线段树

李淳风

长郡中学

2024 年 6 月 24 日

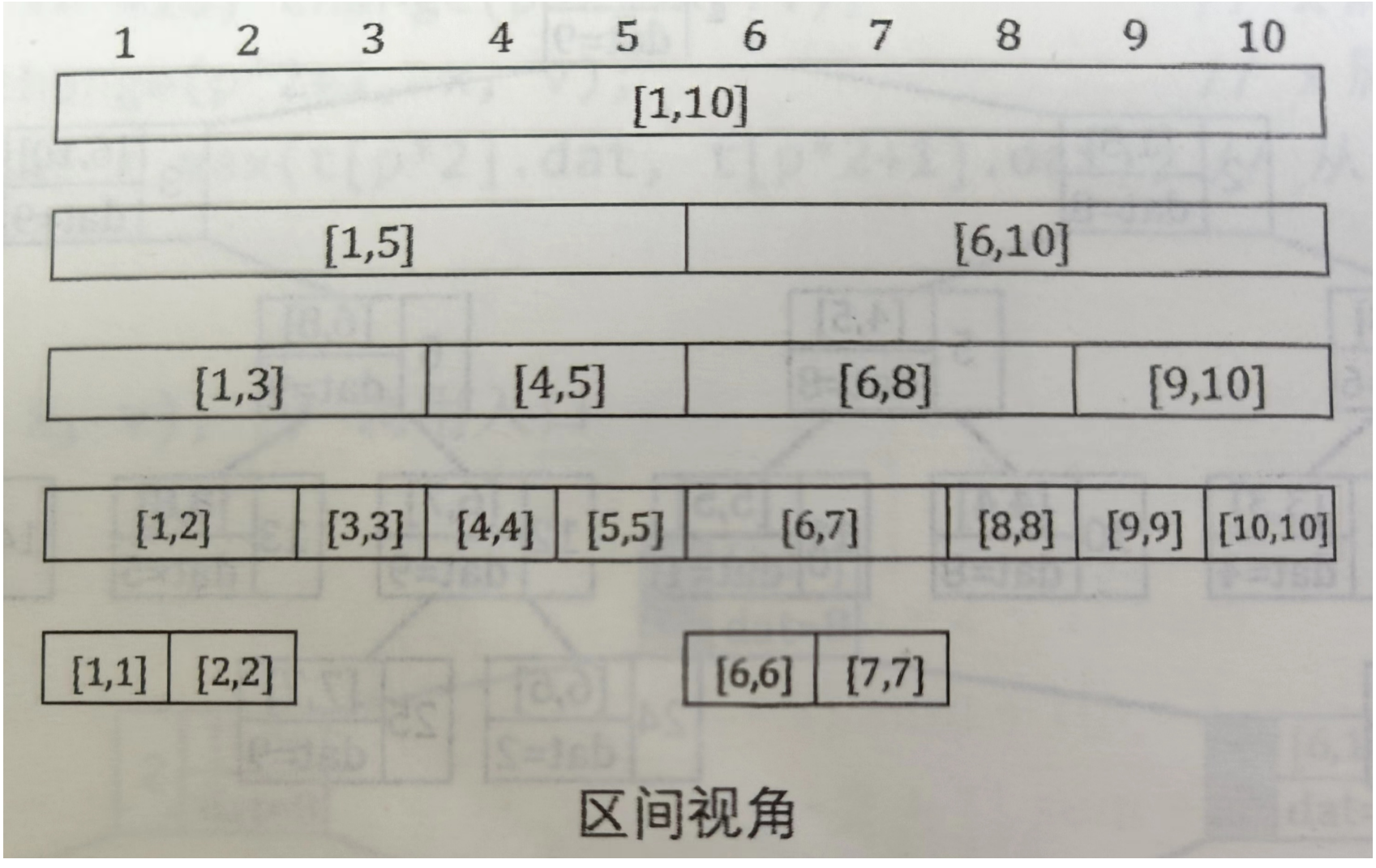
引入

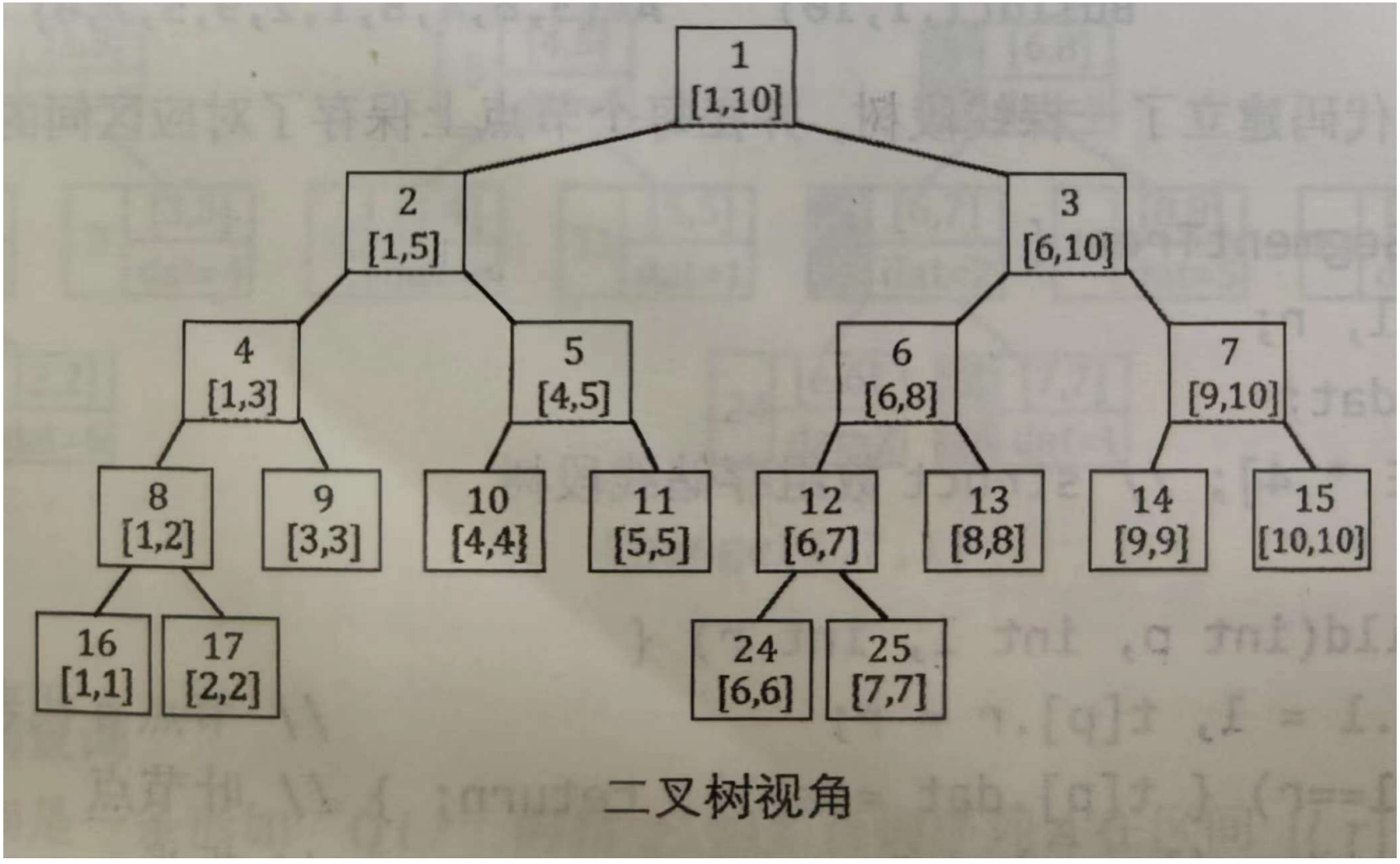
线段树是算法竞赛中常用的用来维护区间信息的数据结构。
线段树可以在单次 $O(\log n)$ 的时间复杂度内实现单点修改、区间修改、区间查询（区间求和、最大值、最小值）等操作。
事实上，线段树可以维护的区间信息非常之多。基本上，只要是能够通过合并两个子区间的信息，来得出新区间的信息，在时间复杂度合适的情况下，都可以使用线段树维护。

原理

线段树是一种基于分治思想的二叉树结构，用于在区间上进行信息统计。与按照二进制位（2 的次幂）进行区间划分的树状数组相比，线段树是一种更加通用的结构。

- 线段树的每个节点都代表一个区间。
- 线段树具有唯一的根节点，代表的区间是整个统计范围，譬如 $[1, n]$。
- 线段树的每个叶子节点都代表一个长度为 1 的区间 $[x, x]$。
- 对于每个内部节点 $[l, r]$，它的左子节点是 $[l, mid]$，右子节点是 $[mid + 1, r]$，其中 $mid = \lfloor (l + r)/2 \rfloor$。


1 2 3 4 5 6 7 8 9 10
[1,10]
[1,5]
[6,10]
[1,3]
[4,5]
[6,8]
[9,10]
[1,2]
[3,3]
[4,4]
[5,5]
[6,7]
[8,8]
[9,9]
[10,10]
[1,1]
[2,2]
[6,6]
[7,7]
区间视角


1
[1,10]
2
[1,5]
3
[6,10]
4
[1,3]
5
[4,5]
6
[6,8]
7
[9,10]
8
[1,2]
9
[3,3]
10
[4,4]
11
[5,5]
12
[6,7]
13
[8,8]
14
[9,9]
15
[10,10]
16
[1,1]
17
[2,2]
24
[6,6]
25
[7,7]

二叉树视角

原理

可以发现，除去最后一层，线段树一定是一棵完全二叉树，树的深度为 $\log n$。因此，我们可以按照二叉堆的编号方法进行编号。

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

这样一来，我们就能使用多个数组，或者一个结构体数组来保存线段树。当然，线段树的最后一层在数组中不是连续的，空出不表示节点的位置即可。在理想情况下，n 个叶子节点的满二叉树有 $n+n/2+n/4+\cdots+2+1=2n-1$ 个节点。而线段树由于最后一层还有空余，所以保存线段树的数组长度要不小于 $4n$ 才能保证不越界。

线段树的建树

线段树的基本用途是对序列进行维护，支持查询与修改指令。给定一个长度为 n 的序列 a，我们可以在区间 $[1, n]$ 上建立一棵线段树，每个叶子节点 $[i, i]$ 保存 a_i 的值。

通常来说，只需要我们实现了查询与修改操作，线段树的初始化过程可以视为 n 次修改，这并不会影响整体的时间复杂度。然而由于线段树常数略大，我们还是尽量使用 $O(n)$ 的时间复杂度进行建树。

建树过程也并不复杂，线段树的二叉树结构可以很方便地把信息从下往上进行合并，递归之后从下往上依次合并即可。

线段树的建树

线段树的基本用途是对序列进行维护，支持查询与修改指令。给定一个长度为 n 的序列 a，我们可以在区间 $[1, n]$ 上建立一棵线段树，每个叶子节点 $[i, i]$ 保存 a_i 的值。

通常来说，只需要我们实现了查询与修改操作，线段树的初始化过程可以视为 n 次修改，这并不会影响整体的时间复杂度。然而由于线段树常数略大，我们还是尽量使用 $O(n)$ 的时间复杂度进行建树。

建树过程也并不复杂，线段树的二叉树结构可以很方便地把信息从下往上进行合并，递归之后从下往上依次合并即可。

```
struct SegmentTree{
  int l,r,dat;
}t[SIZE*4];//struct 数组存储线段树
void build(int p,int l,int r){  //节点 p 代表区间 [l,r]
  t[p].l=l,t[p].r=r;
  if(l==r){t[p].dat=a[l]; return;} //叶子节点
  int mid=(l+r)>>1;                //折半
  build(p<<1,l,mid);               //左子节点
  build(p<<1|1,mid+1,r)            //右子节点
  t[p].dat=max(t[p<<1].dat,t[p<<1|1].dat);//从下往上更新
}
build(1,1,n); //调用入口
```

线段树的单点修改

单点修改是一条形如"C x v" 的指令，表示把 a_x 的值修改为 v。
在线段树中，根节点是执行各种指令的入口。我们需要从根节点出发，递归找到代表区间 $[x, x]$ 的叶子节点，然后从下往上更新 $[x, x]$ 这个节点以及它的所有祖先节点上保存的信息。时间复杂度为 $O(\log n)$。

线段树的单点修改

单点修改是一条形如"C x v" 的指令，表示把 a_x 的值修改为 v。
在线段树中，根节点是执行各种指令的入口。我们需要从根节点出发，递归找到代表区间 $[x, x]$ 的叶子节点，然后从下往上更新 $[x, x]$ 这个节点以及它的所有祖先节点上保存的信息。时间复杂度为 $O(\log n)$。

```
void change(int p,int x,int v){  //节点 p 代表区间 [l,r]
  if(t[p].l==t[p].r){t[p].dat=v; return;} //叶子节点
  int mid=(t[p].l+t[p].r)>>1;             //折半
  if(x<=mid) change(p<<1,x,v);            //左子节点
  else change(p<<1|1,x,v);                //右子节点
  t[p].dat=max(t[p<<1].dat,t[p<<1|1].dat);//从下往上更新
}
change(1,1,n); //调用入口
```

线段树的区间查询

区间查询是一条形如"Q l r" 的指令，例如查询序列 a 在 $[l, r]$ 上的最大值。我们只需要从根节点开始，执行如下过程：

- 若 $[l, r]$ 完全覆盖了当前节点代表的区间，则立即回溯，并将该节点的 dat 值作为候选答案。
- 若左子节点与 $[l, r]$ 有重叠部分，则递归访问左子节点。
- 若右子节点与 $[l, r]$ 右重叠部分，则递归访问右子节点。

线段树的区间查询

区间查询是一条形如"Q l r" 的指令，例如查询序列 a 在 $[l, r]$ 上的最大值。我们只需要从根节点开始，执行如下过程：

- 若 $[l, r]$ 完全覆盖了当前节点代表的区间，则立即回溯，并将该节点的 dat 值作为候选答案。
- 若左子节点与 $[l, r]$ 有重叠部分，则递归访问左子节点。
- 若右子节点与 $[l, r]$ 右重叠部分，则递归访问右子节点。

该过程会把询问区间 $[l, r]$ 在线段树上分成 $O(\log n)$ 个节点。我们可以分情况讨论来证明。

- $l \le p_l \le p_r \le r$，询问区间完全覆盖了当前节点，直接返回。
- $p_l \le l \le p_r \le r$，若 $l > mid$ 则只会递归右子树，若 $l \le mid$ 则右子树会在递归后直接返回，相当于只会递归左子树。
- $l \le p_l \le r \le p_r$，该情况同上。
- $p_l \le l \le r \le p_r$，若 l, r 都位于 mid 的一侧，则只会递归一棵子树；若 l, r 分别位于 mid 两侧，会递归左右两棵子树。

也就是说，只有最后一种情况会对两棵子树进行递归，但是递归后由于 l, r 至少有一个不在当前节点代表的区间内，所以不会再出现情况 4，也就是不会再多出新的分支，因此复杂度为 $O(\log n)$。

线段树的区间查询

```
int ask(int p,int l,int r){
  if(l<=t[p].l && r>=t[p].r) return t[p].dat;
  int mid=(t[p].l+t[p].r)>>1;
  int val=-(1<<30);
  if(l<=mid) val=max(val,ask(p<<1,l,r));
  if(r>mid) val=max(val,ask(p<<1|1,l,r));
  return val;
}
```

线段树的区间查询

```
int ask(int p,int l,int r){
  if(l<=t[p].l && r>=t[p].r) return t[p].dat;
  int mid=(t[p].l+t[p].r)>>1;
  int val=-(1<<30);
  if(l<=mid) val=max(val,ask(p<<1,l,r));
  if(r>mid) val=max(val,ask(p<<1|1,l,r));
  return val;
}
```

至此，线段树已经能够支持维护单点修改和区间查询了，并且可以维护许多树状数组无法维护的信息，譬如区间最大值。

例题

Can you answer these queries III

给定长度为 n 的数列 a，以及 m 条指令，每条指令可能是：

- "1 x y"，查询区间 $[x, y]$ 中的最大连续子段和，即 $\max_{x \le l \le r \le y} \sum_{i=l}^{r} a_i$。
- "2 x y"，把 a_x 改成 y。

$n \le 5 \times 10^5, m \le 10^5$。

例题

Can you answer these queries III

给定长度为 n 的数列 a，以及 m 条指令，每条指令可能是：

- "1 x y"，查询区间 $[x, y]$ 中的最大连续子段和，即 $\max_{x \le l \le r \le y} \sum_{i=l}^{r} a_i$。
- "2 x y"，把 a_x 改成 y。

$n \le 5 \times 10^5, m \le 10^5$。

我们考虑一下如何用分治的做法来求最大子段和。每个区间的最大子段和，可能全部在左儿子中，右儿子中，或者从 mid 开始，分别往左右延伸一段。
所以我们除了区间端点之外，再维护四个信息：区间和 sum，区间最大连续子段和 dat，紧靠左端的最大连续子段和 $lmax$，紧靠右端的最大连续子段和 $rmax$。

例题

线段树的整体结构不变，我们只需要考虑加入的这些信息如何维护即可。

```
int ls=p<<1,rs=p<<1|1;
t[p].lmax=max(t[ls].lmax,t[ls].sum+t[rs].lmax);
t[p].rmax=max(t[rs].rmax,t[rs].sum+t[ls].rmax);
t[p].dat=max(max(t[ls].dat,t[rs].dat),t[ls].rmax+t[rs].lmax);
```

例题

线段树的整体结构不变，我们只需要考虑加入的这些信息如何维护即可。

```
int ls=p<<1,rs=p<<1|1;
t[p].lmax=max(t[ls].lmax,t[ls].sum+t[rs].lmax);
t[p].rmax=max(t[rs].rmax,t[rs].sum+t[ls].rmax);
t[p].dat=max(max(t[ls].dat,t[rs].dat),t[ls].rmax+t[rs].lmax);
```

从这里我们也可以看出，线段树能够维护的各种信息，需要满足可以按照区间进行划分与合并（又称区间可加性），也就是可以通过两个子区间的合并来得到大区间的信息。
我们只需要明确自己需要维护哪些信息，以及这些信息如何通过子区间信息的合并来计算即可。

例题

Interval GCD

给定长度为 n 的数列 a，以及 m 条指令，每条指令可能是：

- "C l r d"，表示把 $a_l, a_{l+1}, \cdots, a_r$ 都加上 d。
- "Q l r"，表示询问 $a_l, a_{l+1}, \cdots, a_r$ 的最大公约数。

$n \leq 5 \times 10^5, m \leq 10^5$。

例题

Interval GCD

给定长度为 n 的数列 a，以及 m 条指令，每条指令可能是：

- "C l r d"，表示把 $a_l, a_{l+1}, \cdots, a_r$ 都加上 d。
- "Q l r"，表示询问 $a_l, a_{l+1}, \cdots, a_r$ 的最大公约数。

$n \leq 5 \times 10^5, m \leq 10^5$。

我们知道 $\gcd(x, y) = \gcd(x, y - x)$。实际上，它可以拓展到三个数的情况：$\gcd(x, y, z) = \gcd(x, y - x, z - y)$。
因此我们可以构造一个 a 的差分数列 b，满足 $b_i = a_i - a_{i-1}, b_1 = a_1$。
我们使用线段树来维护 b 数组的区间 $\gcd$，这样一来，询问"Q l r"，就等于求 $\gcd(a_l, ask(1, l + 1, r))$。
而区间加法对应到差分数列上面，两次单点修改即可。

延迟标记

在“区间查询”指令中，我们已经知道了，被询问区间 $[l, r]$ 会在线段树上被分成 $O(\log n)$ 个小区间，并在 $O(\log n)$ 的时间内求出答案。但是，在这一过程中，我们并没有访问 $[l, r]$ 区间内的每一个叶子节点，因为这样做的复杂度是 $O(n)$ 的。因此，如果我们想要进行区间修改操作，没办法每次修改所有被影响的节点，只能通过别的手段来实现。这一手段就是“延迟标记”。当我们在执行修改指令时，同样在 $l \leq p_l \leq p_r \leq r$ 的情况下直接返回，不再递归它的儿子，同时我们需要给它打上一个标记，表示“该节点的所有子节点需要被修改，但目前尚未更新”。

如果在后续的指令中需要从 p 节点向下接着递归，我们再检查 p 节点是否具有标记。若有标记，就根据标记更新 p 的两个子节点，同时为子节点增加标记，并删除 p 的标记。需要注意的是，如果子节点已经有标记了，那么需要把来自父节点的标记和它本身的标记进行合并。这样一来，对任意节点的修改都被推迟到了“后续操作中访问了它的父节点”时再进行，单次修改的复杂度做到了 $O(\log n)$。延迟标记提供了一种线段树从上往下传递信息的方式。

延迟标记

```
void pushdown(int p){
  if(t[p].add){
    int ls=p<<1,rs=p<<1|1;
    t[ls].sum+=1ll*t[p].add*(t[ls].r-t[ls].l+1);//更新儿子的信息
    t[rs].sum+=1ll*t[p].add*(t[rs].r-t[rs].l+1);
    t[ls].add+=t[p].add,t[rs].add+=t[p].add      //更新儿子的标记
    t[p].add=0;                                  //删除 p 本身的标记
  }
}
void change(int p,int l,int r,int d){ //[l,r] 区间加 d
  if(l<=t[p].l && r>=t[p].r){      //被完全覆盖，打上标记并更新自身
    t[p].sum += 1ll*d*(t[p].r-t[p].l+1);
    t[p].add += d;
  }
  pushdown(p);                               //下传标记
  int mid=(t[p].l+t[p].r)>>1;
  if(l<=mid) change(p<<1,l,r,d);             //递归
  if(r>mid) change(p<<1|1,l,r,d);
  t[p].sum=t[p<<1].sum+t[p<<1|1].sum; //更新
}
```

动态开点

在一些问题中，我们需要使用线段树来维护值域（一段权值范围），这样的线段树也被称为值域线段树。为了降低复杂度，我们可以不构造出整棵线段树，而是初始时只建立一个根节点，代表整个区间，当需要访问线段树上的某个子区间时，再新建一个代表该子区间的节点。采用这种方法维护的线段树杯称为动态开点的线段树，它抛弃了完全二叉树父子节点的 2 倍编号规则，改为使用变量记录节点的编号。

动态开点

在一些问题中，我们需要使用线段树来维护值域（一段权值范围），这样的线段树也被称为值域线段树。为了降低复杂度，我们可以不构造出整棵线段树，而是初始时只建立一个根节点，代表整个区间，当需要访问线段树上的某个子区间时，再新建一个代表该子区间的节点。采用这种方法维护的线段树杯称为动态开点的线段树，它抛弃了完全二叉树父子节点的 2 倍编号规则，改为使用变量记录节点的编号。

```
struct SegmentTree{
    int lc,rc; //左右子节点的编号
    int dat;  //区间最大值
}tr[SIZE];
int root,tot;
int build(){  //新建一个节点并返回
    tot++;
    tr[tot].lc=tr[tot].rc=0;
    tr[tot].dat=-(1<<30);
    return tot;
}
```

动态开点

下面这段代码给出了使用动态开点的线段树，进行单点修改并维护区间最大值的操作。

```
void insert(int p,int l,int r,int val,int delta){
    if(l==r){
        tr[p].dat+=delta;
        return;
    }
    int mid=(l+r)>>1;
    if(val<=mid){
        if(!tr[p].lc) tr[p].lc=build();//新开节点
        insert(tr[p].lc,l,mid,val,delta);
    }
    if(val>mid){
        if(!tr[p].rc) tr[p].rc=build();
        insert(tr[p].rc,mid+1,r,val,delta);
    }
    tr[p].dat=max(tr[tr[p].lc].dat,tr[tr[p].rc].dat);
}
insert(root,1,n,val,delta);//调用
```

动态开点与线段树合并

常规线段树的其它操作也可以通过类似的改动，在动态开点的线段树上实现。注意每次进入其它节点的时候都需要判断，是否需要新建一个节点。

一棵维护值域 $[1, n]$ 的动态开点的线段树在经历 m 次单点操作后，节点数量的规模为 $O(m \log n)$，最终至多有 $2n - 1$ 个节点。

动态开点与线段树合并

常规线段树的其它操作也可以通过类似的改动，在动态开点的线段树上实现。注意每次进入其它节点的时候都需要判断，是否需要新建一个节点。

一棵维护值域 $[1, n]$ 的动态开点的线段树在经历 m 次单点操作后，节点数量的规模为 $O(m \log n)$，最终至多有 $2n - 1$ 个节点。

如果有若干棵线段树，它们都维护相同的值域 $[1, n]$，那么它们对各个子区间的划分，也就是线段树的结构，显然是一致的。假设有 m 次单点修改操作，每次操作在某一棵线段树上执行。所有操作完成后，我们希望把这些线段树对应位置上的值相加，同时维护区间最大值。

线段树合并

该问题可以通过线段树合并算法实现。我们依次合并这些线段树。合并两棵线段树时，用两个指针 p, q 从两个根节点出发，以递归的方式同步遍历两棵线段树。这意味着，p, q 指向的节点总是代表相同的子区间。

- 若 p, q 之一为空，则以非空的那个作为合并后的节点。
- 若 p, q 均不为空，则递归合并两棵左子树和两棵右子树，然后删除节点 q，以 p 为合并后的节点，自底向上更新最值信息。若已经到达叶子节点，则直接把两个最值相加。

```
int merge(int p,int q,int l,int r){
  if(!p || !q) return p+q;//p,q 有一个为空
  if(l==r){   //到达叶子节点
    tr[p].dat+=tr[q].dat;
    return p;
  }
  int mid=(l+r)>>1;
  tr[p].lc=merge(tr[p].lc,tr[q].lc,l,mid);//递归合并左子树
  tr[p].rc=merge(tr[p].rc,tr[q].rc,mid+1,r);//递归合并右子树
  tr[p].dat=max(tr[tr[p].lc].dat,tr[tr[p].rc].dat);//更新
  return p;//返回 p，相当于删除 q
}
```

线段树合并

仔细观察可以发现，若线段树在合并的过程中发生递归，则一定会导致 p, q 之一被删除。因此，在完成所有线段树的合并之后，merge 函数被执行的次数不会超过所有线段树的节点总数加 1。因此，和启发式合并不同，整个合并过程的复杂度等于所有线段树的节点数之和，为 $O(m \log n)$。

例题

永无乡

永无乡包含 n 座岛，编号从 1 到 n，每座岛都有自己的独一无二的重要度，按照重要度可以将这 n 座岛排名，名次用 1 到 n 来表示。某些岛之间由巨大的桥连接，初始有 m 座桥。如果从岛 a 出发经过若干座（含 0 座）桥可以到达岛 b，则称岛 a 和岛 b 是连通的。
现在有 Q 次操作，每次操作是下面两种之一：

- B x y 表示在岛 x 与岛 y 之间修建一座新桥。
- Q x k 表示询问当前与岛 x 连通的所有岛中第 k 重要的是哪座岛，即所有与岛 x 连通的岛中重要度排名第 k 小的岛是哪座，请你输出那个岛的编号。

$1 \leq m \leq n \leq 10^5, 1 \leq q \leq 3 \times 10^5$

线段树合并

对于求 k 小值这么一个操作，除了平衡树，我们还可以使用值域线段树来解决。
具体地，我们对值域用线段树来维护每个值的出现次数，然后就可以在线段树上进行二分了。假设在线段树上当前访问的节点是 p，区间是 $[p_l, p_r]$，我们要求的是第 k 小值，那么我们把 k 和 p 左儿子的区间和（也就是在 $[p_l, mid]$ 范围内的数的出现次数）x 做比较。如果 $x \geq k$，那么我们要找的数在左儿子中；否则要找的数在右儿子中，我们把 k 减去 x，往右儿子递归即可。
现在这道题不但要求 k 小值，还要可以合并集合，而权值线段树是可以进行合并的。因此我们建立若干棵权值线段树，每次连边之前通过并查集判断是否需要合并线段树即可。

扫描线

Atlantis

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

扫描线

Atlantis

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

试想，如果我们用一条竖直直线从左到右扫过整个坐标系，那么直线上被并集图形覆盖的长度只会在每个矩形的左右边界处发生变化。换言之，整个并集图形被 $2n$ 条竖线分为了 $2n-1$ 段，每一段在直线上覆盖的长度 L 是固定的，因此该段的面积就是 $L*$ 该段的宽度，各段面积之和即为总面积。这条直线就被称为扫描线，这种解题思路就被称为扫描线法。

扫描线

具体来说，我们可以取出 n 个矩形的左右边界。若一个矩形的两个对角顶点坐标为 x_1, y_1 和 $x_2, y_2 (x_1 < x_2, y_1 < y_2)$，则左边界记为四元组 $(x_1, y_1, y_2, 1)$，右边界记为四元组 $(x_2, y_1, y_2, -1)$。
我们把这些四元组按照 x 递增排序，问题就转化为了：有若干个区间，求这些区间的并覆盖的总长度。我们先不考虑离散化的问题，用 c_i 表示 $[i, i+1]$ 这一段区间被覆盖的次数，那么对于四元组 (x, y_1, y_2, k)，我们就需要把 $c_{y_1}, c_{y_1+1}, \cdots, c_{y_2-1}$ 都加上 k，相当于覆盖了 $[y_1, y_2]$ 这样一个区间。总的被覆盖的长度就是 $\sum_{c_i>0} 1$。若下一个四元组的横坐标是 x_2，我们就让最终的答案 ans 加上 $(x_2 - x) * (\sum_{c_i>0} 1)$。
需要注意的是，如果我们进行了离散化操作，假设 m 个不同的 y 坐标分别对应 $row_1, row_2, \cdots, row_m$，那么我们的 c_i 维护的第 i 段区间就不再是 $[i, i+1]$，而是 $[row_i, row_{i+1}]$，对应的长度也不再是 1，而是 $row_{i+1} - row_i$，总的被覆盖的长度就是 $\sum_{c_i>0} (row_{i+1} - row_i)$。

扫描线

需要注意的是，四元组中的 y_1, y_2 都是坐标，是“一个点”，我们需要维护的却是扫描线上每一段被覆盖的次数，因此对每个点被覆盖了多少次进行统计是没有意义的，因此我们 c_i 的定义是对第 i 段区间的覆盖次数。在碰到这类题目时一定要注意这一点。
在这道题中，使用线段树对 c 数组进行维护，就可以做到 $O(n \log n)$ 的复杂度了，但我们可以更进一步优化常数，那就是标记永久化。由于我们只关心整个扫描线（线段树根节点）上被覆盖的长度，而且四元组 $(x, y_1, y_2, 1)$，$(x, y_1, y_2, -1)$ 成对出现，所以对线段树上每个区间的修改也是成对出现的，所以我们可以对每个节点记录两个值：该节点代表的区间被覆盖的长度 len、该节点自身被覆盖的次数 cnt。最初这二者都为 0，而在更新的过程中，对于线段树上任意一个节点，若 $cnt \neq 0$，则 len 为该区间的长度，否则 len 为两个子节点的 len 值之和。

例题

Stars in Your Window

在天空中有很多星星（看作平面直角坐标系），已知每颗星星的坐标和亮度（都是整数）。求用宽为 w、高为 h 的矩形（w, h 为正整数）能圈住的星星的亮度总和最大是多少（矩形边界上的星星不算）。

例题

Stars in Your Window

在天空中有很多星星（看作平面直角坐标系），已知每颗星星的坐标和亮度（都是整数）。求用宽为 w、高为 h 的矩形（w, h 为正整数）能圈住的星星的亮度总和最大是多少（矩形边界上的星星不算）。

由于矩形的大小固定，所以矩形可以由它的任意一个顶点唯一确定。
我们可以考虑把矩形的右上角放在什么位置，才能使圈住的星星最大。
对于一颗星星 (x, y, c)，其中 x, y 是坐标，c 为亮度，“能圈住这颗星星的矩形右上角顶点”能取很多位置，这些位置也构成了一个矩形，左下角顶点为 (x, y)，右上角顶点为 $(x + w, y + h)$（边界不算）。
由于我们的矩形顶点不一定是整数，所以我们不太好操作。所以我们可以把每颗星星的坐标从 (x, y) 变成 $(x - 0.5, y - 0.5)$，这样最优解的矩形顶点就可以取到整数了。$(x - 0.5, y - 0.5)$ 的星星对应到的矩阵左下角为 (x, y)，右上角为 $(x + w - 1, y + h - 1)$（边界也算）。
于是，问题就转化为了：平面上有若干个矩形，每个矩形有一个权值，求在哪个坐标上重叠的区域权值和最大。每一个矩形都是由一颗星星产生的，权值等于星星的亮度。
我们同样使用扫描线算法，使用线段树维护扫描线上的区间最大值，每次进行区间加法即可。