  // 记录当前区间内所有数的二进制表示是否都是100...的形式
  ll lazy;  // 区间乘懒标记
}SegT[MAXN << 2];

bool check(int x) { return x == lowbit(x); }  // 判断数的二进制表示是否是100...的形式

void push_up(int u) {
  SegT[u].sum = (SegT[u << 1].sum + SegT[u << 1 | 1].sum) % MOD;
  SegT[u].flag = SegT[u << 1].flag && SegT[u << 1 | 1].flag;
}

void build(int u, int l, int r) {
  SegT[u] = { l,r,0,false,1 };
  if (l == r) {
    SegT[u].sum = a[l];
    SegT[u].flag = check(a[l]);
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void push_down(int u) {
  if (SegT[u].lazy > 1) {
    SegT[u << 1].lazy = SegT[u << 1].lazy * SegT[u].lazy % MOD;
    SegT[u << 1].sum = SegT[u << 1].sum * SegT[u].lazy % MOD;
    SegT[u << 1 | 1].lazy = SegT[u << 1 | 1].lazy * SegT[u].lazy % MOD;
    SegT[u << 1 | 1].sum = SegT[u << 1 | 1].sum * SegT[u].lazy % MOD;
    SegT[u].lazy = 1;  // 注意清空为1
  }
}

void modify(int u, int l, int r) {
  if (SegT[u].l == SegT[u].r) {  // 暴力修改叶子节点
    if (SegT[u].flag) SegT[u].sum = SegT[u].sum * 2 % MOD;
    else {
      SegT[u].sum += lowbit(SegT[u].sum);  // 注意此处不能取模
```

```
      SegT[u].flag = check(SegT[u].sum);
    }
    return;
  }

  if (l <= SegT[u].l && SegT[u].r <= r) {
    if (SegT[u].flag) {  // 已退化为区间乘
      SegT[u].lazy = SegT[u].lazy * 2 % MOD;
      SegT[u].sum = SegT[u].sum * 2 % MOD;
      return;
    }
  }

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify(u << 1, l, r);
  if (r > mid) modify(u << 1 | 1, l, r);
  push_up(u);
}

int query(int u, int l, int r) {  // 询问区间和[l,r]
  if (l <= SegT[u].l && SegT[u].r <= r) return SegT[u].sum % MOD;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  int res = 0;
  if (l <= mid) res = ((ll)res + query(u << 1, l, r))  % MOD;
  if (r > mid) res = ((ll)res + query(u << 1 | 1, l, r)) % MOD;
  return res;
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);

  CaseT{
    int op,l,r; cin >> op >> l >> r;
    if (op == 1) modify(1, l, r);
    else cout << query(1, l, r) << endl;
  }
}

int main() {
  CaseT  // 单测时注释掉该行
  solve();
}
```

17.6.4 Counting Stars

原题指路:https://acm.hdu.edu.cn/showproblem.php?pid=7059

题意 (4 s)

维护一个序列$\{a_n\}$,支持如下三种操作:

①1 l r,表示询问$\sum_{i=l}^{r} a_i$,答案对998244353取模.

②2 l r,表示将所有$a_i -= lowbit(a_i)$ $(l \le i \le r)$.

③3 l r,表示将所有$a_i += highbit(a_i)$ $(1 \le i \le r)$.

有t $(1 \le t \le 5)$组测试数据.每组测试数据第一行输入一个整数n $(1 \le n \le 1e5)$,表示序列长度.第二行输入n个整数$a_1, \cdots, a_n$ $(1 \le a_i \le 1e9)$.第三行输入一个整数q $(1 \le q \le 1e5)$,表示操作个数.接下来q行每行输入一个操作,输入格式如上.

思路

注意到每个元素的二进制表示中1的个数不增,操作②会使得1的个数-1,操作③不改变1的个数,在节点处用变量$maxcnt$记录当前区间的元素的二进制表示中1的个数的最大值.修改时,若当前区间的$cnt = 0$,则修改操作不会对该区间产生影响,直接返回.

操作②当$maxcnt = 1$和$maxcnt > 1$时影响不同,故可将元素的$highbit$与其他部分$rest$分开存储,以快速更新.

代码

```
const int MAXN = 1e5 + 5;
const int MOD = 998244353;
int n;  // 序列长度
int a[MAXN];
```

```
int pow2[MAXN];  // pow2[i]表示2的i次方模MOD
struct Node {
  int l, r;  // 区间左右端点
  ll hbit, rest;  // 区间highbit、除了highbit的部分之和
  int maxcnt;  // 区间元素的二进制表示中1的个数的最大值
  int lazy;  // hbit的懒标记
}SegT[MAXN << 2];

void init() {  // 预处理出pow2
  pow2[0] = 1;
  for (int i = 1; i < MAXN; i++) pow2[i] = (ll)pow2[i - 1] * 2 % MOD;
}

int get_ones(int x) {  // 求x的二进制表示中1的个数
  int res = 0;
  while (x) x -= lowbit(x), res++;
  return res;
}

int highbit(int x) {
  for (int i = 30; i >= 0; i--)
    if (x >> i & 1) return 1 << i;
  return 0;
}

void push_up(int u) {
  SegT[u].hbit = ((ll)SegT[u << 1].hbit + SegT[u << 1 | 1].hbit) % MOD;
  SegT[u].rest = ((ll)SegT[u << 1].rest + SegT[u << 1 | 1].rest) % MOD;
  SegT[u].maxcnt = max(SegT[u << 1].maxcnt, SegT[u << 1 | 1].maxcnt);
}

void build(int u, int l, int r) {
  SegT[u] = { l,r,0,0,0,0 };  // 注意将其余变量初始化为0
  if (l == r) {
    SegT[u].hbit = highbit(a[l]);
    SegT[u].rest = a[l] - SegT[u].hbit;
    SegT[u].maxcnt = get_ones(a[l]);
    return;
  }

  int mid = l + r >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  push_up(u);
}

void push_down(int u) {
  if (SegT[u].lazy) {
    SegT[u << 1].lazy += SegT[u].lazy, SegT[u << 1 | 1].lazy += SegT[u].lazy;
    SegT[u << 1].hbit = (ll)SegT[u << 1].hbit * pow2[SegT[u].lazy] % MOD;
    SegT[u << 1 | 1].hbit = (ll)SegT[u << 1 | 1].hbit * pow2[SegT[u].lazy] % MOD;
    SegT[u].lazy = 0;
  }
}

void modify_sub(int u, int l, int r) {  // [l,r] -= lowbit
  if (!SegT[u].maxcnt) return;  // 本操作不会对该区间产生影响

  if (SegT[u].l == SegT[u].r) {  // 暴力修改叶子节点
    if (SegT[u].maxcnt > 1) {  // 不影响区间highbit
      SegT[u].rest -= lowbit(SegT[u].rest);
      SegT[u].maxcnt--;
    }
    else SegT[u].hbit = SegT[u].maxcnt = 0;
    return;
  }

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify_sub(u << 1, l, r);
  if (r > mid) modify_sub(u << 1 | 1, l, r);
  push_up(u);
}

void modify_add(int u, int l, int r) {  // [l,r] += highbit
  if (!SegT[u].maxcnt) return;  // 本操作不会对该区间产生影响

  if (l <= SegT[u].l && SegT[u].r <= r) {
    SegT[u].hbit = (ll)SegT[u].hbit * 2 % MOD;
    SegT[u].lazy++;
    return;
  }

```

```
  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  if (l <= mid) modify_add(u << 1, l, r);
  if (r > mid) modify_add(u << 1 | 1, l, r);
  push_up(u);
}

int query(int u, int l, int r) {  // 询问区间和[l,r]
  if (!SegT[u].maxcnt) return 0;

  if (l <= SegT[u].l && SegT[u].r <= r) return (SegT[u].hbit + SegT[u].rest) % MOD;

  push_down(u);
  int mid = SegT[u].l + SegT[u].r >> 1;
  int res = 0;
  if (l <= mid) res = ((ll)res + query(u << 1, l, r)) % MOD;
  if (r > mid) res = ((ll)res + query(u << 1 | 1, l, r)) % MOD;
  return res;
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  build(1, 1, n);

  CaseT{
    int op, l, r; cin >> op >> l >> r;
    if (op == 1) cout << query(1, l, r) << endl;
    else if (op == 2) modify_sub(1, l, r);
    else modify_add(1, l, r);
  }
}

int main() {
  init();
  CaseT  // 单测时注释掉该行
  solve();
}
```

17.7 权值线段树

权值线段树可视为一个动态的桶.

元素值域较大时,可先离散化再建线段树,也可动态开点.动态开点时,空间大小约$m \cdot \log A$,其中m是操作次数,A是元素值域的上界.

17.7.1 静态区间第k小

原题指路:https://www.luogu.com.cn/problem/P3834

题意 ($1 \sim 1.2$ s)

给定一个长度为n $(1 \le n \le 2e5)$的序列$a_1, \cdots, a_n$ $(|a_i| \le 1e9)$和m $(1 \le m \le 2e5)$个询问,每个询问给定三个整数l, r, k $(1 \le l \le r \le n, 1 \le k \le r - l + 1)$,表示询问区间$[l, r]$的第$k$小的元素.

代码

```
const int MAXN = 2e5 + 5;
namespace WeightedSegmentTree{
  int MAXM;  // 值域
  int n;
  int a[MAXN], backup[MAXN];  // 原序列及其备份,下标从1开始
  struct Node {
    int l, r;  // 每个节点的左右儿子节点的下标
    int sum;  // 区间的元素个数

    Node(int _sum = 0) :l(0), r(0), sum(_sum) {}
  }SegT[MAXN * 20];
  int idx = 0;  // 当前用到的节点编号
  int root[MAXN];  // 每棵线段树的根节点

  void discretize() {  // 将a[]离散化
    memcpy(backup, a, so(a));
    sort(backup + 1, backup + n + 1);
    MAXM = unique(backup + 1, backup + n + 1) - (backup + 1);
    for (int i = 1; i <= n; i++) a[i] = lower_bound(backup + 1, backup + MAXM + 1, a[i]) - backup;
  }
```

```
  // last为上一棵树的根节点,cur为当前要创建的根节点,x∈[l,r]为要插入的值(离散化后的)
  void insert(int last, int& cur, int l, int r, int x) {
    cur = ++idx;  // 新建节点
    SegT[cur] = SegT[last];  // 复制上一棵树的节点,因为它们表示的权值区间相同
    SegT[cur].sum++;  // 更新区间元素出现次数

    if (l == r) return;  // 叶子节点
    int mid = l + r >> 1;
    if (x <= mid) insert(SegT[last].l, SegT[cur].l, l, mid, x);
    else insert(SegT[last].r, SegT[cur].r, mid + 1, r, x);
  }

  void build() {  // 建权值线段树
    discretize();  // 离散化
    for (int i = 1; i <= n; i++) insert(root[i - 1], root[i], 1, MAXM, a[i]);
  }

  // root1、root2分别为(l-1)和r对应的子树的根节点
  int query(int root1, int root2, int l, int r, int k) {
    if (l == r) return l;  // 叶子节点
    int tmp = SegT[SegT[root2].l].sum - SegT[SegT[root1].l].sum;
    int mid = l + r >> 1;
    if (k <= tmp) return query(SegT[root1].l, SegT[root2].l, l, mid, k);
    else return query(SegT[root1].r, SegT[root2].r, mid + 1, r, k - tmp);
  }

  int query(int l, int r, int k) {  // 查询区间[l,r]的第k小元素
    return backup[query(root[l - 1], root[r], 1, MAXM, k)];
  }
}
using namespace WeightedSegmentTree;

void solve() {
  int q; cin >> n >> q;
  for (int i = 1; i <= n; i++) cin >> a[i];

  build();

  while (q--) {
    int l, r, k; cin >> l >> r >> k;
    cout << query(l, r, k) << endl;
  }
}

int main() {
  solve();
}
```

17.7.2 Petya and Array

原题指路:https://codeforces.com/contest/1042/problem/D

题意 (2 s)

给定一个长度为n $(1 \le n \le 2e5)$的序列$a_1, \cdots, a_n$ $(|a_i| \le 1e9)$和一个整数t $(|t| \le 2e14)$,问有多少对(l, r) $(1 \le l \le r \le n)$ $s.t.$ $a_l + \cdots + a_r < t$.

思路

先求$a[]$的前缀和$pre[]$, 问题转化为求有多少对(l, r) $(1 \le l \le r \le n)$ $s.t.$ $pre[r] - pre[l-1] < t$, 即对$\forall r \in [1, n]$, 求区间$[1, r-1]$中$\ge pre[r] - t + 1$的数的个数.

因值域可能为负, 用动态开点的权值线段树即可.

代码

```
const int MAXN = 2e5 + 5;
namespace WeightedSegmentTree {
  const ll MAXA = 2e14;
  struct Node {
    int lson, rson;  // 左、右儿子节点的编号
    int cnt;  // 当前区间内的元素个数
  }SegT[MAXN * 48];
  int root = 1;  // 根节点
  int idx = 1;  // 当前用到的节点的编号,1号节点为根节点

  void insert(int& u, ll l, ll r, ll x) {
    if (!u) {
      u = ++idx;
      SegT[u].lson = SegT[u].rson = SegT[u].cnt = 0;
```

```
    }
    SegT[u].cnt++;

    if (l == r) return;

    ll mid = l + r >> 1;
    if (x <= mid) insert(SegT[u].lson, l, mid, x);
    else insert(SegT[u].rson, mid + 1, r, x);
  }

  int query(int u, ll l, ll r, ll L, ll R) {
    if (!u) return 0;
    if (L <= l && r <= R) return SegT[u].cnt;

    ll mid = l + r >> 1;
    int res = 0;
    if (L <= mid) res += query(SegT[u].lson, l, mid, L, R);
    if (R > mid) res += query(SegT[u].rson, mid + 1, r, L, R);
    return res;
  }
}
using namespace WeightedSegmentTree;

void solve() {
  int n; ll t; cin >> n >> t;
  vector<ll> pre(n + 1);
  for (int i = 1; i <= n; i++) {
    cin >> pre[i];
    pre[i] += pre[i - 1];
  }

  ll ans = 0;
  for (int r = 1; r <= n; r++) {
    insert(root, -MAXA, MAXA, pre[r - 1]);
    ans += query(root, -MAXA, MAXA, pre[r] - t + 1, MAXA);
  }
  cout << ans << endl;
}

int main() {
  solve();
}
```

17.7. Longest k-Good Segment

原题指路: https://codeforces.com/problemset/problem/616/D

题意

给定一个正整数k, 称序列的一个区间是好的, 如果它不包含超过k种元素. 给定一个长度为n $(1 \le k \le n \le 5e5)$的序列$a = [a_1, \cdots, a_n]$ $(0 \le a_i \le 1e6, 1 \le i \le n)$. 求$a[]$的任一最长的好的区间.

思路I

因$k \ge 1$, 则必有解. 为使得好的区间最长, 应使得区间中包含的元素的种数在不超过k的前提下尽量大.

为用桶记录每种元素出现的次数, 用双指针维护每个出现了k种元素(若存在)的区间, 更新答案即可.

时间复杂度$O(n)$.

代码I

```
const int MAXA = 1e6 + 5;

void solve() {
    int n, m; cin >> n >> m;  // 序列长度、区间最大元素种数
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    int ansl = 1, ansr = 1;
    vector<int> cnt(MAXA);
    for (int l = 1, r = 1, cur = 0; r <= n; r++) {  // cur表示当前区间中的元素种数
        if (++cnt[a[r]] == 1) cur++;
        while (cur > m) {
            if (--cnt[a[l]] == 0) cur--;
            l++;
        }

        if (r - l + 1 > ansr - ansl + 1) ansl = l, ansr = r;
    }
```

```
    cout << ansl << ' ' << ansr << endl;
}

int main() {
    solve();
}
```

思路II

线段树维护数组$cnt = [cnt_1, \cdots, cnt_n, cnt_{n+1}]$, 其中$cnt_i = 1$表示下标$i$是序列中某元素第一次出现的位置.

枚举好的区间的左端点$l \in [1, n]$, 对每个l, 用线段树求 $s.t.$ 前缀$a[1...r']$中元素个数$> k$的最靠左的下标r', 则能与l匹配的最靠右的好的区间的右端点$r = r' - 1$.

对每个好的区间的左端点l, 显然后缀$a[l...n]$中只有每个元素第一次出现的位置对区间的元素种数有贡献. $fir[i]$表示值i第一次出现的下标, 若值i未出现, 则定义$fir[i] = n + 1$. $nxt[i]$表示值$a[i]$下一次出现的下标, 若值$a[i]$无下一次出现, 则定义$nxt[i] = n + 1$. 这两个数组可从后往前递推求得, 即先将$fir[]$都初始化为$(n + 1)$, 再枚举下标i从n到1, 令$nxt[i] = fir[a[i]]$, $fir[a[i]] = i$.

初始时, 将序列的所有元素的第一次出现的下标的$cnt[]$值在线段树中$+1$.

区间左端点l右移时, 先令$cnt[l] = 0$, 即若下标l原本是某元素第一次出现的位置, 则将其$cnt[]$值清空; 再令$cnt[nxt[l]] = 1$, 即若元素$a[l]$有下一次出现的位置, 则将其$cnt[]$值置为1.

代码II

```
const int MAXA = 1e6 + 5;

struct SegmentTree {
    int n;
    struct Node {
        int sum;

        Node(int _sum = 0) :sum(_sum) {}

        friend Node operator+(const Node A, const Node B) {
            Node res;
            res.sum = A.sum + B.sum;
            return res;
        }
    };
    vector<Node> SegT;

    SegmentTree(int _n) :n(_n) {
        n++;  // 注意序列长度 + 1以区分无解的情况
        SegT.resize(n + 5 << 2);
    }

    void pushUp(int u) {
        SegT[u] = SegT[u << 1] + SegT[u << 1 | 1];
    }

    void modify(int u, int l, int r, int pos, int val) {  // cnt[pos] += val
        if (l == r) {
            SegT[u].sum += val;
            return;
        }

        int mid = l + r >> 1;
        if (pos <= mid) modify(u << 1, l, mid, pos, val);
        else modify(u << 1 | 1, mid + 1, r, pos, val);
        pushUp(u);
    }

    void modify(int pos, int val) {
        modify(1, 1, n, pos, val);
    }

    int find(int u, int l, int r, int k) {  // 求 s.t. 前缀a[1...r']中元素种数 > k的最靠左的下标r'
        if (l == r) return l;

        int mid = l + r >> 1;
        if (SegT[u << 1].sum > k) return find(u << 1, l, mid, k);
        else return find(u << 1 | 1, mid + 1, r, k - SegT[u << 1].sum);
    }

    int find(int k) {
        return find(1, 1, n, k);
    }
};

void solve() {
    int n, m; cin >> n >> m;  // 序列长度、区间最大元素种数
```

```
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    vector<int> fir(MAXA, n + 1);  // fir[i]表示值i第一次出现的下标
    vector<int> nxt(n + 1);  // nxt[i]表示值a[i]下一次出现的下标
    for (int i = n; i; i--) {
        nxt[i] = fir[a[i]];
        fir[a[i]] = i;
    }

    SegmentTree st(n);
    for (int i = 0; i < MAXA; i++)  // 注意值域从0开始枚举
        if (fir[i] <= n) st.modify(fir[i], 1);  // cnt[fir[i]]++

    int ansl = 1, ansr = 1;
    for (int l = 1; l <= n; l++) {
        int r = st.find(m) - 1;  // - 1后为当前元素种数 <= k的最靠右的下标
        if (r - l + 1 > ansr - ansl + 1)
            ansl = l, ansr = r;
        if (r == n) break;

        st.modify(l, -1);  // cnt[l]--
        if (nxt[l] <= n) st.modify(nxt[l], 1);  // cnt[nxt[l]]++
    }
    cout << ansl << ' ' << ansr << endl;
}

int main() {
    solve();
}
```

17.8 二维树状数组

二维树状数组可视为树状数组套树状数组,它对一个二维矩阵的行、列方向分别按lowbit进行划分,树状数组的每个节点维护一个树状数组.

二维树状数组相比起二维线段树更省空间.

17.8.1 二维树状数组I

原题指路:https://loj.ac/p/133

题意

维护一个初始时为零矩阵的$n \times m$ $(1 \leq n, m \leq 2^{12})$矩阵$A = (a_{i,j})_{n \times m}$,支持如下两种操作:

①单点修改:1 x y k,表示令$a_{x,y} += k$ $(1 \leq x \leq n, 1 \leq y \leq m, |k| \leq 1e5)$.

②区间查询:2 a b c d,表示询问以(a, b) $(1 \leq a \leq n, 1 \leq b \leq y)$为左上角、以$(c, d)$ $(1 \leq c \leq n, 1 \leq d \leq m)$为右下角的子矩阵的元素和.

数据保证操作个数不超过$3e5$.

思路

将矩阵的行、列分别按lowbit进行划分后,因二维BIT的每个节点都维护一个BIT,则对原矩阵的单点修改,在第一维会影响一个区间,在第二维也会影响一个区间,故对原矩阵的单点修改会影响二维BIT的一个矩形,且一次单点修改的时间复杂度为$O(\log n \log m)$.

显然二维BIT支持求以$(1, 1)$为左上角、以(x, y)为右下角的矩形的元素之和,单次查询的时间复杂度为$O(\log n \log m)$.

对区间查询操作,求二维前缀和即可.

代码

```
const int MAXN = (1 << 12) + 5;
namespace BIT2D {
  int lowbit(int x) {
    return x & -x;
  }

  int n, m;
  ll BIT[MAXN][MAXN];  // 二维BIT维护原矩阵

  void modify(int x, int y, int val) {  // 单点修改:a[x][y]+=val
    for (int i = x; i <= n; i += lowbit(i)) {
      for (int j = y; j <= m; j += lowbit(j))
        BIT[i][j] += val;
    }
  }

  ll getSum(int x, int y) {  // 求以(1,1)为左上角、以(x,y)为右下角的矩形的元素之和
```

```
    ll res = 0;
    for (int i = x; i; i -= lowbit(i)) {
      for (int j = y; j; j -= lowbit(j))
        res += BIT[i][j];
    }
    return res;
  }

  // 求以(x1,y1)为左上角、以(x2,y2)为右下角的矩形的元素之和
  ll query(int x1, int y1, int x2, int y2) {
    return getSum(x2, y2) - getSum(x1 - 1, y2) - getSum(x2, y1 - 1) + getSum(x1 - 1, y1 - 1);
  }
}
using namespace BIT2D;

void solve() {
  cin >> n >> m;

  int op;
  while (cin >> op) {
    if (op == 1) {
      int x, y, k; cin >> x >> y >> k;
      modify(x, y, k);
    }
    else {
      int a, b, c, d; cin >> a >> b >> c >> d;
      cout << query(a, b, c, d) << endl;
    }
  }
}

int main() {
  solve();
}
```

17.8.2 二维树状数组II

原题指路:https://loj.ac/p/134

题意

维护一个初始时为零矩阵的$n \times m$ $(1 \leq n, m \leq 2^{12})$矩阵$A = (a_{i,j})_{n \times m}$,支持如下两种操作:

①区间修改:$1\ a\ b\ c\ d\ k$,表示令以(a, b)为左上角、以(c, d)为右下角的矩阵中的元素$+= k$ $(|k| \leq 1e5)$.

②单点查询:$2\ x\ y$,表示询问元素$a_{x,y}$的值.

数据保证操作个数不超过$3e5$.

思路

类似于一维树状数组实现区间修改和单点查询的思路,用二维BIT维护二维差分数组,则单点查询转化为求矩阵的元素和:$a_{x,y} = \sum_{i=1}^{x}\sum_{j=1}^{y} diff_{i,j}$,区间修改转化为单点修改差分数组.

代码

```
const int MAXN = (1 << 12) + 5;
namespace BIT2D {
  int lowbit(int x) {
    return x & -x;
  }

  int n, m;
  ll BIT[MAXN][MAXN];  // 二维BIT维护差分矩阵

  void add(int x, int y, int val) {  // diff[x][y]+=val
    for (int i = x; i <= n; i += lowbit(i)) {
      for (int j = y; j <= m; j += lowbit(j))
        BIT[i][j] += val;
    }
  }

  // 区间修改:令diff[][]中以(x1,y1)为左上角、以(x2,y2)为右下角的矩形的元素+=val
  void modify(int x1, int y1, int x2, int y2, int val) {
    add(x1, y1, val), add(x1, y2 + 1, -val), add(x2 + 1, y1, -val), add(x2 + 1, y2 + 1, val);
  }

  ll getSum(int x, int y) {  // 求diff[][]中以(1,1)为左上角、以(x,y)为右下角的矩形的元素之和
    ll res = 0;
```

```
    for (int i = x; i; i -= lowbit(i)) {
      for (int j = y; j; j -= lowbit(j))
        res += BIT[i][j];
    }
    return res;
  }

  ll query(int x, int y) {  // 单点查询:求a[x][y]
    return getSum(x, y);
  }
}
using namespace BIT2D;

void solve() {
  cin >> n >> m;

  int op;
  while (cin >> op) {
    if (op == 1) {
      int x1, y1, x2, y2, val; cin >> x1 >> y1 >> x2 >> y2 >> val;
      modify(x1, y1, x2, y2, val);
    }
    else {
      int x, y; cin >> x >> y;
      cout << query(x, y) << endl;
    }
  }
}

int main() {
  solve();
}
```

17.8.3 二维树状数组III

原题指路:https://loj.ac/p/135

题意

维护一个初始时为零矩阵的$n \times m$ $(1 \le n, m \le 2048)$矩阵$A = (a_{i,j})_{n \times m}$,支持如下两种操作:

①区间修改:1 a b c d k,表示令以(a, b)为左上角、以(c, d)为右下角的矩阵中的元素$+= k$ $(|k| \le 500)$.

②区间查询:2 a b c d,表示询问以(a, b) $(1 \le a \le n, 1 \le b \le y)$为左上角、以$(c, d)$ $(1 \le c \le n, 1 \le d \le m)$为右下角的子矩阵的元素和.

数据保证操作个数不超过2e5,且运算过程和最终结果都在long long范围内.

思路

类似于一维BIT实现区间修改和区间查询的思路,二维BIT维护原矩阵的差分矩阵.

设原矩阵为$a[][]$,二维前缀和数组$pre[][]$,二维差分数组为$diff[][]$.

注意到$pre_{x,y} = \sum_{i=1}^{x}\sum_{j=1}^{y} a_{i,j} = \sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{k=1}^{i}\sum_{l=1}^{j} diff_{k,l}$,考察每个$diff_{i,j}$对$pre[][]$的贡献.类似于二维前缀和,$diff_{i,j}$对前$(i-1)$行和前$(j-1)$列的$pre[][]$无贡献,对以$(i, j)$为左上角、以$(n, m)$为右下角的矩形贡献了$(n-i+1)(m-j+1)diff_{i,j}$,则$pre_{x,y} = \sum_{i=1}^{x}\sum_{j=1}^{y}(x-i+1)(y-j+1)diff_{i,j}$.因上式中的$x, y$随询问的改变而改变,故无法直接维护.考虑将$pre_{x,y}$的表达式变形,将$x, y$分离出来:

$$pre_{x,y} = \sum_{i=1}^{x}\sum_{j=1}^{y}[xy - (i-1)y - x(j-1) + (i-1)(j-1)]diff_{i,j}$$

$$= xy\sum_{i=1}^{x}\sum_{j=1}^{y}diff_{i,j} - y\sum_{i=1}^{x}\sum_{j=1}^{y}(i-1)diff_{i,j} - x\sum_{i=1}^{x}\sum_{j=1}^{y}(j-1)diff_{i,j} + \sum_{i=1}^{x}\sum_{j=1}^{y}(i-1)(j-1)diff_{i,j}.$$

用四棵二维BIT分别维护$diff_{i,j}, (i-1)diff_{i,j}, (j-1)diff_{i,j}, (i-1)(j-1)diff_{i,j}$即可.

代码

```
const int MAXN = (1 << 12) + 5;
namespace BIT2D {
  int lowbit(int x) {
    return x & -x;
  }

  int n, m;
  ll BIT1[MAXN][MAXN];  // 维护差分矩阵diff[][]
  ll BIT2[MAXN][MAXN];  // 维护(i-1)*diff[i][j]
  ll BIT3[MAXN][MAXN];  // 维护(j-1)*diff[i][j]
```

```
  ll BIT4[MAXN][MAXN];  // 维护(i-1)*(j-1)*diff[i][j]

  void add(int x, int y, int val) {  // diff[x][y]+=val
    for (int i = x; i <= n; i += lowbit(i)) {
      for (int j = y; j <= m; j += lowbit(j)) {
        BIT1[i][j] += val;
        BIT2[i][j] += ((ll)x - 1) * val;
        BIT3[i][j] += ((ll)y - 1) * val;
        BIT4[i][j] += ((ll)x - 1) * (y - 1) * val;
      }
    }
  }

  // 区间修改:令diff[][]中以(x1,y1)为左上角、以(x2,y2)为右下角的矩形的元素+=val
  void modify(int x1, int y1, int x2, int y2, int val) {
    add(x1, y1, val), add(x1, y2 + 1, -val), add(x2 + 1, y1, -val), add(x2 + 1, y2 + 1, val);
  }

  ll getSum(int x, int y) {  // 求diff[][]中以(1,1)为左上角、以(x,y)为右下角的矩形的元素之和
    ll res = 0;
    for (int i = x; i; i -= lowbit(i)) {
      for (int j = y; j; j -= lowbit(j)) {
        res += (ll)x * y * BIT1[i][j] - (ll)y * BIT2[i][j]
          - (ll)x * BIT3[i][j] + BIT4[i][j];
      }
    }
    return res;
  }

  // 区间查询:求a[][]中以(x1,y1)为左上角、以(x2,y2)为右下角的矩形的元素之和
  ll query(int x1, int y1, int x2, int y2) {
    return getSum(x2, y2) - getSum(x1 - 1, y2) - getSum(x2, y1 - 1) + getSum(x1 - 1, y1 - 1);
  }
}
using namespace BIT2D;

void solve() {
  cin >> n >> m;

  int op;
  while (cin >> op) {
    if (op == 1) {
      int x1, y1, x2, y2, val; cin >> x1 >> y1 >> x2 >> y2 >> val;
      modify(x1, y1, x2, y2, val);
    }
    else {
      int x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
      cout << query(x1, y1, x2, y2) << endl;
    }
  }
}

int main() {
  solve();
}
```

17.8.4 Kera's line segment

原题指路:https://ac.nowcoder.com/acm/contest/17797/K

题意 (2 s, 512 MB)

数轴上有n条线段,每条线段的左、右端点分别用两个整数l,r表示,每条线段有一个权值val.现有m个操作,分为如下两种:

①$1\ l\ r\ val$,表示添加一条左、右端点分别为l,r、权值为val的线段.

②$2\ L\ R$,表示询问完全包含于区间$[L,R]$的线段的权值的极差.

第一行输入两个整数n,m $(1\le n,m\le 1e5)$.接下来n行每行输入三个整数l,r,val $(1\le l\le r\le 3000, 0\le val\le 1e9)$.接下来$m$行每行输入一个操作:

(1)对操作①,输入四个整数$1\ a\ b\ val$,则要添加的线段的左、右端点分别为$l=a \text{ xor } LastAns, r=b \text{ xor } LastAns$,其中$LastAns$表示上一个询问的答案,初始时为$0$.

(2)对操作②,输入三个整数$2\ A\ B$,则询问区间的左、右端点分别为$L=A \text{ xor } LastAns, R=B \text{ xor } LastAns$,输出该询问的答案后更新$LastAns$.数据保证询问区间至少完全包含一条线段.

思路

$maxval[l][r]$、$minval[l][r]$分别表示完全包含于区间$[l,r]$的线段的权值的最大、最小值,则操作②转化为查询二维矩阵中以(l,l)为左上角、以(r,r)为右下角的矩形中的最值.

显然可用二维线段树实现单点修改和区间最值查询,但所需的空间为$\dfrac{3000\times3000\times4\times16}{1024\times1024}\approx550\text{ MB}>512\text{ MB}$.

考虑用二维BIT维护区间最值.设完全包含于区间$[L,R]$的线段的左、右端点分别为l,r,注意到BIT维护的信息需具有前缀的形式,而包含关系中$r\le R$是前缀的形式,但$l\ge L$不是前缀的形式.注意到$l\ge L\Leftrightarrow3000-l+1\le3000-L+1$,这是前缀的形式,故可用二维BIT维护,支持单点修改区间查询即可.

代码

```cpp
const int MAXN = 3000;
namespace BIT2D {
  int lowbit(int x) {
    return x & -x;
  }

  // maxval[l][r]、minval[l][r]分别表示完全包含于区间[l,r]的线段的权值的最大、最小值
  int maxval[MAXN + 5][MAXN + 5], minval[MAXN + 5][MAXN + 5];

  void modify(int l, int r, int val) {  // 添加一条左、右端点分别为l,r的、权值为val的线段
    l = MAXN - l + 1;  // 将询问转化为前缀的形式

    for (int i = l; i <= MAXN; i += lowbit(i)) {
      for (int j = r; j <= MAXN; j += lowbit(j)) {
        maxval[i][j] = max(maxval[i][j], val);
        minval[i][j] = min(minval[i][j], val);
      }
    }
  }

  int query(int l, int r) {  // 查询完全包含于区间[l,r]的线段的权值的极差
    l = MAXN - l + 1;  // 将询问转化为前缀的形式

    int maxnum = 0, minnum = INF;
    for (int i = l; i; i -= lowbit(i)) {
      for (int j = r; j; j -= lowbit(j)) {
        maxnum = max(maxnum, maxval[i][j]);
        minnum = min(minnum, minval[i][j]);
      }
    }
    return maxnum - minnum;
  }
}
using namespace BIT2D;

void solve() {
  memset(minval, INF, sizeof(minval));

  int n, m; cin >> n >> m;
  while (n--) {
    int l, r, val; cin >> l >> r >> val;
    modify(l, r, val);
  }

  int ans = 0;
  while (m--) {
    int op, l, r; cin >> op >> l >> r;
    l ^= ans, r ^= ans;
    if (l > r) swap(l, r);

    if (op == 1) {
      int val; cin >> val;
      modify(l, r, val);
    }
    else cout << (ans = query(l, r)) << endl;
  }
}

int main() {
  solve();
}
```

17.9 Kruskal重构树

[**最小瓶颈路**] 无向图中节点u到节点v的最小瓶颈路是一条简单路径, 使得路径上的最大边权不超过u到v的所有简单路径上的边权.

由MST的定理: 节点u到节点v的最小瓶颈路上的最大边权等于MST中u到v的简单路径上的最大边权. 虽MST可能不唯一, 但每个MST中u到v的简单路径的最大边权都相等, 即每个MST中u到v的简单路径都是最小瓶颈路. 但并非所有最小瓶颈路都存在一棵MST $s.t.$ 其为MST中的简单路径.

因最小瓶颈路不唯一, 一般会询问最小瓶颈路上的最大边权, 即求MST的链上的最大边权, 这可用树上倍增、树剖等方法求解, 但针对最小瓶颈路问题, 可用Kruskal重构树求解.

[**Kruskal重构树**] 同Kruskal算法求MST, 按边权从小到大加入边. 初始时新建n个集合, 每个集合恰有1个节点, 点权为0. Kruskal算法每次加边会合并两集合, 此时新建一个节点, 点权为加入的边的边权, 同时讲两集合的根节点作为新节点的左、右儿子节点. 若图存在MST, 则加入$(n-1)$条边后得到一棵有n个叶子的二叉树, 且每个非叶子节点都恰有两个儿子节点, 称该树为Kruskal重构树.

[**定理**] 原图中节点u与节点v间的所有简单路径上的最大边权的最小值 $=$ MST上u与v间的简单路径上的最大边权 $=$ 该图的Kruskal重构树上u与v的LCA的权值.

17.9.1 最小瓶颈路 (加强版)

原题指路: https://loj.ac/p/137

题意

给定一张包含编号$1 \sim n$的n $(1 \le n \le 7e4)$个节点和m $(1 \le m \le 1e5)$条边的无向连通图, 每条边有一个正权值w $(1 \le w \le 1e9+7)$, 图中无自环, 但可能有重边. 现有q $(1 \le q \le 1e7)$个询问, 每个询问给定相异的两节点u $(1 \le u \le n)$和v $(1 \le v \le n, u \ne v)$, 求从节点u到节点v的所有简单路径中, 路径上的最大边权的最小值. 规定$u = v$时答案为0.

为压缩输入, 读入四个整数A, B, C, P $(0 \le A < P, 0 \le C < P, P(B+1) < 2^{31}-1)$, 每次询问用如下函数生成$u$和$v$:

```
int A, B, C, P;

int rnd() {
    return A = (A * B + C) % P;
}

int u = rnd() % n + 1, v = rnd() % n + 1;
```

为压缩输出, 只需输出所有询问的答案之和模$(1e9+7)$的值.

思路

因本题的询问数过多, 下面的代码中有较多的STL, 开启Ofast优化也无法通过.

代码

```
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct KruskalTree {
    int n, m;
    vector<tuple<int, int, int, int>> edges;  // w, u, v, id
    vector<int> fa;
    vector<int> used;  // 记录边是否在生成树中
    int idx;  // 当前用到的节点编号
    vector<vector<int>> tr;  // Kruskal重构树
    vector<int> weights;  // Kruskal重构树中节点的点权

    KruskalTree(int _n, int _m, const vector<tuple<int, int, int, int>>& _edges)
        :n(_n), m(_m), edges(_edges), idx(n) {
        fa.resize(2 * n + 1), used.resize(m + 1);
        tr.resize(2 * n + 1), weights.resize(2 * n + 1);
        iota(all(fa), 0);
        sort(all(edges));
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto [w, u, v, id] : edges) {
            int tmpu = find(u), tmpv = find(v);
            if (tmpu != tmpv) {
                // 新建节点并连边
                fa[tmpu] = fa[tmpv] = ++idx;
                tr[tmpu].push_back(idx), tr[idx].push_back(tmpu);
                tr[tmpv].push_back(idx), tr[idx].push_back(tmpv);
                weights[idx] = w;

                used[id] = true;
                res += w, cnt++;
            }
```

```
        }
        return cnt == n - 1 ? res : INFF;
    }
};

struct RMQLCA {
    int MAXJ;
    int n;
    vector<vector<int>> edges;
    vector<int> depth;
    vector<vector<int>> fa;

    RMQLCA() {}
    RMQLCA(int _n, const vector<vector<int>>& _edges)
        :n(_n), edges(_edges), MAXJ(log2(_n) + 1) {
        depth = vector<int>(n + 1, INF);
        fa = vector<vector<int>>(n + 1, vector<int>(MAXJ + 1));
    }

    void bfs(int root) {  // 预处理depth[], fa[][]: 根节点为root
        depth[0] = 0, depth[root] = 1;

        queue<int> que;
        que.push(root);

        while (que.size()) {
            auto u = que.front(); que.pop();
            for (auto v : edges[u]) {
                if (depth[v] > depth[u] + 1) {
                    depth[v] = depth[u] + 1;
                    que.push(v);

                    fa[v][0] = u;
                    for (int k = 1; k <= MAXJ; k++)
                        fa[v][k] = fa[fa[v][k - 1]][k - 1];
                }
            }
        }
    }

    int lca(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);  // 保证节点u深度大

        // 将节点u与节点v跳到同一深度
        for (int k = MAXJ; k >= 0; k--)
            if (depth[fa[u][k]] >= depth[v]) u = fa[u][k];

        if (u == v) return u;  // u与v原本在一条链上

        // u与v同时跳到LCA的下一层
        for (int k = MAXJ; k >= 0; k--) {
            if (fa[u][k] != fa[v][k])
                u = fa[u][k], v = fa[v][k];
        }
        return fa[u][0];  // 节点u向上跳1步到LCA
    }
};

const int MOD = 1e9 + 7;

int A, B, C, P;

int rnd() {
    return A = (A * B + C) % P;
}

void solve() {
    int n, m; cin >> n >> m;
    vector<tuple<int, int, int, int>> edges(m);  // w, u, v, id
    for (int i = 0; i < m; i++) {
        auto& [w, u, v, id] = edges[i];
        cin >> u >> v >> w;
        id = i + 1;
    }

    KruskalTree kr(n, m, edges);
    kr.kruskal();
    RMQLCA sol(kr.idx, kr.tr);
    sol.bfs(kr.idx);

    int q; cin >> q >> A >> B >> C >> P;

```

```
    int ans = 0;
    while (q--) {
        int u = rnd() % n + 1, v = rnd() % n + 1;
        ans = (ans + (u != v ? kr.weights[sol.lca(u, v)] : 0)) % MOD;
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

17.9.2 Minimum spanning tree for each edge

原题指路: https://codeforces.com/contest/609/problem/E

题意 (2 s)

给定一张包含编号$1 \sim n$的n $(1 \leq n \leq 2e5)$和m $(n-1 \leq m \leq 2e5)$的简单无向图. 对每条边, 求该图包含该边的MST的权值.

思路I

先求出该图的MST, 则对含于MST中的边, 其答案为MST的权值sum. 下面考察如何求不含于MST中的边的答案.

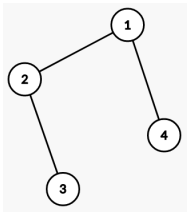
[**引理**] [**Tarjan定理**] 对一张图, 一个生成树的权值是最小的, 当且仅当不含于该生成树的边的权值 $\geq$ 含于该生成树中的边的最大权值.

由**引理**: 对不含于MST中的边(u, v), 设其权值为w, 则其答案为$sum - maxLength + w$, 其中$maxLength$为MST中节点u与节点v间的简单路径的最大边权.

考察如何求$maxLength$. 用Kruskal算法求原图的MST后, 取定MST的根节点, 从根节点开始DFS将边权下放到点权(实现时在树剖的dfs1()中完成), 对MST树剖, 用ST表维护点权最大值.

查询节点u到节点v的简单路径的最大边权res的过程类似于树剖求LCA, 即每次将所在重链的顶点的深度较大者(不妨设为$top[u]$)跳到重链顶点的父亲节点$father[top[u]]$, 同时res与ST表中的区间$[dfn[top[u]], dfn[u]]$的权值取max. 重复上述过程直至u与v在同一条重链上, 不妨设此时u为LCA, 则res与ST表中的区间$[dfn[u]+1, dfn[v]]$取max.

事实上, 存在$dfn[u]+1 > dfn[v]$的情况, 如下图中询问节点$u=1$到节点$v=4$间的最大边权时, 节点$1, 2, 3$在以节点1为顶点的重链上, 节点$v=4$在以其自身为顶点的重链上, 此时$depth[top[v]] > depth[top[u]]$, v跳至$father[top[v]]=1$, 同时res与ST表中的区间$[dfn[top[v]], dfn[v]] = [4, 4]$取max. 此时节点$u=1$与节点$v=4$在同一条重链上, 且$u$为LCA, 则$res$与ST表中的区间$[dfn[u]+1, dfn[v]] = [1+1, 1] = [2, 1]$取max, 此时ST表的询问中出现$\log_2 0$, 进而RE. 故需加判断, 当且仅当$depth[u] < depth[v]$时才需$res$与ST表中的区间$[dfn[u]+1, dfn[v]]$取max.



代码I

```
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct Kruskal {
    int n, m;
    vector<tuple<int, int, int, int>> edges;  // w, u, v, id
    vector<vector<pair<int, int>>> mst;
    vector<int> fa;
    vector<int> used;  // 记录边是否在生成树中

    Kruskal(int _n, int _m, const vector<tuple<int, int, int, int>>& _edges)
        :n(_n), m(_m), edges(_edges) {
        fa.resize(n + 1), used.resize(m + 1), mst.resize(n + 1);
        iota(all(fa), 0);
        sort(all(edges));
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto [w, u, v, id] : edges) {
            int tmpu = find(u), tmpv = find(v);
            if (tmpu != tmpv) {
```

```
                fa[tmpu] = tmpv, used[id] = true;
                mst[u].push_back({ v, w }), mst[v].push_back({ u, w });
                res += w, cnt++;
            }
        }
        return cnt == n - 1 ? res : INFF;
    }
};

struct HeavyPathDecomposition {
    int n;
    vector<vector<pair<int, int>>> edges;
    vector<int> siz;  // siz[u]表示以u为根节点的子树的大小
    vector<int> father;  // 节点的父亲节点
    vector<int> depth;  // 节点的深度
    vector<int> son;  // son[u]表示以u为根节点的子树的重儿子
    int tim;  // 时间戳
    vector<int> dfn, idfn;  // 节点的DFS序、DFS序对应的节点
    vector<int> top;  // top[u]表示节点u所在重链的顶点
    vector<int> w;  // w[v]表示节点u与其儿子节点v间的边的边权

    HeavyPathDecomposition() {}
    HeavyPathDecomposition(int _n, const vector<vector<pair<int, int>>>& _edges)
        :n(_n), edges(_edges), tim(0) {
        siz.resize(n + 1), father.resize(n + 1), depth.resize(n + 1),
        son.resize(n + 1), dfn.resize(n + 1), idfn.resize(n + 1), top.resize(n + 1);
        w.resize(n + 1);
        int root = 1;
        dfs1(root, 0);  // 从根节点开始搜，根节点无前驱节点
        dfs2(root, root);  // 从根节点开始搜，根节点是自身所在重链的顶点
    }

    void dfs1(int u, int pre) {  // 求重儿子：当前节点、前驱节点
        siz[u] = 1, father[u] = pre, depth[u] = depth[pre] + 1;
        for (auto [v, dis] : edges[u]) {
            if (v != pre) {
                w[v] = dis;
                dfs1(v, u);

                siz[u] += siz[v];
                if (siz[son[u]] < siz[v]) son[u] = v;
            }
        }
    }

    void dfs2(int u, int tp) {  // 求DFS序、重链顶点：当前节点、所在重链的顶点
        dfn[u] = ++tim, idfn[tim] = u, top[u] = tp;
        if (!son[u]) return;  // 叶子节点

        dfs2(son[u], tp);  // 先搜重儿子，重儿子与节点u在同一重链上
        for (auto [v, _] : edges[u]) {  // 再搜轻儿子
            if (v != father[u] && v != son[u])  // 不是往回搜且不是重儿子
                dfs2(v, v);  // 轻儿子在以自身为顶点的重链上
        }
    }

    int lca(int u, int v) {
        while (top[u] != top[v]) {
            if (depth[top[u]] > depth[top[v]]) u = father[top[u]];
            else v = father[top[v]];
        }
        return depth[u] < depth[v] ? u : v;
    }
};

template<class T>
struct RMQ1D {
    int n;
    vector<vector<T>> st;  // ST表
    function<T(T, T)> cmp;  // 比较规则

    RMQ1D() {}
    RMQ1D(int _n, const vector<T>& a, function<T(T, T)> _cmp) :n(_n), cmp(_cmp) {
        const int LG = log2(n);
        st = vector<vector<T>>(LG + 1, vector<T>(n + 1));

        st[0] = a;
        for (int j = 1; j <= LG; j++) {
            for (int i = 1; i <= n - (1 << j) + 1; i++)
                st[j][i] = cmp(st[j - 1][i], st[j - 1][i + (1 << j - 1)]);
        }
    }
```

```
110
111     T query(int l, int r) {  // 求区间[l, r]中满足条件cmp的最大值
112         int k = log2(r - l + 1);
113         return cmp(st[k][l], st[k][r - (1 << k) + 1]);
114     }
115 };
116
117 struct Solver {
118     int n;
119     vector<vector<pair<int, int>>> edges;
120     HeavyPathDecomposition sol;
121     vector<int> father, depth, dfn, top;
122     RMQ1D<int> st;
123
124     Solver(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {
125         sol = HeavyPathDecomposition(n, edges);
126         father = sol.father, depth = sol.depth,
127         dfn = sol.dfn, top = sol.top;
128
129         auto idfn = sol.idfn, w = sol.w;
130         vector<int> a(n + 1);
131         for (int tim = 1; tim <= n; tim++) a[tim] = w[idfn[tim]];
132
133         st = RMQ1D<int>(n, a, [&](int a, int b) {
134             return max(a, b);
135         });
136     }
137
138     int query(int u, int v) {  // 求节点u到节点v的路径的最大边权
139         int res = 0;
140         while (top[u] != top[v]) {
141             if (depth[top[u]] < depth[top[v]]) swap(u, v);  // 保证u深度大
142             res = max(res, st.query(dfn[top[u]], dfn[u]));
143             u = father[top[u]];
144         }
145         if (depth[u] > depth[v]) swap(u, v);  // u = LCA(u, v)
146         if (depth[u] < depth[v])  // 相同深度则两节点都在LCA处, 无需取max
147             res = max(res, st.query(dfn[u] + 1, dfn[v]));  // 注意取max不包含LCA的权值
148         return res;
149     }
150 };
151
152 void solve() {
153     int n, m; cin >> n >> m;
154     vector<tuple<int, int, int, int>> eds;  // w, u, v, id
155     for (int i = 1; i <= m; i++) {
156         int u, v, w; cin >> u >> v >> w;
157         eds.push_back({ w, u, v, i });
158     }
159
160     Kruskal kr(n, m, eds);
161     ll sum = kr.kruskal();
162     Solver solver(n, kr.mst);
163
164     for (auto [w, u, v, id] : eds) {
165         if (kr.used[id])  // 该边包含于该图的MST中
166             cout << sum << endl;
167         else {
168             auto maxLength = solver.query(u, v);
169             cout << sum - maxLength + w << endl;
170         }
171     }
172 }
173
174 int main() {
175     solve();
176 }
```

思路II

同**思路I**, 但采用另一种方式统计答案. 产生ST表统计答案时需加入判断的问题的原因是求节点u与节点v的简单路径上的最大边权res时不能将res与$LCA(u,v)$的点权取max, 而ST表不支持修改, 故只能跳过LCA, 分段取max. 将ST表换为线段树, 可在询问时将$LCA(u,v)$的点权修改为$-INF$, 求得答案后再修改回去, 即可避免分段取max的问题.

代码II中用结构体SegmentTree替换了**代码I**中的结构体RMQ1D, 并修改了**代码I**中的结构体Solver, 其余不变.

代码II

```
struct SegmentTree {
    int n;
    vector<int> a;
    struct Node {
        int maxv;  // 区间最大值

        Node() :maxv(-INF) {}
    };
    vector<Node> SegT;

    SegmentTree() {}
    SegmentTree(int _n, const vector<int>& _a) :n(_n), a(_a) {
        SegT.resize(n + 1 << 2);
        build(1, 1, n);
    }

    friend Node operator+(const Node A, const Node B) {
        Node res;
        res.maxv = max(A.maxv, B.maxv);
        return res;
    }

    void pushUp(int u) {
        SegT[u] = SegT[u << 1] + SegT[u << 1 | 1];
    }

    void build(int u, int l, int r) {
        if (l == r) {
            SegT[u].maxv = a[l];
            return;
        }

        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushUp(u);
    }

    Node query(int u, int l, int r, int L, int R) {
        if (R < l || L > r) return Node();
        if (L <= l && r <= R) return SegT[u];

        int mid = l + r >> 1;
        return query(u << 1, l, mid, L, R) + query(u << 1 | 1, mid + 1, r, L, R);
    }

    int query(int L, int R) {  // 求max a[L...R]
        return query(1, 1, n, L, R).maxv;
    }

    void modify(int u, int l, int r, int pos, int val) {
        if (l == r && l == pos) {
            SegT[u].maxv = val;
            return;
        }

        int mid = l + r >> 1;
        if (pos <= mid) modify(u << 1, l, mid, pos, val);
        else modify(u << 1 | 1, mid + 1, r, pos, val);
        pushUp(u);
    }

    void modify(int pos, int val) {  // a[pos] = val
        modify(1, 1, n, pos, val);
    }
};

struct Solver {
    int n;
    vector<vector<pair<int, int>>> edges;
    HeavyPathDecomposition sol;
    vector<int> father, depth, dfn, top;
    SegmentTree st;

    Solver(int _n, const vector<vector<pair<int, int>>>& _edges) :n(_n), edges(_edges) {
        sol = HeavyPathDecomposition(n, edges);
        father = sol.father, depth = sol.depth,
        dfn = sol.dfn, top = sol.top;

        auto idfn = sol.idfn, w = sol.w;
```

```
        vector<int> a(n + 1);
        for (int tim = 1; tim <= n; tim++) a[tim] = w[idfn[tim]];

        st = SegmentTree(n, a);
    }

    int query(int u, int v) {  // 求节点u到节点v的路径的最大边权
        int lca = sol.lca(u, v);
        int tmp = st.query(dfn[lca], dfn[lca]);  // LCA(u, v)的点权
        st.modify(dfn[lca], -INF);  // 暂时修改LCA的点权

        int res = 0;
        while (top[u] != top[v]) {
            if (depth[top[u]] < depth[top[v]]) swap(u, v);  // 保证u深度大
            res = max(res, st.query(dfn[top[u]], dfn[u]));
            u = father[top[u]];
        }
        if (depth[u] > depth[v]) swap(u, v);  // u = LCA(u, v)
        res = max(res, st.query(dfn[u], dfn[v]));

        st.modify(dfn[lca], tmp);  // 撤销对LCA点权的修改
        return res;
    }
};
```

思路III

MST中节点u与节点v间的简单路径的最大边权等于Kruskal重构树上$LCA(u,v)$的点权, 倍增求Kruskal重构树的LCA即可.

代码III

```
const ll INFF = 0x3f3f3f3f3f3f3f3f;

struct KruskalTree {
    int n, m;
    vector<tuple<int, int, int, int>> edges;  // w, u, v, id
    vector<int> fa;
    vector<int> used;  // 记录边是否在生成树中
    int idx;  // 当前用到的节点编号
    vector<vector<int>> tr;  // Kruskal重构树
    vector<int> weights;  // Kruskal重构树中节点的点权

    KruskalTree(int _n, int _m, const vector<tuple<int, int, int, int>>& _edges)
        :n(_n), m(_m), edges(_edges), idx(n) {
        fa.resize(2 * n + 1), used.resize(m + 1);
        tr.resize(2 * n + 1), weights.resize(2 * n + 1);
        iota(all(fa), 0);
        sort(all(edges));
    }

    int find(int x) {
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }

    ll kruskal() {
        ll res = 0;
        int cnt = 0;  // 当前连的边数

        for (auto [w, u, v, id] : edges) {
            int tmpu = find(u), tmpv = find(v);
            if (tmpu != tmpv) {
                // 新建节点并连边
                fa[tmpu] = fa[tmpv] = ++idx;
                tr[tmpu].push_back(idx), tr[idx].push_back(tmpu);
                tr[tmpv].push_back(idx), tr[idx].push_back(tmpv);
                weights[idx] = w;

                used[id] = true;
                res += w, cnt++;
            }
        }
        return cnt == n - 1 ? res : INFF;
    }
};

struct RMQLCA {
    int MAXJ;
    int n;
    vector<vector<int>> edges;
    vector<int> depth;
```

```
    vector<vector<int>> fa;

    RMQLCA() {}
    RMQLCA(int _n, const vector<vector<int>>& _edges)
        :n(_n), edges(_edges), MAXJ(log2(_n) + 1) {
        depth = vector<int>(n + 1, INF);
        fa = vector<vector<int>>(n + 1, vector<int>(MAXJ + 1));
    }

    void bfs(int root) {  // 预处理depth[], fa[][]: 根节点为root
        depth[0] = 0, depth[root] = 1;

        queue<int> que;
        que.push(root);

        while (que.size()) {
            auto u = que.front(); que.pop();
            for (auto v : edges[u]) {
                if (depth[v] > depth[u] + 1) {
                    depth[v] = depth[u] + 1;
                    que.push(v);

                    fa[v][0] = u;
                    for (int k = 1; k <= MAXJ; k++)
                        fa[v][k] = fa[fa[v][k - 1]][k - 1];
                }
            }
        }
    }

    int lca(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);  // 保证节点u深度大

        // 将节点u与节点v跳到同一深度
        for (int k = MAXJ; k >= 0; k--)
            if (depth[fa[u][k]] >= depth[v]) u = fa[u][k];

        if (u == v) return u;  // u与v原本在一条链上

        // u与v同时跳到LCA的下一层
        for (int k = MAXJ; k >= 0; k--) {
            if (fa[u][k] != fa[v][k])
                u = fa[u][k], v = fa[v][k];
        }
        return fa[u][0];  // 节点u向上跳1步到LCA
    }
};

void solve() {
    int n, m; cin >> n >> m;
    vector<tuple<int, int, int, int>> eds;  // w, u, v, id
    for (int i = 1; i <= m; i++) {
        int u, v, w; cin >> u >> v >> w;
        eds.push_back({ w, u, v, i });
    }

    KruskalTree kr(n, m, eds);
    ll sum = kr.kruskal();
    RMQLCA sol(kr.idx, kr.tr);  // 注意第一个参数为Kruskal重构树当前用到的节点编号
    sol.bfs(kr.idx);

    for (auto [w, u, v, id] : eds) {
        if (kr.used[id])  // 该边包含于该图的MST中
            cout << sum << endl;
        else {
            auto maxLength = kr.weights[sol.lca(u, v)];
            cout << sum - maxLength + w << endl;
        }
    }
}

int main() {
    solve();
}
```