线段树详解

By 岩之痕

目录:

一：综述
二：原理
三：递归实现
四：非递归原理
五：非递归实现
六：线段树解题模型
七：扫描线
八：可持久化 (主席树)
九：练习题

一：综述

假设有编号从1到n的n个点，每个点都存了一些信息，用[L,R]表示下标从L到R的这些点。
线段树的用处就是，对编号连续的一些点进行修改或者统计操作，修改和统计的复杂度都是O(log2(n)).

线段树的原理，就是，将[1,n]分解成若干特定的子区间(数量不超过4*n),然后，将每个区间[L,R]都分解为
少量特定的子区间，通过对这些少量子区间的修改或者统计，来实现快速对[L,R]的修改或者统计。

由此看出，用线段树统计的东西，必须符合**区间加法**，否则，不可能通过分成的子区间来得到[L,R]的统计结果。

符合区间加法的例子：
数字之和——总数字之和 = 左区间数字之和 + 右区间数字之和
最大公因数(GCD)——总GCD = gcd(左区间GCD , 右区间GCD);
最大值——总最大值=max(左区间最大值，右区间最大值)
不符合区间加法的例子：
众数——只知道左右区间的众数，没法求总区间的众数
01序列的最长连续零——只知道左右区间的最长连续零，没法知道总的最长连续零

一个问题，只要能化成对一些连续点的修改和统计问题，基本就可以用线段树来解决了，具体怎么转化在第六节会讲。
由于点的信息可以千变万化，所以线段树是一种非常灵活的数据结构，可以做的题的类型特别多，只要会转化。
线段树当然是可以维护线段信息的，因为线段信息也是可以转换成用点来表达的（每个点代表一条线段）。
所以在以下对结构的讨论中，都是对点的讨论，线段和点的对应关系在第七节扫描线中会讲。

本文二到五节是讲对线段树操作的原理和实现。
六到八节介绍了线段树解题模型，以及一些例题。

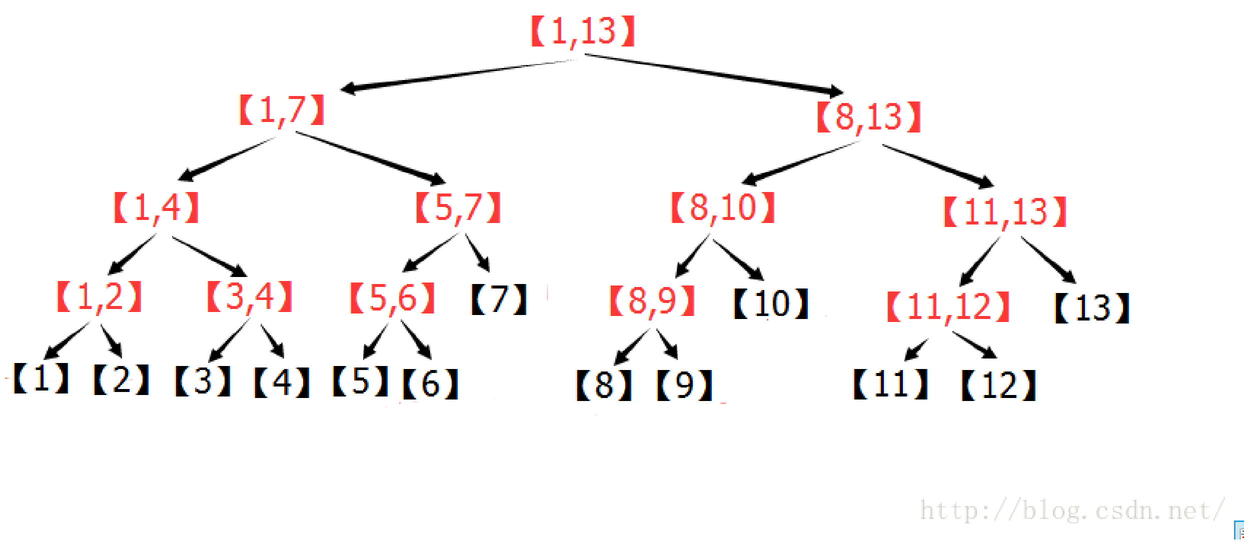
初学者可以先看这篇文章： 线段树从零开始

二：原理

（注：由于线段树的每个节点代表一个区间，以下叙述中不区分节点和区间，只是根据语境需要，选择合适的词）
线段树本质上是维护下标为1,2,..,n的n个按顺序排列的数的信息，所以，其实是“点树”，是维护n的点的信息，至于每个点的数据的含义可以有很多，
在对线段操作的线段树中，每个点代表一条线段，在用线段树维护数列信息的时候，每个点代表一个数，但本质上都是每个点代表一个数。以下，在讨论线段树的时候，区间[L,R]指的是下标从L到R的这(R-L+1)个数，而不是指一条连续的线段。只是有时候这些数代表实际上一条线段的统计结果而已。

线段树是将每个区间[L,R]分解成[L,M]和[M+1,R] (其中M=(L+R)/2 这里的除法是整数除法，即对结果下取整)直到 L==R 为止。

开始时是区间[1,n] ,通过递归来逐步分解，假设根的高度为1的话，树的最大高度为 $\lfloor \log_2(n-1) \rfloor + 2$ （n>1）。
线段树对于每个n的分解是唯一的，所以n相同的线段树结构相同，这也是实现可持久化线段树的基础。
下图展示了区间[1,13]的分解过程：



上图中，每个区间都是一个节点，每个节点存自己对应的区间的统计信息。

(1)线段树的点修改：

假设要修改[5]的值，可以发现，每层只有一个节点包含[5],所以修改了[5]之后，只需要每层更新一个节点就可以线段树每个节点的信息都是正确的，所以修改次数的最大值为层数 $\lfloor \log_2(n-1) \rfloor + 2$ 。
复杂度O(log2(n))

(2)线段树的区间查询：

线段树能快速进行区间查询的基础是下面的定理：

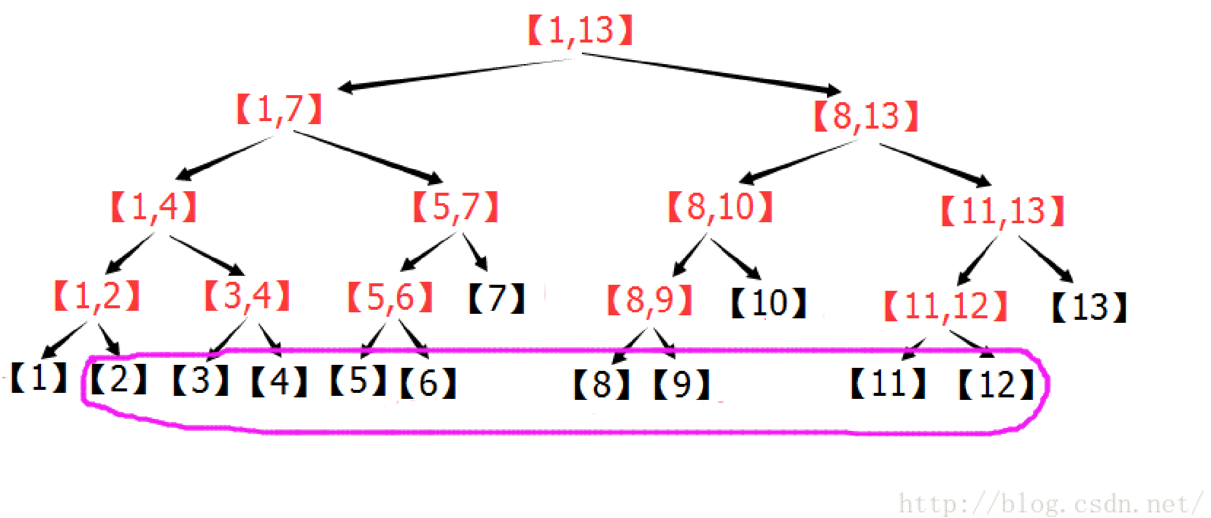
定理：n>=3时，一个[1,n]的线段树可以将[1,n]的任意子区间[L,R]分解为不超过 $2\lfloor \log_2(n-1) \rfloor$ 个子区间。

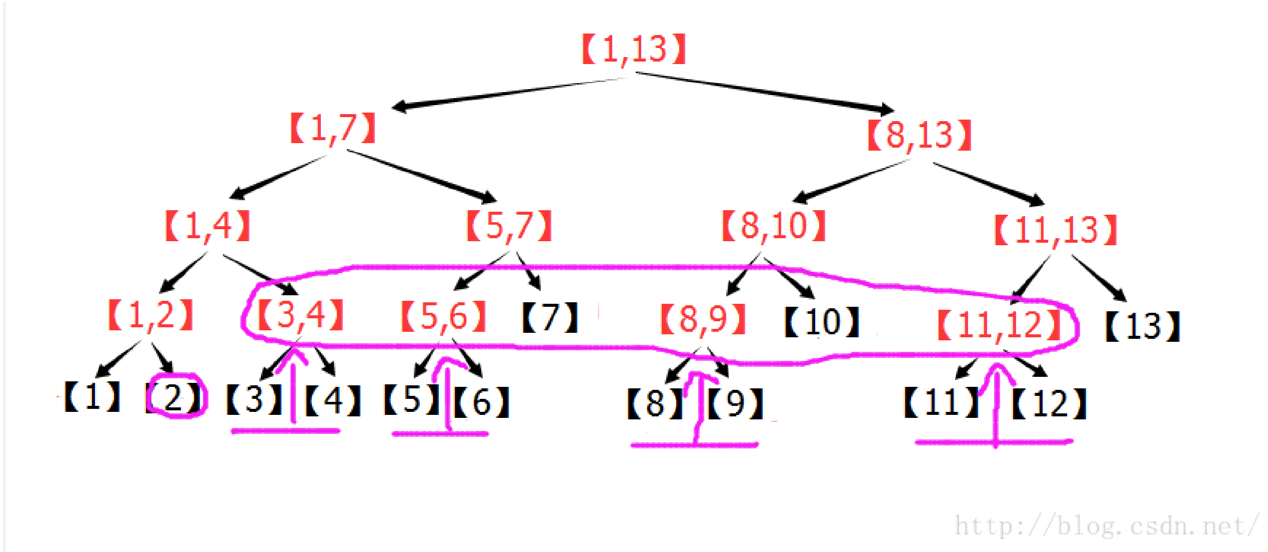
这样，在查询[L,R]的统计值的时候，只需要访问不超过 $2\lfloor \log_2(n-1) \rfloor$ 个节点，就可以获得[L,R]的统计信息，实现了O(log2(n))的区间查询。

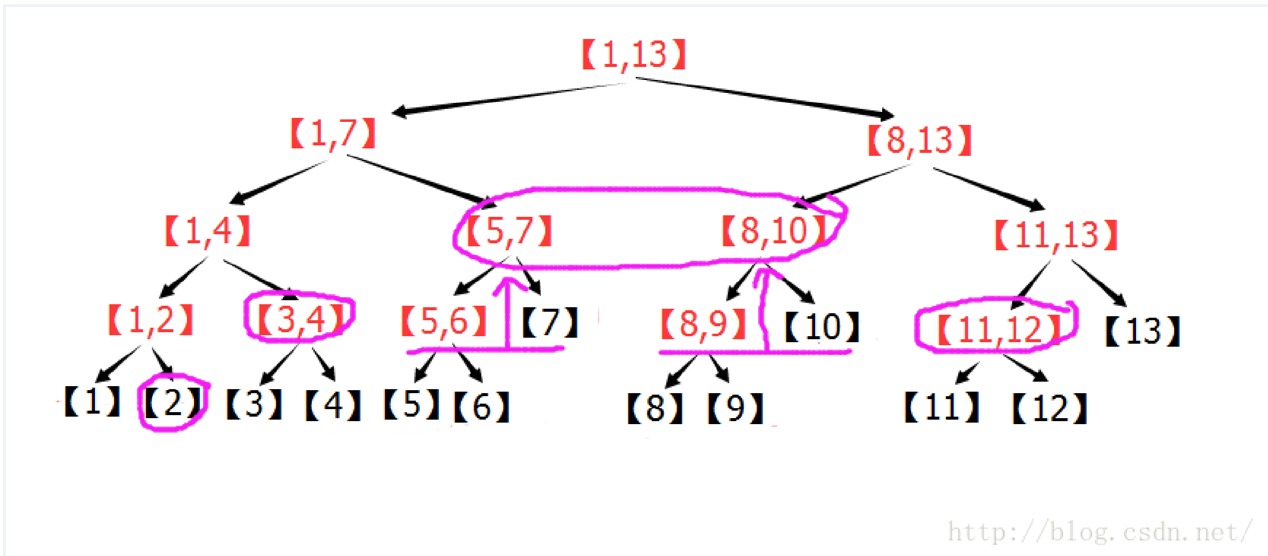
下面给出证明：

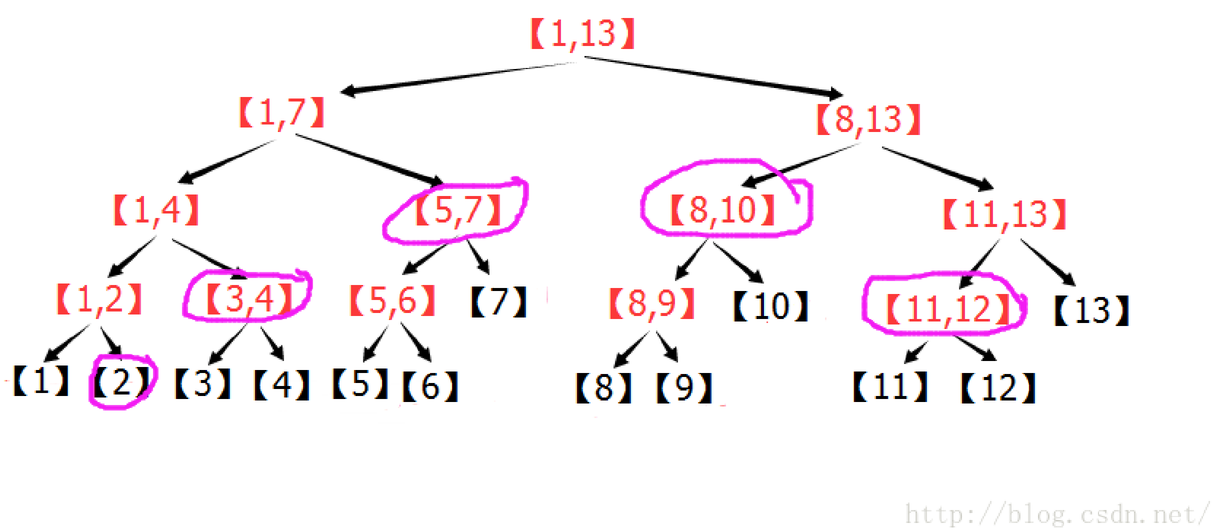
(2.1)先给出一个粗略的证明（结合下图）：
先考虑树的最下层，将所有在区间[L,R]内的点选中，然后，若相邻的点的直接父节点是同一个，那么就用这个父节点代替这两个节点（父节点在上一层）。这样操作之后，本层最多剩下两个节点。若最左侧被选中的节点是它父节点的右子树，那么这个节点会被剩下。若最右侧被选中的节点是它的父节点的左子树，那么这个节点会被剩下。中间的所有节点都被父节点取代。
对最下层处理完之后，考虑它的上一层，继续进行同样的处理，可以发现，每一层最多留下2个节点，其余的节点升往上一层，这样可以说明分割成的区间（节点）个数是大概是树高的两倍左右。

下图为n=13的线段树，区间[2,12]，按照上面的叙述进行操作的过程图：


【1,13】
【1,7】 【8,13】
【1,4】 【5,7】 【8,10】 【11,13】
【1,2】 【3,4】 【5,6】 【7】 【8,9】 【10】 【11,12】 【13】
【1】 【2】 【3】 【4】 【5】 【6】 【8】 【9】 【11】 【12】
http://blog.csdn.net/


【1,13】
【1,7】 【8,13】
【1,4】 【5,7】 【8,10】 【11,13】
【1,2】 【3,4】 【5,6】 【7】 【8,9】 【10】 【11,12】 【13】
【1】 【2】 【3】 【4】 【5】 【6】 【8】 【9】 【11】 【12】
http://blog.csdn.net/


【1,13】
【1,7】 【8,13】
【1,4】 【5,7】 【8,10】 【11,13】
【1,2】 【3,4】 【5,6】 【7】 【8,9】 【10】 【11,12】 【13】
【1】 【2】 【3】 【4】 【5】 【6】 【8】 【9】 【11】 【12】
http://blog.csdn.net/



由图可以看出：在n=13的线段树中，[2,12]=[2] + [3,4] + [5,7] + [8,10] + [11,12] 。

(2.2)然后给出正式一点的证明：

定理：n>=3时，一个[1,n]的线段树可以将[1,n]的任意子区间[L,R]分解为不超过 $2\lfloor \log_2(n-1) \rfloor$ 个子区间。

用数学归纳法，证明上面的定理：
首先,n=3,4,5时，用穷举法不难证明定理成立。
假设对于n= 3,4,5,...,k-1上式都成立，下面来证明对于n=k (k>=6)成立：
分为4种情况来证明：

情况一：[L,R]包含根节点(L=1且R=n)，此时，[L,R]被分解为了一个节点，定理成立。

情况二：[L,R]包含根节点的左子节点，此时[L,R]一定不包含根的右子节点（因为如果包含，就可以合并左右子节点，用根节点替代，此时就是情况一）。这时，以右子节点为根的这课树的元素个数为 $\left\lfloor \frac{k}{2} \right\rfloor \ge 3$ 。
[L,R]分成的子区间由两部分组成：
一：根的左子结点，区间数为1
二：以根的右子节点为根的树中，进行区间查询，这个可以递归使用本定理。

由归纳假设可得，[L,R]一共被分成了 $1+2\left\lfloor \log_2\left(\left\lfloor \frac{k}{2} \right\rfloor - 1\right) \right\rfloor$ 个区间。

情况三：跟情况二对称，不一样的是，以根的左子节点为根的树的元素个数为 $\left\lfloor \frac{k+1}{2} \right\rfloor \ge 3$ 。
[L,R]一共被分成了 $1+2\left\lfloor \log_2\left(\left\lfloor \frac{k+1}{2} \right\rfloor - 1\right) \right\rfloor$ 个区间。
从公式可以看出，情况二的区间数小于等于情况三的区间数，于是只需要证明情况三的区间数符合条件就行了。

$$
\begin{aligned}
&1+2\left\lfloor \log_2\left(\left\lfloor \frac{k+1}{2} \right\rfloor - 1\right) \right\rfloor \\
&=1+2\left\lfloor \log_2\left(\left\lfloor \frac{k-1}{2} \right\rfloor\right) \right\rfloor \\
&\le 1+2\left\lfloor \log_2\left(\frac{k-1}{2}\right) \right\rfloor \\
&=1+2\lfloor \log_2(k-1)-1 \rfloor \\
&=2\lfloor \log_2(k-1) \rfloor - 1 < 2\lfloor \log_2(k-1) \rfloor
\end{aligned}
$$

于是，情况二和情况三定理成立。

情况四：[L,R]不包括根节点以及根节点的左右子节点。

于是，剩下的 $\lfloor \log_2(k-1) \rfloor$ 层，每层最多两个节点（参考粗略证明中的内容）。

于是[L,R]最多被分解成了 $2\lfloor \log_2(k-1) \rfloor$ 个区间，定理成立。

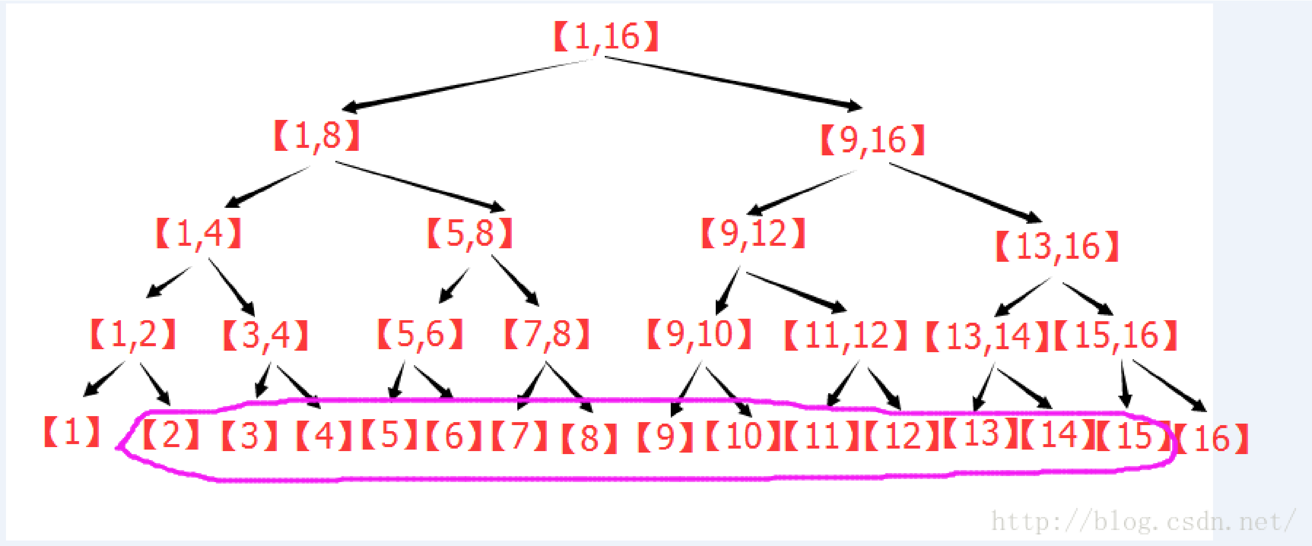
上面只证明了 $2\lfloor \log_2(k-1) \rfloor$ 是上界，但是，其实它是最小上界。

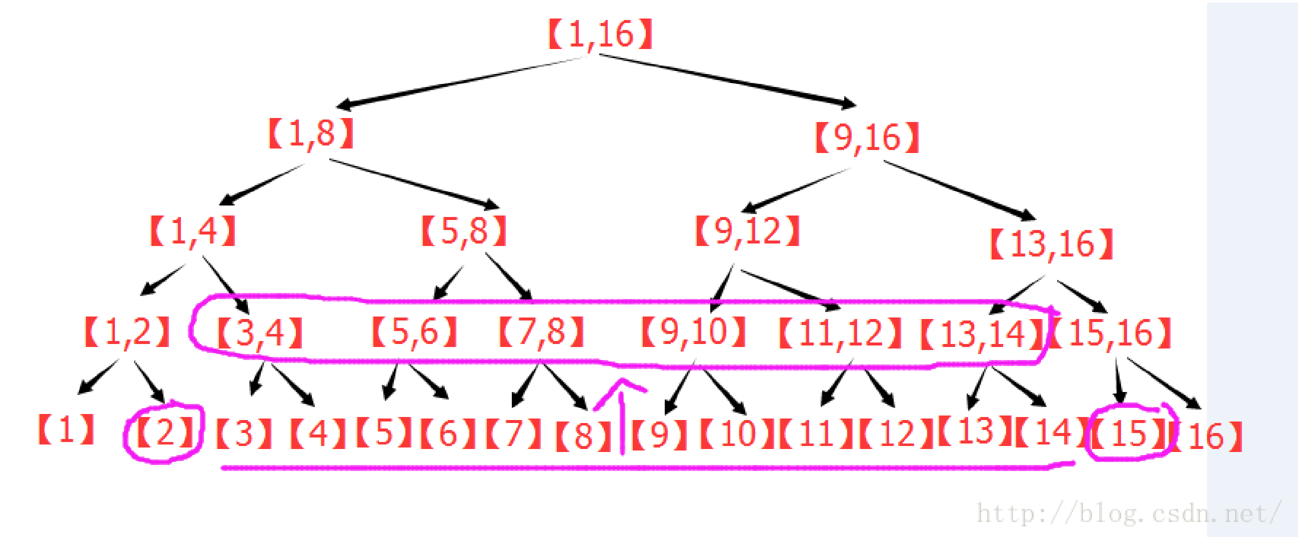
n=3,4时，有很多组区间的分解可以达到最小上界。

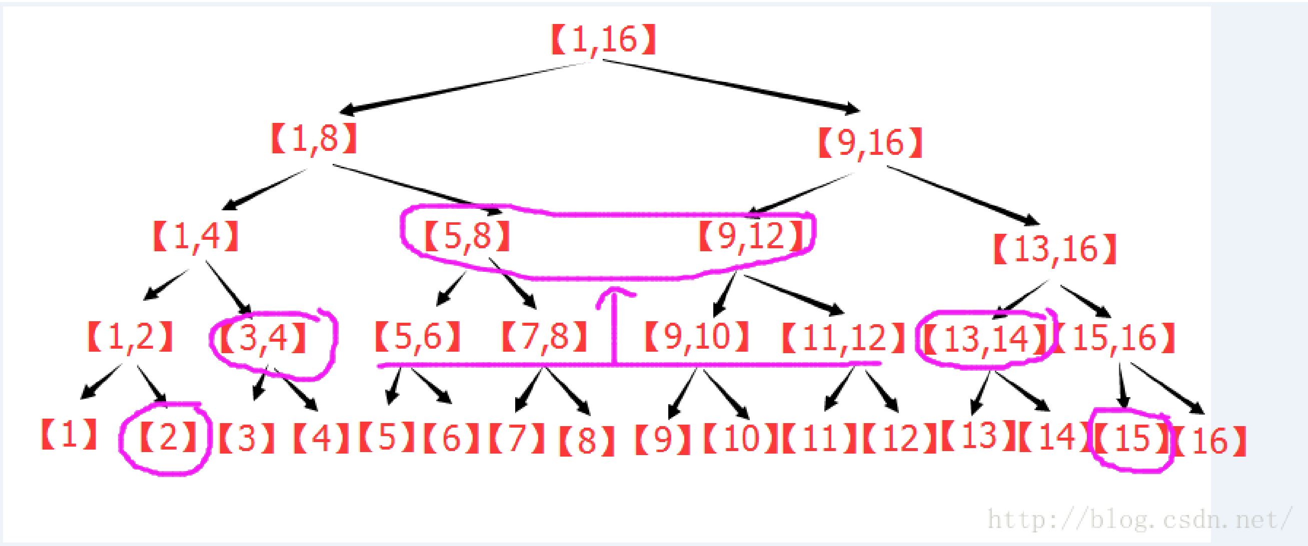
当n>4时，当且仅当n=2^t (t>=3),L=2,R=2^t -1 时，区间[L,R]的分解可以达到最小上界 $2\lfloor \log_2(k-1) \rfloor$ 。

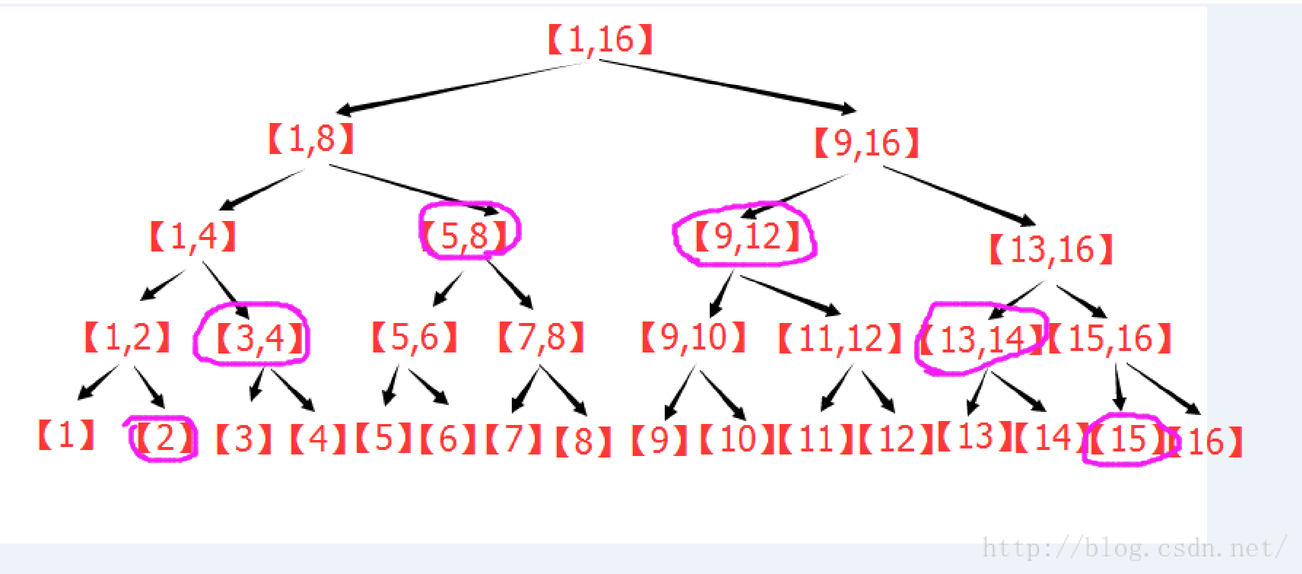
就不证明了，有兴趣可以自己去证明。

下图是n=16，L=2，R=15 时的操作图，此图展示了达到最小上界的树的结构。









(3)线段树的区间修改:

线段树的区间修改也是将区间分成子区间，但是要加一个标记，称作懒惰标记。

标记的含义:

本节点的统计信息已经根据标记更新过了，但是本节点的子节点仍需要进行更新。

即，如果要给一个区间的所有值都加上1，那么，实际上并没有给这个区间的所有值都加上1，而是打个标记，记下来，这个节点所包含的区间需要加1.打上标记后，要根据标记更新本节点的统计信息，比如，如果本节点维护的是区间和，而本节点包含5个数，那么，打上+1的标记之后，要给本节点维护的和+5。这是向下延迟修改，但是向上显示的信息是修改以后的信息，所以查询的时候可以得到正确的结果。有的标记之间会相互影响，所以比较简单的做法是，每递归到一个区间，首先下推标记（若本节点有标记，就下推标记），然后再打上新的标记，这样仍然每个区间操作的复杂度是O(log2(n))。

标记有**相对标记**和**绝对标记**之分:

相对标记是将区间的所有数+a之类的操作，标记之间可以共存，跟打标记的顺序无关（跟顺序无关才是重点）。

所以，可以在区间修改的时候不下推标记，留到查询的时候再下推。

注意: 如果区间修改时不下推标记，那么PushUp函数中，必须考虑本节点的标记。

而如果所有操作都下推标记，那么PushUp函数可以不考虑本节点的标记，因为本节点的标记一定已经被下推了（也就是对本节点无效了）

绝对标记是将区间的所有数变成a之类的操作，打标记的顺序直接影响结果，

所以这种标记在区间修改的时候必须下推旧标记，不然会出错。

注意，有多个标记的时候，标记下推的顺序也很重要，错误的下推顺序可能会导致错误。

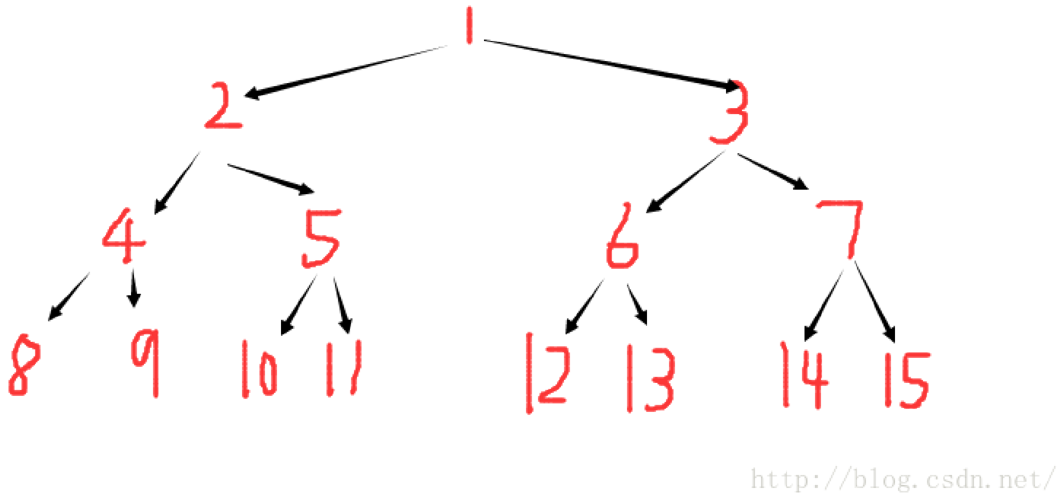
之所以要区分两种标记，是因为**非递归线段树**只能维护相对标记。

因为非递归线段树是自底向上直接修改分成的每个子区间，所以根本做不到在区间修改的时候下推标记。

非递归线段树一般不下推标记，而是自下而上求答案的过程中，根据标记更新答案。

(4)线段树的存储结构：

线段树是用数组来模拟树形结构，对于每一个节点R ,左子节点为 2*R (一般写作R<<1)右子节点为 2*R+1（一般写作R<<1|1）
然后以1为根节点，所以，整体的统计信息是存在节点1中的。
这么表示的原因看下图就很明白了，左子树的节点标号都是根节点的两倍，右子树的节点标号都是左子树+1：



线段树需要的数组元素个数是： $2^{\lceil \log_2(n) \rceil + 1}$,一般都开4倍空间，比如： int A[n<<2];

三：递归实现

以下以维护数列区间和的线段树为例，演示最基本的线段树代码。

(0)定义：

```cpp
#define maxn 100007  //元素总个数
#define ls l,m,rt<<1
#define rs m+1,r,rt<<1|1
int Sum[maxn<<2],Add[maxn<<2];//Sum求和，Add为懒惰标记
int A[maxn],n;//存原数组数据下标[1,n]
```


```cpp
#define maxn 100007  //元素总个数
#define ls l,m,rt<<1
#define rs m+1,r,rt<<1|1
int Sum[maxn<<2],Add[maxn<<2];//Sum求和，Add为懒惰标记
int A[maxn],n;//存原数组数据下标[1,n]
```


(1)建树：

```cpp
//PushUp函数更新节点信息 ，这里是求和
void PushUp(int rt){Sum[rt]=Sum[rt<<1]+Sum[rt<<1|1];}
//Build函数建树
void Build(int l,int r,int rt){ //l,r表示当前节点区间，rt表示当前节点编号
    if(l==r) {//若到达叶节点
        Sum[rt]=A[l];//储存数组值
        return;
    }
    int m=(l+r)>>1;
    //左右递归
    Build(l,m,rt<<1);
    Build(m+1,r,rt<<1|1);
    //更新信息
    PushUp(rt);
```

```
}
```

[cpp]

```cpp
//PushUp函数更新节点信息 ，这里是求和
void PushUp(int rt){Sum[rt]=Sum[rt<<1]+Sum[rt<<1|1];}
//Build函数建树
void Build(int l,int r,int rt){ //l,r表示当前节点区间，rt表示当前节点编号
    if(l==r) {//若到达叶节点
        Sum[rt]=A[l];//储存数组值
        return;
    }
    int m=(l+r)>>1;
    //左右递归
    Build(l,m,rt<<1);
    Build(m+1,r,rt<<1|1);
    //更新信息
    PushUp(rt);
}
```

(2)点修改:

假设A[L]+=C:

[cpp]

```cpp
void Update(int L,int C,int l,int r,int rt){//l,r表示当前节点区间，rt表示当前节点编号
    if(l==r){//到叶节点，修改
        Sum[rt]+=C;
        return;
    }
    int m=(l+r)>>1;
    //根据条件判断往左子树调用还是往右
    if(L <= m) Update(L,C,l,m,rt<<1);
    else       Update(L,C,m+1,r,rt<<1|1);
    PushUp(rt);//子节点更新了，所以本节点也需要更新信息
}
```

[cpp]

```cpp
void Update(int L,int C,int l,int r,int rt){//l,r表示当前节点区间，rt表示当前节点编号
    if(l==r){//到叶节点，修改
        Sum[rt]+=C;
        return;
    }
    int m=(l+r)>>1;
    //根据条件判断往左子树调用还是往右
    if(L <= m) Update(L,C,l,m,rt<<1);
    else       Update(L,C,m+1,r,rt<<1|1);
    PushUp(rt);//子节点更新了，所以本节点也需要更新信息
}
```


(3)区间修改:

假设A[L,R]+=C

[cpp]

```cpp
void Update(int L,int R,int C,int l,int r,int rt){//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if(L <= l && r <= R){//如果本区间完全在操作区间[L,R]以内
        Sum[rt]+=C*(r-l+1);//更新数字和，向上保持正确
        Add[rt]+=C;//增加Add标记，表示本区间的Sum正确，子区间的Sum仍需要根据Add的值来调整
        return ;
    }
    int m=(l+r)>>1;
    PushDown(rt,m-l+1,r-m);//下推标记
    //这里判断左右子树跟[L,R]有无交集，有交集才递归
```

```
    if(L <= m) Update(L,R,C,l,m,rt<<1);
    if(R >  m) Update(L,R,C,m+1,r,rt<<1|1);
    PushUp(rt);//更新本节点信息
}
```

```cpp
void Update(int L,int R,int C,int l,int r,int rt){//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if(L <= l && r <= R){//如果本区间完全在操作区间[L,R]以内
        Sum[rt]+=C*(r-l+1);//更新数字和，向上保持正确
        Add[rt]+=C;//增加Add标记，表示本区间的Sum正确，子区间的Sum仍需要根据Add的值来调整
        return ;
    }
    int m=(l+r)>>1;
    PushDown(rt,m-l+1,r-m);//下推标记
    //这里判断左右子树跟[L,R]有无交集，有交集才递归
    if(L <= m) Update(L,R,C,l,m,rt<<1);
    if(R >  m) Update(L,R,C,m+1,r,rt<<1|1);
    PushUp(rt);//更新本节点信息
}
```

(4)区间查询：

询问A[L,R]的和

首先是下推标记的函数：

```cpp
void PushDown(int rt,int ln,int rn){
    //ln,rn为左子树，右子树的数字数量。
    if(Add[rt]){
        //下推标记
        Add[rt<<1]+=Add[rt];
        Add[rt<<1|1]+=Add[rt];
        //修改子节点的Sum使之与对应的Add相对应
        Sum[rt<<1]+=Add[rt]*ln;
        Sum[rt<<1|1]+=Add[rt]*rn;
        //清除本节点标记
        Add[rt]=0;
    }
}
```


```cpp
void PushDown(int rt,int ln,int rn){
    //ln,rn为左子树，右子树的数字数量。
    if(Add[rt]){
        //下推标记
        Add[rt<<1]+=Add[rt];
        Add[rt<<1|1]+=Add[rt];
        //修改子节点的Sum使之与对应的Add相对应
        Sum[rt<<1]+=Add[rt]*ln;
        Sum[rt<<1|1]+=Add[rt]*rn;
        //清除本节点标记
        Add[rt]=0;
    }
}
```


然后是区间查询的函数：

```cpp
int Query(int L,int R,int l,int r,int rt){//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if(L <= l && r <= R){
        //在区间内，直接返回
        return Sum[rt];
    }
    int m=(l+r)>>1;
```

```cpp
    //下推标记，否则Sum可能不正确
    PushDown(rt,m-l+1,r-m);

    //累计答案
    int ANS=0;
    if(L <= m) ANS+=Query(L,R,l,m,rt<<1);
    if(R >  m) ANS+=Query(L,R,m+1,r,rt<<1|1);
    return ANS;
}
```

[cpp]

```cpp
int Query(int L,int R,int l,int r,int rt){//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if(L <= l && r <= R){
        //在区间内，直接返回
        return Sum[rt];
    }
    int m=(l+r)>>1;
    //下推标记，否则Sum可能不正确
    PushDown(rt,m-l+1,r-m);

    //累计答案
    int ANS=0;
    if(L <= m) ANS+=Query(L,R,l,m,rt<<1);
    if(R >  m) ANS+=Query(L,R,m+1,r,rt<<1|1);
    return ANS;
}
```

(5)函数调用:

[cpp]

```cpp
//建树
Build(1,n,1);
//点修改
Update(L,C,1,n,1);
//区间修改
Update(L,R,C,1,n,1);
//区间查询
int ANS=Query(L,R,1,n,1);
```

[cpp]

```cpp
//建树
Build(1,n,1);
//点修改
Update(L,C,1,n,1);
//区间修改
Update(L,R,C,1,n,1);
//区间查询
int ANS=Query(L,R,1,n,1);
```

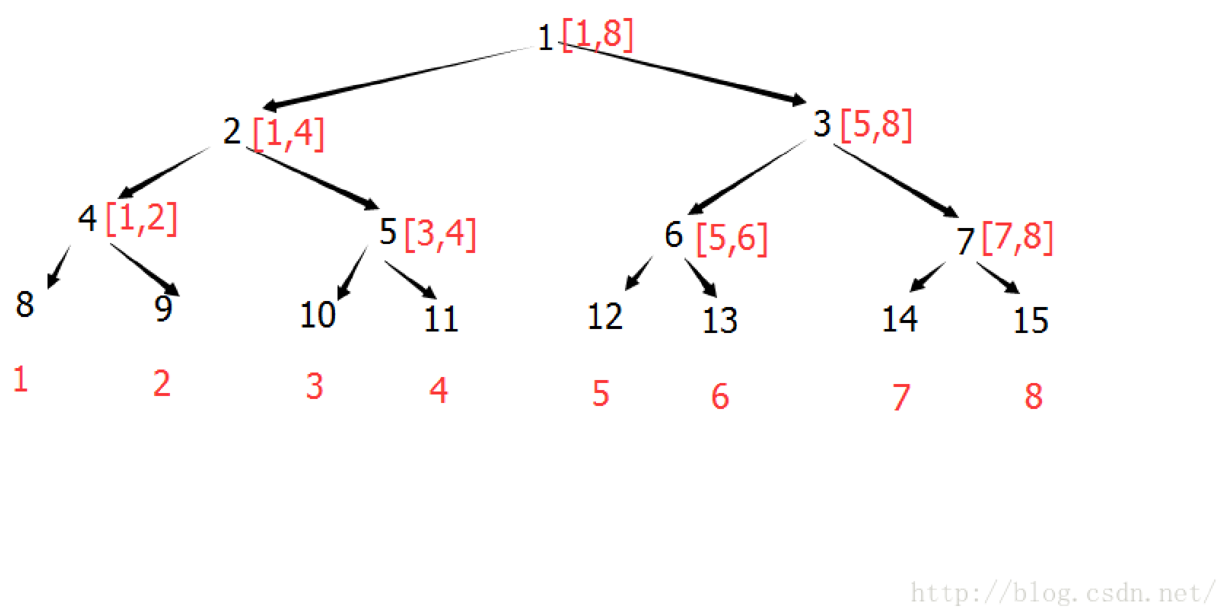

感谢几位网友指出了我的错误。
我说相对标记在Update时可以不下推，这一点是对的，但是原来的代码是错误的。
因为原来的代码中，PushUP函数是没有考虑本节点的Add值的，如果Update时下推标记，那么PushUp的时候，节点的Add值一定为零，所以不需要考虑Add。
但是，如果Update时暂时不下推标记的话，那么PushUp函数就必须考虑本节点的Add值，否则会导致错误。
为了简便，上面函数中，PushUp函数没有考虑Add标记。所以无论是相对标记还是绝对标记，在更新信息的时候，
到达的每个节点都必须调用PushDown函数来下推标记，另外，代码中，点修改函数中没有PushDown函数，因为这里假设只有点修改一种操作，
如果题目中是点修改和区间修改混合的话，那么点修改中也需要PushDown。

四：非递归原理

非递归的思路很巧妙，思路以及部分代码实现 来自 清华大学 张昆玮 《统计的力量》 ，有兴趣可以去找来看。
非递归的实现，代码简单（尤其是点修改和区间查询），速度快，建树简单，遍历元素简单。总之能非递归就非递归吧。
不过，要支持区间修改的话，代码会变得复杂，所以区间修改的时候还是要取舍。有个特例，如果区间修改，但是只需要在所有操作结束之后，一次性下推所有标记，然后求结果，这样的话，非递归写起来也是很方便的。
下面先讲思路，再讲实现。

点修改：

非递归的思想总的来说就是自底向上进行各种操作。回忆递归线段树的点修改，首先由根节点1向下递归，找到对应的叶节点，然后，修改叶节点的值，再向上返回，在函数返回的过程中，更新路径上的节点的统计信息。而非递归线段树的思路是，如果可以直接找到叶节点，那么就可以直接从叶节点向上更新，而一个节点找父节点是很容易的，编号除以2再下取整就行了。
那么，如何可以直接找到叶节点呢？非递归线段树将普通线段树(假设元素数量为n)扩充成一颗满二叉树。
来观察一下满二叉树的性质：



可以注意到红色和黑色数字的差是固定的，如果事先算出这个差值，就可以直接找到叶节点。

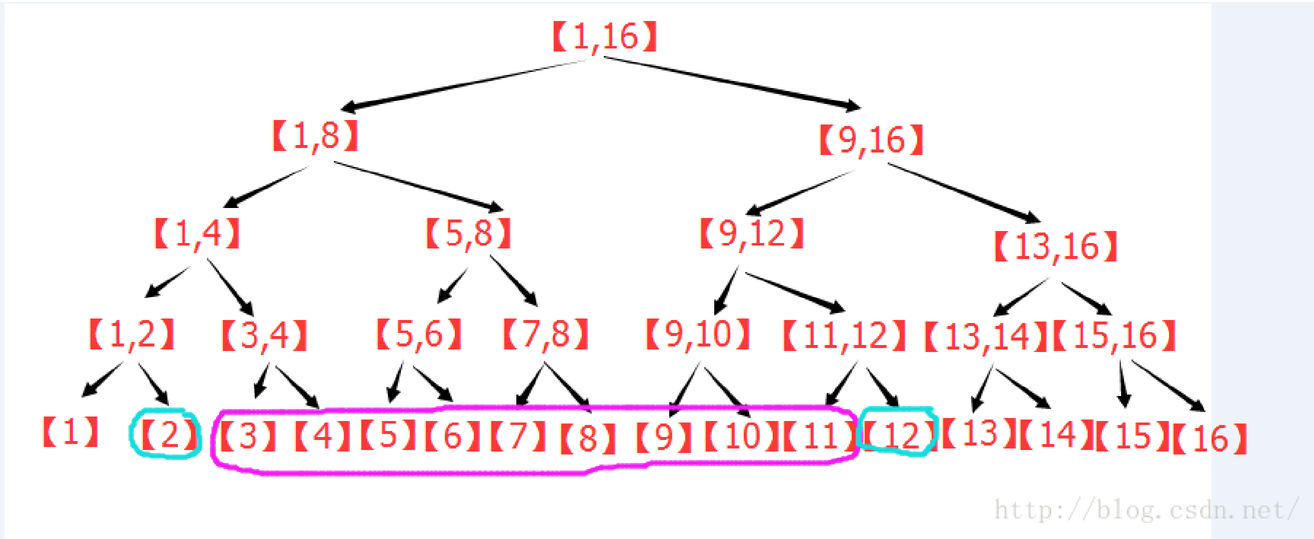
注意：区分3个概念：原数组下标，线段树中的下标和存储下标。
原数组下标，是指，需要维护统计信息（比如区间求和）的数组的下标，这里都默认下标从1开始（一般用A数组表示）
线段树下标，是指，加入线段树中某个位置的下标，比如，原数组中的第一个数，一般会加入到线段树中的第二个位置，为什么要这么做，后面会讲。
存储下标，是指该元素所在的叶节点的编号，即实际存储的位置。

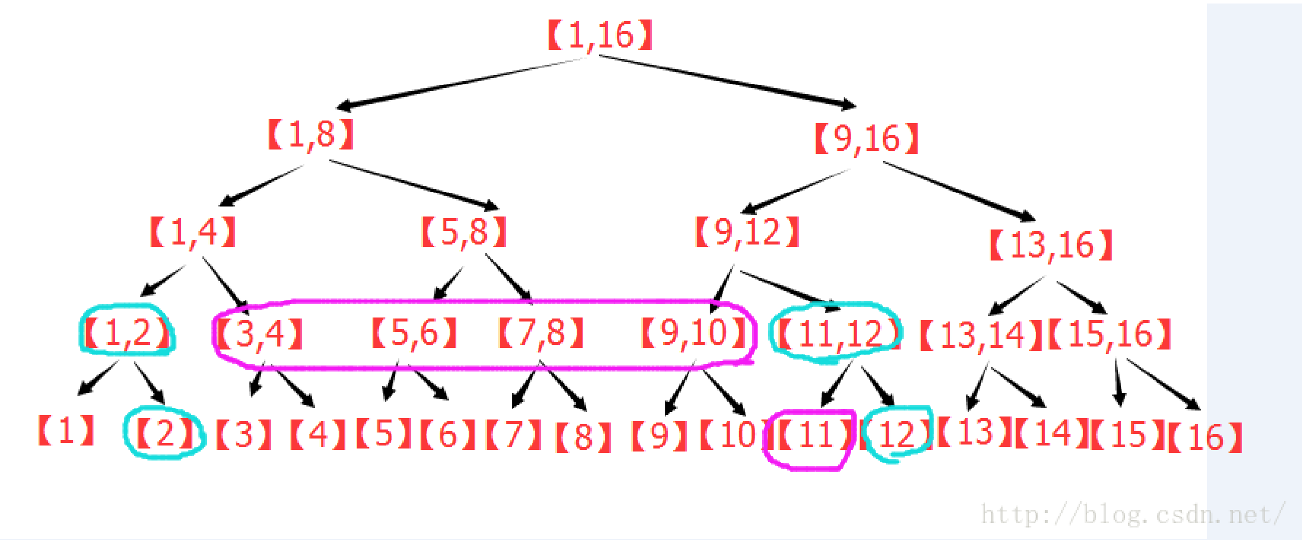
【在上面的图片中，红色为原数组下标，黑色为存储下标】

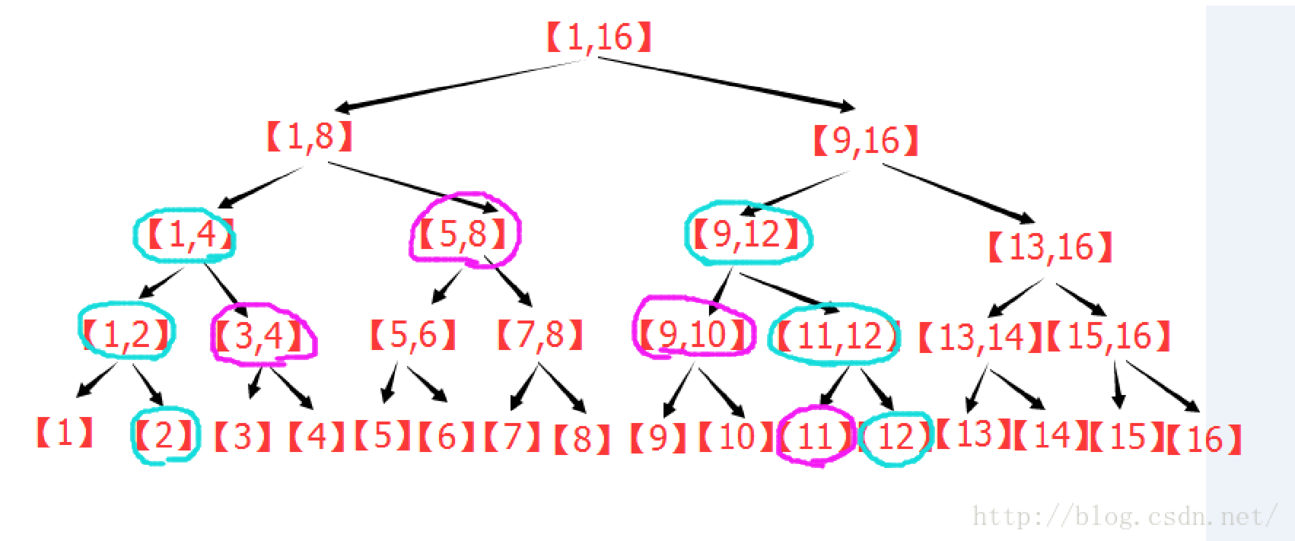
有了这3个概念，下面开始讲区间查询。

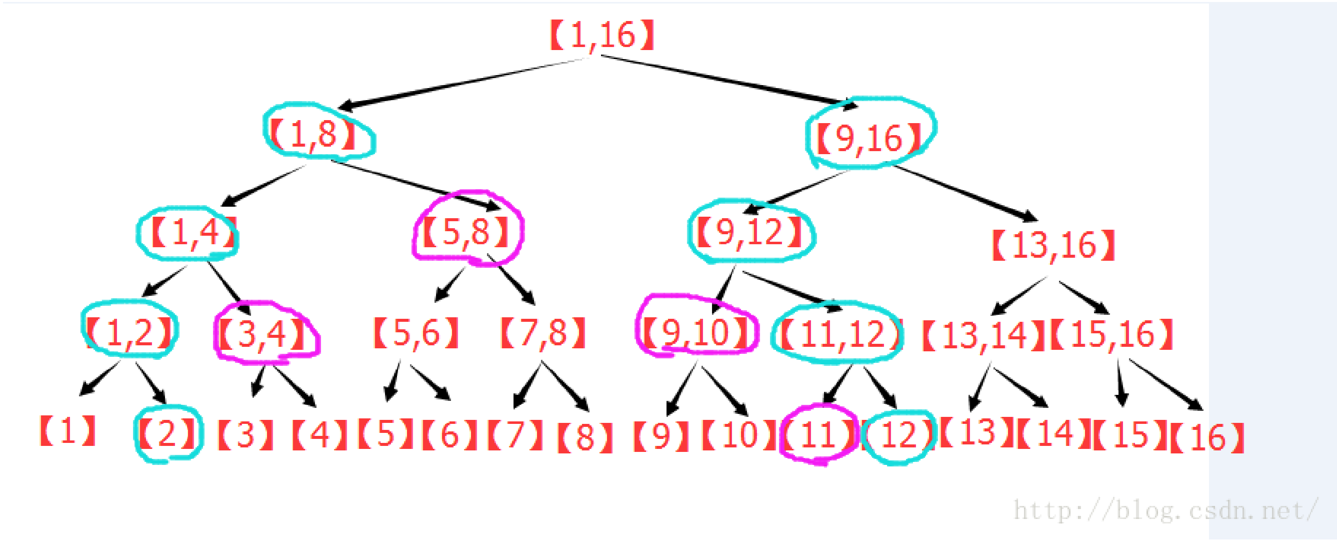
点修改下的区间查询：

首先，区间的划分没有变，现在关键是如何直接找到被分成的区间。原来是递归查找，判断左右子区间跟[L,R]是否有交点，若有交点则向下递归。现在要非递归实现，这就是巧妙之处，见下图，以查询[3,11]为例子。


【1,16】
【1,8】 【9,16】
【1,4】 【5,8】 【9,12】 【13,16】
【1,2】 【3,4】 【5,6】 【7,8】 【9,10】 【11,12】 【13,14】 【15,16】
【1】 【2】 【3】 【4】 【5】 【6】 【7】 【8】 【9】 【10】 【11】 【12】 【13】 【14】 【15】 【16】
http://blog.csdn.net/

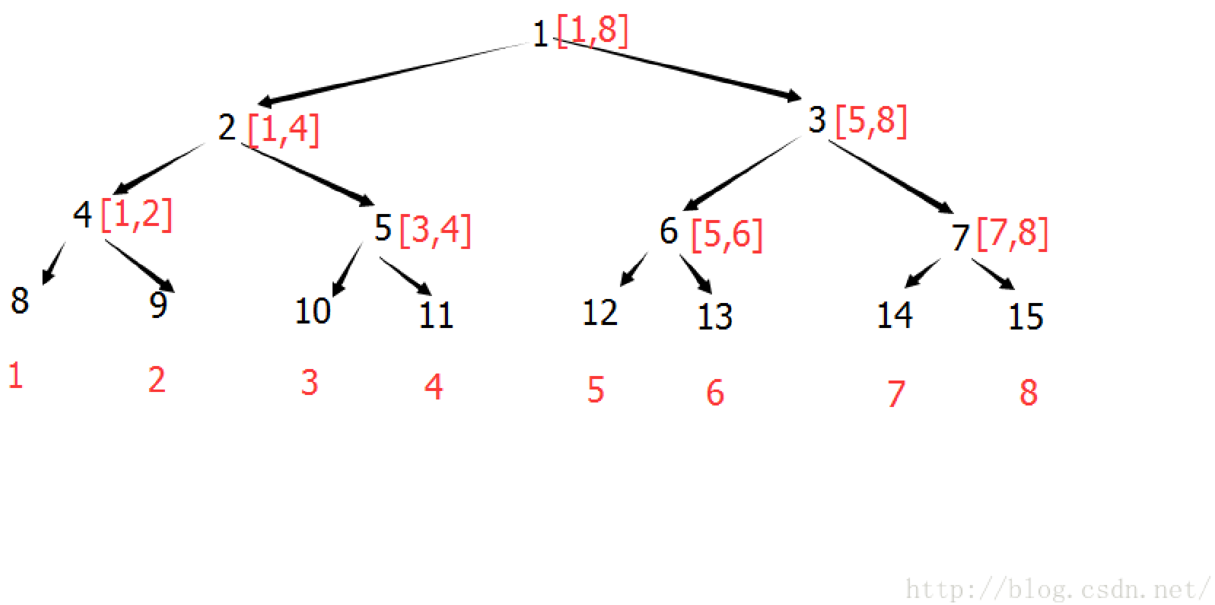

【1,16】
【1,8】 【9,16】
【1,4】 【5,8】 【9,12】 【13,16】
【1,2】 【3,4】 【5,6】 【7,8】 【9,10】 【11,12】 【13,14】 【15,16】
【1】 【2】 【3】 【4】 【5】 【6】 【7】 【8】 【9】 【10】 【11】 【12】 【13】 【14】 【15】 【16】
http://blog.csdn.net/


【1,16】
【1,8】 【9,16】
【1,4】 【5,8】 【9,12】 【13,16】
【1,2】 【3,4】 【5,6】 【7,8】 【9,10】 【11,12】 【13,14】 【15,16】
【1】 【2】 【3】 【4】 【5】 【6】 【7】 【8】 【9】 【10】 【11】 【12】 【13】 【14】 【15】 【16】
http://blog.csdn.net/



其实，容易发现，紫色部分的变化，跟原来分析线段树的区间分解的时候是一样的规则，图中多的蓝色是什么意思呢?
首先注意到，蓝色节点刚好在紫色节点的两端。
回忆一下，原来线段树在区间逐层被替代的过程中，哪些节点被留了下来？最左侧的节点，若为其父节点的右子节点，则留下。
最右侧的节点，若为其父节点的左子节点则留下。那么对于包裹着紫色的蓝色节点来看，刚好相反。
比如，以左侧的的蓝色为例，若该节点是其父节点的右子节点，就证明它右侧的那个紫色节点不会留下，会被其父替代，所以没必要在这一步计算，若该节点是其父节点的左子节点，就证明它右侧的那个紫色节点会留在这一层，所以必须在此刻计算，否则以后都不会再计算这个节点了。这样逐层上去，容易发现，对于左侧的蓝色节点来说，只要它是左子节点，那么就要计算对应的右子节点。同理，对于右侧的蓝色节点，只要它是右子节点，就需要计算它对应的左子节点。这个计算一直持续到左右蓝色节点的父亲为同一个的时候，才停止。于是，区间查询，其实就是两个蓝色节点一路向上走，在路径上更新答案。
这样，区间修改就变成了两条同时向根走的链，明显复杂度O(log2(n))。并且可以非递归实现。
至此，区间查询也解决了，可以直接找到所有分解成的区间。
但是有一个问题，如果要查询[1,5]怎么办？[1]左边可是没地方可以放置蓝色节点了。
问题的解决办法简单粗暴，原数组的1到n就不存在线段树的1到n了，而是存在线段树的2到n+1，
而开始要建立一颗有n+2个元素的树，空出第一个和最后一个元素的空间。

现在来讲如何扩充成一颗满二叉树。



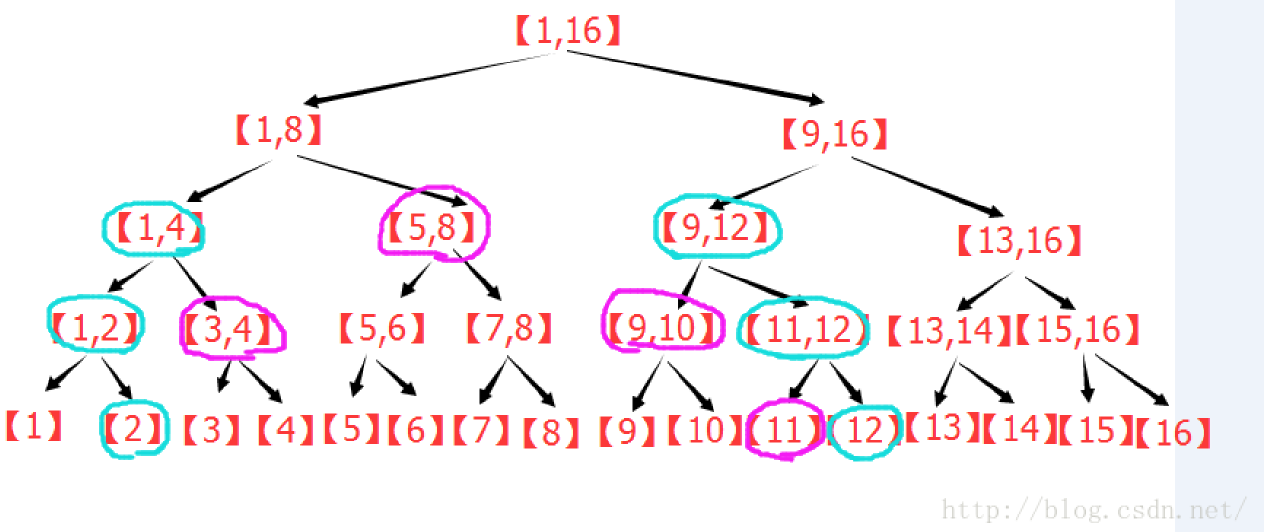
再来看这课满二叉树，令N=8;注意到，该树可以存8个元素，并且[1..7]是非叶节点，[8..15]是叶节点。
也就是说，左下角为N的满二叉树，可以存N个元素，并且[1..N-1]是非叶节点,[N..2N-1]是叶节点。
并且，**线段树下标+N-1=存储下标** （还记不记得原来对三个下标的定义）

这时，这个线段树存在两段坐标映射：
原数组下标+1=线段树下标
线段树下标+N-1=存储下标

联立方程得到：**原数组下标+N=存储下标**
于是从原数组下标到存储下标的转换及其简单。

下一个问题：N怎么确定？
上面提到了，N的含义之一是，这棵树可以存N个元素，也就是说N必须大于等于n+2
于是，N的定义，N是大于等于n+2的，某个2的次方。

区间修改下的区间查询：
方法之一：如果题目许可，可以直接打上标记，最后一次下推所有标记，然后就可以遍历叶节点来获取信息。
方法之二：如果题目查询跟修改混在一起，那么，采用**标记永久化**思想。也就是，不下推标记。
递归线段树是在查询区间的时候下推标记，使得到达每个子区间的时候，Sum已经是正确值。
非递归没法这么做，非递归是从下往上，遇到标记就更新答案。
这题是Add标记，一个区间Add标记表示这个区间所有元素都需要增加Add
Add含义不变，Add仍然表示本节点的Sum已经更新完毕，但是子节点的Sum仍需要更新.
现在就是如何在**查询**的时候根据标记更新答案。
观察下图：



左边的蓝色节点从下往上走，在蓝色节点到达[1,4]时，注意到，左边蓝色节点之前计算过的所有节点（即[3,4]）都是目前蓝色节点的子节点也就是说，当前蓝色节点的Add是要影响这个节点已经计算过的所有数。多用一个变量来记录这个蓝色节点已经计算过多少个数，根据个数以及当前蓝色节点的Add，来更新最终答案。
更新完答案之后，再加上[5,8]的答案，同时当前蓝色节点计算过的个数要+4(因为[5,8]里有4个数)
然后当这个节点到达[1,8]节点时，可以更新[1,8]的Add.
这里，本来左右蓝色节点相遇之后就不再需要计算了，但是由于有了Add标记，左右蓝色节点的公共祖先上的Add标记会影响目前的所有数，所以还需要一路向上查询到根，沿路根据Add更新答案。

区间修改：
这里讲完了查询，再来讲讲**修改**，
修改的时候，给某个区间的Add加上了C，这个区间的子区间向上查询时，会经过这个节点，也就是会计算这个Add,但是
如果路径经过这个区间的父节点，就不会计算这个节点的Add,也就会出错。这里其实跟递归线段树一样，改了某个区间的Add
仍需要向上更新所有包含这个区间的Sum，来保持上面所有节点的正确性。

五：非递归实现

以下以维护数列区间和的线段树为例，演示最基本的非递归线段树代码。
(0)定义：

```cpp
//
#define maxn 100007
int A[maxn],n,N;//原数组,n为原数组元素个数 ,N为扩充元素个数
int Sum[maxn<<2];//区间和
int Add[maxn<<2];//懒惰标记
```

[cpp]

```cpp
//
#define maxn 100007
int A[maxn],n,N;//原数组,n为原数组元素个数 ,N为扩充元素个数
int Sum[maxn<<2];//区间和
int Add[maxn<<2];//懒惰标记
```

(1)建树:

[cpp]

```cpp
//
void Build(int n){
    //计算N的值
    N=1;while(N < n+2) N <<= 1;
    //更新叶节点
    for(int i=1;i<=n;++i) Sum[N+i]=A[i];//原数组下标+N=存储下标
    //更新非叶节点
    for(int i=N-1;i>0;--i){
        //更新所有非叶节点的统计信息
        Sum[i]=Sum[i<<1]+Sum[i<<1|1];
        //清空所有非叶节点的Add标记
        Add[i]=0;
    }
}
```


[cpp]

```cpp
//
void Build(int n){
    //计算N的值
    N=1;while(N < n+2) N <<= 1;
    //更新叶节点
    for(int i=1;i<=n;++i) Sum[N+i]=A[i];//原数组下标+N=存储下标
    //更新非叶节点
    for(int i=N-1;i>0;--i){
        //更新所有非叶节点的统计信息
        Sum[i]=Sum[i<<1]+Sum[i<<1|1];
        //清空所有非叶节点的Add标记
        Add[i]=0;
    }
}
```


(2)点修改:

A[L]+=C

[cpp]

```cpp
//
void Update(int L,int C){
    for(int s=N+L;s;s>>=1){
        Sum[s]+=C;
    }
}
```


[cpp]

```cpp
//
void Update(int L,int C){
    for(int s=N+L;s;s>>=1){
        Sum[s]+=C;
    }
}
```

(3)点修改下的区间查询:

求A[L..R]的和（点修改没有使用Add所以不需要考虑）

代码非常简洁，也不难理解，

s和t分别代表之前的论述中的左右蓝色节点，其余的代码根据之前的论述应该很容易看懂了。

s^t^1 在s和t的父亲相同时值为0，终止循环。

两个if是判断s和t分别是左子节点还是右子节点，根据需要来计算Sum

```cpp
[cpp]

//
int Query(int L,int R){
    int ANS=0;
    for(int s=N+L-1,t=N+R+1;s^t^1;s>>=1,t>>=1){
        if(~s&1) ANS+=Sum[s^1];
        if( t&1) ANS+=Sum[t^1];
    }
    return ANS;
}
```


```cpp
[cpp]

//
int Query(int L,int R){
    int ANS=0;
    for(int s=N+L-1,t=N+R+1;s^t^1;s>>=1,t>>=1){
        if(~s&1) ANS+=Sum[s^1];
        if( t&1) ANS+=Sum[t^1];
    }
    return ANS;
}
```


(4)区间修改:

A[L..R]+=C

```cpp
[cpp]

<span style="font-size:14px;">//
void Update(int L,int R,int C){
    int s,t,Ln=0,Rn=0,x=1;
    //Ln:  s一路走来已经包含了几个数
    //Rn:  t一路走来已经包含了几个数
    //x:   本层每个节点包含几个数
    for(s=N+L-1,t=N+R+1;s^t^1;s>>=1,t>>=1,x<<=1){
        //更新Sum
        Sum[s]+=C*Ln;
        Sum[t]+=C*Rn;
        //处理Add
        if(~s&1) Add[s^1]+=C,Sum[s^1]+=C*x,Ln+=x;
        if( t&1) Add[t^1]+=C,Sum[t^1]+=C*x,Rn+=x;
    }
    //更新上层Sum
    for(;s;s>>=1,t>>=1){
        Sum[s]+=C*Ln;
        Sum[t]+=C*Rn;
    }
} </span>
```


```cpp
[cpp]

<span style="font-size:14px;">//
void Update(int L,int R,int C){
    int s,t,Ln=0,Rn=0,x=1;
    //Ln:  s一路走来已经包含了几个数
    //Rn:  t一路走来已经包含了几个数
    //x:   本层每个节点包含几个数
    for(s=N+L-1,t=N+R+1;s^t^1;s>>=1,t>>=1,x<<=1){
        //更新Sum
```

```
            Sum[s]+=C*Ln;
            Sum[t]+=C*Rn;
            //处理Add
            if(~s&1) Add[s^1]+=C,Sum[s^1]+=C*x,Ln+=x;
            if( t&1) Add[t^1]+=C,Sum[t^1]+=C*x,Rn+=x;
        }
        //更新上层Sum
        for(;s;s>>=1,t>>=1){
            Sum[s]+=C*Ln;
            Sum[t]+=C*Rn;
        }
    } </span>
```

(5)区间修改下的区间查询:

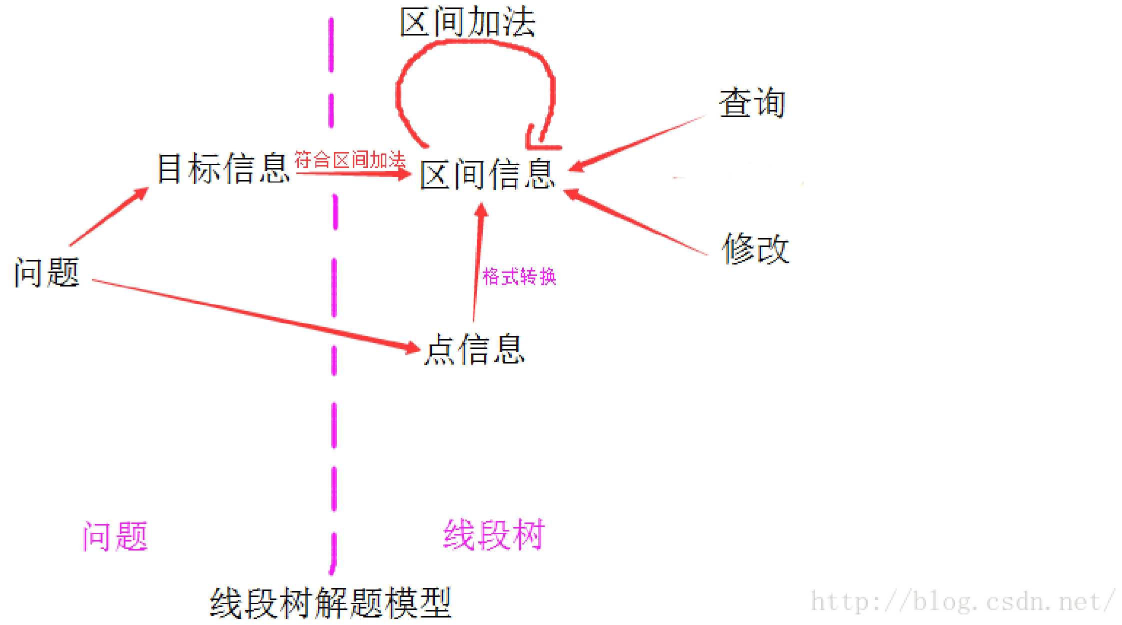
求A[L..R]的和

```cpp
//
int Query(int L,int R){
    int s,t,Ln=0,Rn=0,x=1;
    int ANS=0;
    for(s=N+L-1,t=N+R+1;s^t^1;s>>=1,t>>=1,x<<=1){
        //根据标记更新
        if(Add[s]) ANS+=Add[s]*Ln;
        if(Add[t]) ANS+=Add[t]*Rn;
        //常规求和
        if(~s&1) ANS+=Sum[s^1],Ln+=x;
        if( t&1) ANS+=Sum[t^1],Rn+=x;
    }
    //处理上层标记
    for(;s;s>>=1,t>>=1){
        ANS+=Add[s]*Ln;
        ANS+=Add[t]*Rn;
    }
    return ANS;
}
```


```cpp
//
int Query(int L,int R){
    int s,t,Ln=0,Rn=0,x=1;
    int ANS=0;
    for(s=N+L-1,t=N+R+1;s^t^1;s>>=1,t>>=1,x<<=1){
        //根据标记更新
        if(Add[s]) ANS+=Add[s]*Ln;
        if(Add[t]) ANS+=Add[t]*Rn;
        //常规求和
        if(~s&1) ANS+=Sum[s^1],Ln+=x;
        if( t&1) ANS+=Sum[t^1],Rn+=x;
    }
    //处理上层标记
    for(;s;s>>=1,t>>=1){
        ANS+=Add[s]*Ln;
        ANS+=Add[t]*Rn;
    }
    return ANS;
}
```


六：线段树解题模型

给出线段树解题模型以及一些例题。



先对图中各个名字给出定义：
问题：可能可以用线段树解决的问题
目标信息：由问题转换而成的，为了解决问题而需要统计的信息（可能不满足区间加法）。
点信息：每个点储存的信息
区间信息：每个区间维护的信息（线段树节点定义） （必须满足区间加法）
区间信息包括 统计信息和标记
--------统计信息：统计节点代表的区间的信息，一般自下而上更新
--------标记：对操作进行标记（在区间修改时需要），一般自上而下传递，或者不传递
区间加法：实现区间加法的代码
查询：实现查询操作的代码
修改：实现修改操作的代码

图中紫线右边是实际线段树的实现，左边是对问题的分析以及转换。

一个问题，若能转换成对一些连续点的修改或者统计，就可以考虑用线段树解决。
首先确定目标信息和点信息，然后将目标信息转换成区间信息（必要时，增加信息，使之符合区间加法）。
之后就是线段树的代码实现了，包括：
1.区间加法
2.建树，点信息到区间信息的转换
3.每种操作（包括查询，修改）对区间信息的调用，修改

这样，点的信息不同，区间信息不同，线段树可以维护很多种类的信息，所以是一种非常实用的数据结构。
可以解决很多问题，下面给出几个例子来说明。

（1）：字符串哈希

题目：URAL1989 Subpalindromes 题解
给定一个字符串(长度<=100000)，有两个操作。 1：改变某个字符。 2：判断某个子串是否构成回文串。
直接判断会超时。这个题目，是用线段树维护字符串哈希
对于一个字符串a[0],a[1],...,a[n-1] 它对应的哈希函数为a[0]+a[1]*K + a[2]*K^2 +...+a[n-1]*K^(n-1)
再维护一个从右往左的哈希值：a[0]*K^(n-1) + a[1]*K^(n-2) +...+a[n-1]
若是回文串，则左右的哈希值会相等。而左右哈希值相等，则很大可能这是回文串。
若出现误判，可以再用一个K2，进行二次哈希判断，可以减小误判概率。
实现上，哈希值最好对某个质数取余数，这样分布更均匀。

解题模型：
问题经过转换之后：
目标信息：某个区间的左，右哈希值
点信息：一个字符
目标信息已经符合区间加法，所以区间信息=目标信息。
所以线段树的结构为：
区间信息：区间哈希值
点信息：一个字符
代码主要需要注意2个部分：

1.区间加法：（PushUp函数,Pow[a]=K^a）
2.点信息->区间信息：(叶节点上，区间只包含一个点，所以需要将点信息转换成区间信息)
修改以及查询，在有了区间加法的情况下，没什么难度了。

可以看出，上述解题过程的核心，就是找到区间信息， 写好区间加法。
下面是维护区间和的部分，下面的代码没有取余，也就是实际上是对2^32取余数，这样其实分布不均匀，容易出现误判：

[cpp]

```cpp
//
#define K 137
#define maxn 100001
char str[maxn];
int Pow[maxn];//K的各个次方
struct Node{
    int KeyL,KeyR;
    Node():KeyL(0),KeyR(0){}
    void init(){KeyL=KeyR=0;}
}node[maxn<<2];
void PushUp(int L,int R,int rt){
    node[rt].KeyL=node[rt<<1].KeyL+node[rt<<1|1].KeyL*Pow[L];
    node[rt].KeyR=node[rt<<1].KeyR*Pow[R]+node[rt<<1|1].KeyR;
}
```

[cpp]

```cpp
//
#define K 137
#define maxn 100001
char str[maxn];
int Pow[maxn];//K的各个次方
struct Node{
    int KeyL,KeyR;
    Node():KeyL(0),KeyR(0){}
    void init(){KeyL=KeyR=0;}
}node[maxn<<2];
void PushUp(int L,int R,int rt){
    node[rt].KeyL=node[rt<<1].KeyL+node[rt<<1|1].KeyL*Pow[L];
    node[rt].KeyR=node[rt<<1].KeyR*Pow[R]+node[rt<<1|1].KeyR;
}
```


（2）：最长连续零

题目：Codeforces 527C Glass Carving　题解
题意是给定一个矩形，不停地纵向或横向切割，问每次切割后，最大的矩形面积是多少。
最大矩形面积=最长的长*最宽的宽
这题，长宽都是10^5，所以，用01序列表示每个点是否被切割，然后，
最长的长就是长的最长连续0的数量+1
最长的宽就是宽的最长连续0的数量+1
于是用线段树维护最长连续零

问题转换成：
目标信息：区间最长连续零的个数
点信息：0 或 1
由于目标信息不符合区间加法，所以要扩充目标信息。

转换后的线段树结构：
区间信息：从左，右开始的最长连续零，本区间是否全零，本区间最长连续零。
点信息：0 或 1
然后还是那2个问题：

1.区间加法：
这里，一个区间的最长连续零，需要考虑3部分：
-（1）：左子区间最长连续零
-（2）：右子区间最长连续零
-（3）：左右子区间拼起来，而在中间生成的连续零（可能长于两个子区间的最长连续零）
而中间拼起来的部分长度，其实是左区间从右开始的最长连续零+右区间从左开始的最长连续零。

所以每个节点需要多两个量，来存从左右开始的最长连续零。
然而，左开始的最长连续零分两种情况，
--（1）：左区间不是全零，那么等于左区间的左最长连续零
--（2）：左区间全零，那么等于左区间0的个数加上右区间的左最长连续零
于是，需要知道左区间是否全零，于是再多加一个变量。
最终，通过维护4个值，达到了维护区间最长连续零的效果。

2.点信息->区间信息 ：
如果是0，那么 最长连续零=左最长连续零=右最长连续零=1 ，全零=true。
如果是1，那么 最长连续零=左最长连续零=右最长连续零=0， 全零=false。

至于修改和查询，有了区间加法之后，机械地写一下就好了。
由于这里其实只有对整个区间的查询，所以查询函数是不用写的，直接找根的统计信息就行了。

代码如下：

[cpp]

```cpp
//
#define maxn 200001
using namespace std;
int L[maxn<<2][2];//从左开始连续零个数
int R[maxn<<2][2];//从右
int Max[maxn<<2][2];//区间最大连续零
bool Pure[maxn<<2][2];//是否全零
int M[2];
void PushUp(int rt,int k){//更新rt节点的四个数据  k来选择两棵线段树
    Pure[rt][k]=Pure[rt<<1][k]&&Pure[rt<<1|1][k];
    Max[rt][k]=max(R[rt<<1][k]+L[rt<<1|1][k],max(Max[rt<<1][k],Max[rt<<1|1][k]));
    L[rt][k]=Pure[rt<<1][k]?L[rt<<1][k]+L[rt<<1|1][k]:L[rt<<1][k];
    R[rt][k]=Pure[rt<<1|1][k]?R[rt<<1|1][k]+R[rt<<1][k]:R[rt<<1|1][k];
}
```

[cpp]

```cpp
//
#define maxn 200001
using namespace std;
int L[maxn<<2][2];//从左开始连续零个数
int R[maxn<<2][2];//从右
int Max[maxn<<2][2];//区间最大连续零
bool Pure[maxn<<2][2];//是否全零
int M[2];
void PushUp(int rt,int k){//更新rt节点的四个数据  k来选择两棵线段树
    Pure[rt][k]=Pure[rt<<1][k]&&Pure[rt<<1|1][k];
    Max[rt][k]=max(R[rt<<1][k]+L[rt<<1|1][k],max(Max[rt<<1][k],Max[rt<<1|1][k]));
    L[rt][k]=Pure[rt<<1][k]?L[rt<<1][k]+L[rt<<1|1][k]:L[rt<<1][k];
    R[rt][k]=Pure[rt<<1|1][k]?R[rt<<1|1][k]+R[rt<<1][k]:R[rt<<1|1][k];
}
```


（3）：计数排序

题目：Codeforces 558E A Simple Task 题解
给定一个长度不超过10^5的字符串（小写英文字母），和不超过5000个操作。
每个操作 L R K 表示给区间[L,R]的字符串排序，K=1为升序，K=0为降序。
最后输出最终的字符串。

题目转换成：
目标信息：区间的计数排序结果
点信息：一个字符
这里，目标信息是符合区间加法的，但是为了支持区间操作，还是需要扩充信息。

转换后的线段树结构：
区间信息：区间的计数排序结果，排序标记，排序种类（升，降）
点信息：一个字符

代码中需要解决的四个问题（难点在于标记下推和区间修改）：

1.区间加法
对应的字符数量相加即可（注意标记是不上传的，所以区间加法不考虑标记）。
2.点信息->区间信息：把对应字符的数量设置成1，其余为0，排序标记为false。
3.标记下推
明显，排序标记是绝对标记，也就是说，标记对子节点是覆盖式的效果，一旦被打上标记，下层节点的一切信息都无效。
下推标记时，根据自己的排序结果，将元素分成对应的部分，分别装入两个子树。
4.区间修改
这个是难点，由于要对某个区间进行排序，首先对各个子区间求和（求和之前一定要下推标记，才能保证求的和是正确的）
由于使用的计数排序，所以求和之后，新顺序也就出来了。然后按照排序的顺序按照每个子区间的大小来分配字符。
操作后，每个子区间都被打上了标记。

最后，在所有操作结束之后，一次下推所有标记，就可以得到最终的字符序列。

这里只给出节点定义。

[cpp]

```
//
struct Node{
    int d[26];//计数排序
    int D;//总数
    bool sorted;//是否排好序
    bool Inc;//是否升序
};
```


[cpp]

```
//
struct Node{
    int d[26];//计数排序
    int D;//总数
    bool sorted;//是否排好序
    bool Inc;//是否升序
};
```


（4）总结：

总结一下，线段树解题步骤。

一：将问题转换成点信息和目标信息。
即，将问题转换成对一些点的信息的统计问题。

二：将目标信息根据需要扩充成区间信息
1.增加信息符合区间加法。
2.增加标记支持区间操作。

三：代码中的主要模块：
1.区间加法
2.标记下推
3.点信息->区间信息
4.操作（各种操作，包括修改和查询）

完成第一步之后，题目有了可以用线段树解决的可能。
完成第二步之后，题目可以由线段树解决。
第三步就是慢慢写代码了。

七：扫描线

线段树的一大应用是扫描线。

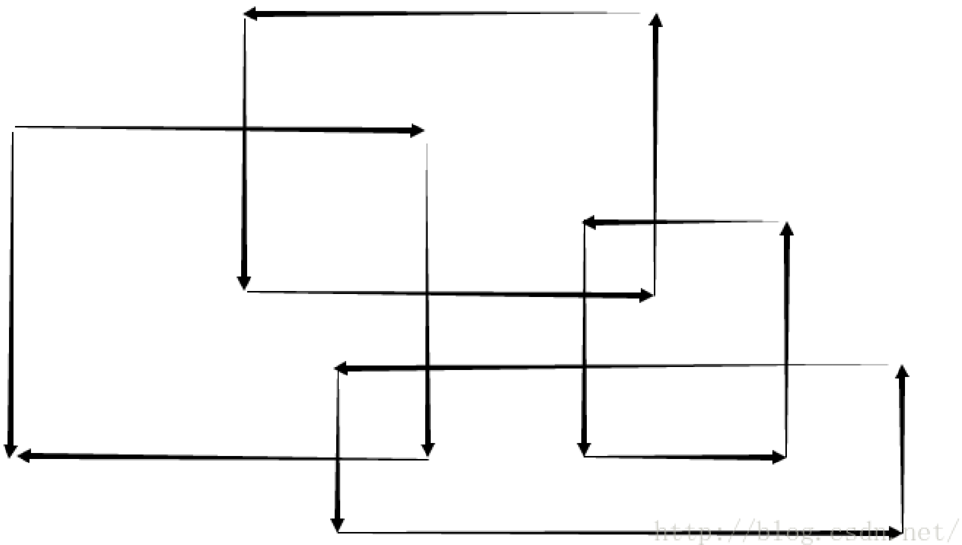
先把相关题目给出，有兴趣可以去找来练习：

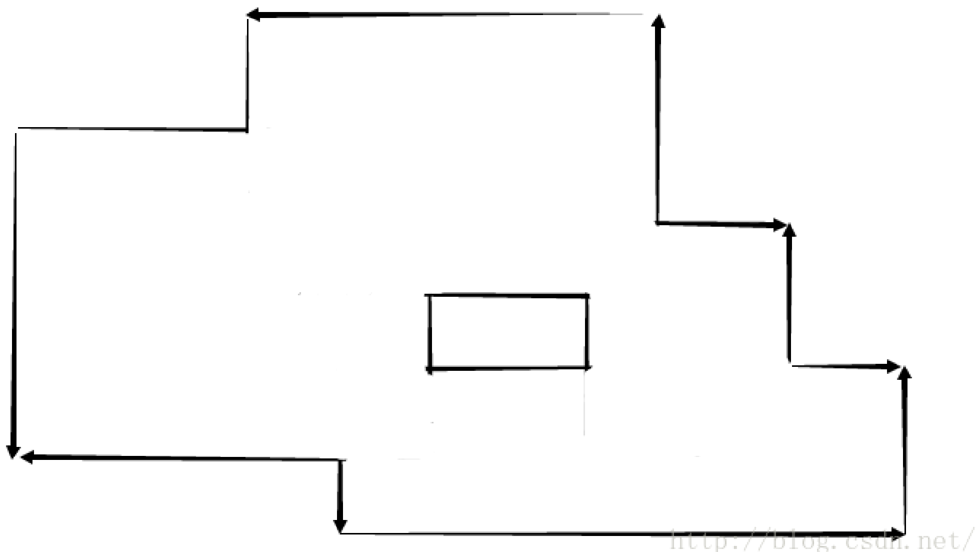
POJ 1177 Picture:给定若干矩形求合并之后的图形周长　　题解
HDU 1255 覆盖的面积：给定平面上若干矩形,求出被这些矩形覆盖过至少两次的区域的面积.　题解
HDU 3642 Get The Treasury：给定若干空间立方体，求重叠了3次或以上的体积（这个是扫描面，每个面再扫描线）题解

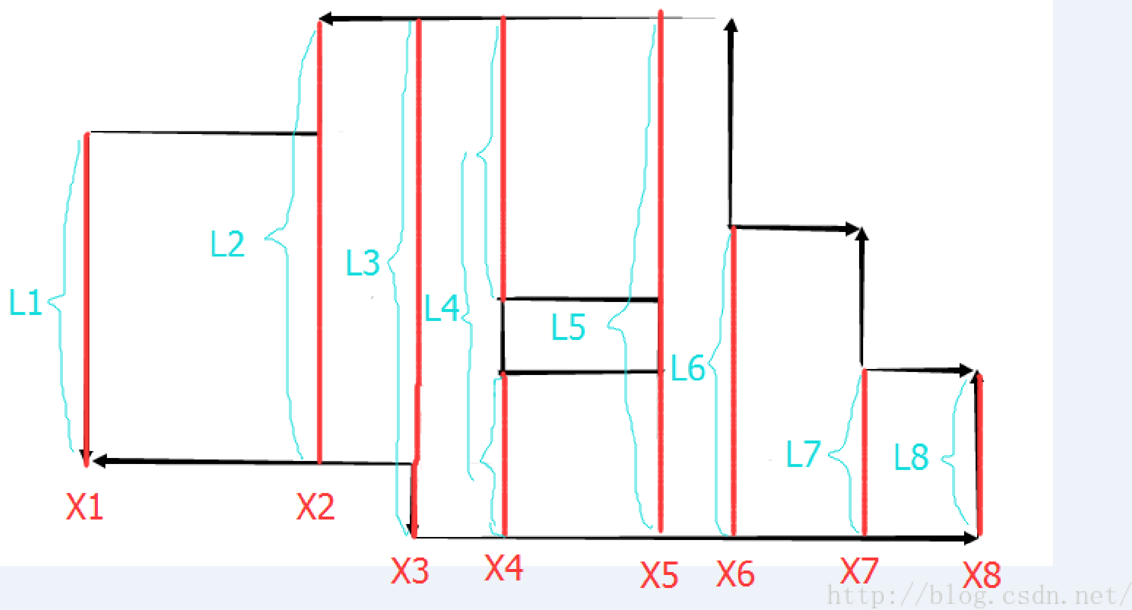
再补充一道稍微需要一点模型转换的扫描线题：
POJ 2482 Stars in your window ：给定一些星星的位置和亮度，求用W*H的矩形能够框住的星星亮度之和最大为多少。
这题是把星星转换成了矩形，把矩形框转换成了点，然后再扫描线。 题解

扫描线求重叠矩形面积：

考虑下图中的四个矩形：







观察第三个图：
扫描线的思路：使用一条垂直于X轴的直线，从左到右来扫描这个图形，明显，只有在碰到矩形的左边界或者右边界的时候，
这个线段所扫描到的情况才会改变，所以把所有矩形的入边，出边按X值排序。然后根据X值从小到大去处理，就可以
用线段树来维护扫描到的情况。如上图，X1到X8是所有矩形的入边，出边的X坐标。
而红色部分的线段，是这样，如果碰到矩形的入边，就把这条边加入，如果碰到出边，就拿走。红色部分就是有线段覆盖的部分。
要求面积，只需要知道图中的L1到L8。而线段树就是用来维护这个L1到L8的。
扫描线算法流程：

X1:首先遇到X1,将第一条线段加入线段树，由线段树统计得到线段长度为L1.

X2:然后继续扫描到X2,此时要进行两个动作：
1.计算面积，目前扫过的面积=L1*(X2-X1)
2.更新线段。由于X2处仍然是入边，所以往线段树中又加了一条线段，加的这条线段可以参考3幅图中的第一幅。
然后线段树自动得出此时覆盖的线段长度为L2 （注意两条线段有重叠部分，重叠部分的长度只能算一次）

X3:继续扫描到X3，步骤同X2
先计算 扫过的面积+=L2*(X3-X2)
再加入线段，得到L3.

X4:扫描到X4有些不一样了。
首先还是计算 扫过的面积+=L3*(X4-X3)
然后这时遇到了第一个矩形的出边，这时要从线段树中删除一条线段。
删除之后的结果是线段树中出现了2条线段，线段树自动维护这两条线段的长度之和L4

讲到这里算法流程应该很清晰了。
首先将所有矩形的入边，出边都存起来，然后根据X值排序。
这里用一个结构体，来存这些信息，然后排序。

[cpp]

```cpp
//
struct LINE{
    int x;//横坐标
    int y1,y2;//矩形纵向线段的左右端点
    bool In;//标记是入边还是出边
    bool operator < (const Line &B)const{return x < B.x;}
}Line[maxn];
```

[cpp]

```cpp
//
struct LINE{
    int x;//横坐标
    int y1,y2;//矩形纵向线段的左右端点
    bool In;//标记是入边还是出边
    bool operator < (const Line &B)const{return x < B.x;}
}Line[maxn];
```


然后扫描的时候，需要两个变量，一个叫PreL，存前一个x的操作结束之后的L值，和X，前一个横坐标。
假设一共有Ln条线段，线段下标从0开始，已经排好序。
那么算法大概是这样：

[cpp]

```cpp
//
int PreL=0;//前一个L值,刚开始是0，所以第一次计算时不会引入误差
int X;//X值
int ANS=0;//存累计面积
int I=0;//线段的下标

while(I < Ln){
    //先计算面积
    ANS+=PreL*(Line[I].x-X);
    X=Line[I].x;//更新X值
    //对所有X相同的线段进行操作
```

```cpp
    while(I < Ln && Line[I].x == X){
        //根据入边还是出边来选择加入线段还是移除线段
        if(Line[I].In) Cover(Line[I].y1,Line[I].y2-1,1,n,1);
        else         Uncover(Line[I].y1,Line[I].y2-1,1,n,1);
        ++I;
    }
}
```

[cpp]

```cpp
//
int PreL=0;//前一个L值,刚开始是0, 所以第一次计算时不会引入误差
int X;//X值
int ANS=0;//存累计面积
int I=0;//线段的下标

while(I < Ln){
    //先计算面积
    ANS+=PreL*(Line[I].x-X);
    X=Line[I].x;//更新X值
    //对所有X相同的线段进行操作
    while(I < Ln && Line[I].x == X){
        //根据入边还是出边来选择加入线段还是移除线段
        if(Line[I].In) Cover(Line[I].y1,Line[I].y2-1,1,n,1);
        else         Uncover(Line[I].y1,Line[I].y2-1,1,n,1);
        ++I;
    }
}
```

无论是求面积还是周长，扫描线的结构大概就是上面的样子。

需要解决的几个问题：

现在有两点需要说明一下。

（1）：线段树进行线段操作时，每个点的含义（比如为什么Cover函数中，y2后面要-1）。

（2）：线段树如何维护扫描线过程中的覆盖线段长度。

（3）：线段树如何维护扫描线过程中线段的数量。

（1）：线段树中点的含义

线段树如果没有离散化，那么线段树下标为1，就代表线段[1,2)

线段树下标为K的时候，代表的线段为[K,K+1) （长度为1）

所以，将上面的所有线段都化为[y1,y2)就可以理解了，线段[y1,y2)只包括线段树下标中的y1,y1+1,...,y2-1

当y值的范围是10^9时，就不能再按照上面的办法按值建树了，这时需要离散化。

下面是离散化的代码：

[cpp]

```cpp
//
int Rank[maxn],Rn;
void SetRank(){//调用前，所有y值被无序存入Rank数组，下标为[1..Rn]
    int I=1;
    //第一步排序
    sort(Rank+1,Rank+1+Rn);
    //第二步去除重复值
    for(int i=2;i<=Rn;++i) if(Rank[i]!=Rank[i-1]) Rank[++I]=Rank[i];
    Rn=I;
    //此时，所有y值被从小到大无重复地存入Rank数组，下标为[1..Rn]
}
int GetRank(int x){//给定x，求x的下标
    //二分法求下标
    int L=1,R=Rn,M;//[L,R] first >=x
    while(L!=R){
        M=(L+R)>>1;
        if(Rank[M]<x) L=M+1;
        else R=M;
    }
    return L;
}
```

[cpp]

```cpp
//
int Rank[maxn],Rn;
void SetRank(){//调用前，所有y值被无序存入Rank数组，下标为[1..Rn]
    int I=1;
    //第一步排序
    sort(Rank+1,Rank+1+Rn);
    //第二步去除重复值
    for(int i=2;i<=Rn;++i) if(Rank[i]!=Rank[i-1]) Rank[++I]=Rank[i];
    Rn=I;
    //此时，所有y值被从小到大无重复地存入Rank数组，下标为[1..Rn]
}
int GetRank(int x){//给定x，求x的下标
    //二分法求下标
    int L=1,R=Rn,M;//[L,R] first >=x
    while(L!=R){
        M=(L+R)>>1;
        if(Rank[M]<x) L=M+1;
        else R=M;
    }
    return L;
}
```

此时，线段树的下标的含义就变成：如果线段树下标为K,代表线段[Rank[K] , Rank[K+1])。
下标为K的线段长度为Rank[K+1]-Rank[K]
所以此时叶节点的线段长度不是1了。
这时，之前的扫描线算法的函数调用部分就稍微的改变了一点：

[cpp]

```cpp
//
if(Line[I].In) Cover(GetRank(Line[I].y1),GetRank(Line[I].y2)-1,1,n,1);
else         Uncover(GetRank(Line[I].y1),GetRank(Line[I].y2)-1,1,n,1);
```

[cpp]

```cpp
//
if(Line[I].In) Cover(GetRank(Line[I].y1),GetRank(Line[I].y2)-1,1,n,1);
else         Uncover(GetRank(Line[I].y1),GetRank(Line[I].y2)-1,1,n,1);
```


看着有点长，其实不难理解，只是多了一步从y值到离散之后的下标的转换。

注意一点，如果下标为K的线段长度为Rank[K+1]-Rank[K]，那么下标为Rn的线段树的长度呢？
其实这个不用担心，Rank[Rn]作为所有y值中的最大值，它肯定是一个线段的右端点，
而右端点求完离散之后的下标还要-1，所以上面的线段覆盖永远不会覆盖到Rn。
所以线段树其实只需要建立Rn-1个元素，因为下标为Rn的无法定义，也不会被访问。
不过有时候留着也有好处，这个看具体实现时自己取舍。

（2）：如何维护覆盖线段长度
先提一个小技巧，一般，利用两个子节点来更新本节点的函数写成PushUp();
但是，对于比较复杂的子区间合并问题，在区间查询的时候，需要合并若干个子区间。
而合并子区间是没办法用PushUp函数的。于是，对于比较复杂的问题，把单个节点的信息写成一个结构体。
在结构体内重载运算符"+"，来实现区间合并。这样，不仅在PushUp函数可以调用这个加法，区间询问时也可以调用这个加法，这样更加方便。

下面给出维护线段覆盖长度的节点定义：

[cpp]

```cpp
//
struct Node{
    int Cover;//区间整体被覆盖的次数
    int L;//Length : 所代表的区间总长度
    int CL;//Cover Length :实际覆盖长度
    Node operator +(const Node &B)const{
        Node X;
        X.Cover=0;//因为若上级的Cover不为0，不会调用子区间加法函数
```

```cpp
        X.L=L+B.L;
        X.CL=CL+B.CL;
        return X;
    }
}K[maxn<<2];
```

[cpp]

```cpp
//
struct Node{
    int Cover;//区间整体被覆盖的次数
    int L;//Length : 所代表的区间总长度
    int CL;//Cover Length :实际覆盖长度
    Node operator +(const Node &B)const{
        Node X;
        X.Cover=0;//因为若上级的Cover不为0，不会调用子区间加法函数
        X.L=L+B.L;
        X.CL=CL+B.CL;
        return X;
    }
}K[maxn<<2];
```

这样定义之后，区间的信息更新是这样的：

若本区间的覆盖次数大于0，那么令CL=L,直接为全覆盖，不管下层是怎么覆盖的，反正本区间已经全被覆盖。

若本区间的覆盖次数等于0，那么调用上面结构体中的加法函数，利用子区间的覆盖来计算。

加入一条线段就是给每一个分解的子区间的Cover+1,删除线段就-1，每次修改Cover之后，更新区间信息。

这里完全没有下推标记的过程。

查询的代码如下：

如果不把区间加法定义成结构体内部的函数，而是定义在PushUp函数内，那么这里几乎就要重写一遍区间合并。

因为PushUp在这里用不上。

[cpp]

```cpp
//
Node Query(int L,int R,int l,int r,int rt){
    if(L <= l && r <= R){
        return K[rt];
    }
    int m=(l+r)>>1;
    Node LANS,RANS;
    int X=0;
    if(L <= m) LANS=Query(L,R,ls),X+=1;
    if(R >  m) RANS=Query(L,R,rs),X+=2;
    if(X==1) return LANS;
    if(X==2) return RANS;
    return LANS+RANS;
}
```


[cpp]

```cpp
//
Node Query(int L,int R,int l,int r,int rt){
    if(L <= l && r <= R){
        return K[rt];
    }
    int m=(l+r)>>1;
    Node LANS,RANS;
    int X=0;
    if(L <= m) LANS=Query(L,R,ls),X+=1;
    if(R >  m) RANS=Query(L,R,rs),X+=2;
    if(X==1) return LANS;
    if(X==2) return RANS;
    return LANS+RANS;
}
```


维护线段覆盖3次或以上的长度：

[cpp]

```cpp
//
struct Nodes{
    int C;//Cover
    int CL[4];//CoverLength[0~3]
    //CL[i]表示被覆盖了大于等于i次的线段长度，CL[0]其实就是线段总长
}ST[maxn<<2];
void PushUp(int rt){
    for(int i=1;i<=3;++i){
        if(ST[rt].C < i) ST[rt].CL[i]=ST[rt<<1].CL[i-ST[rt].C]+ST[rt<<1|1].CL[i-ST[rt].C];
        else ST[rt].CL[i]=ST[rt].CL[0];
    }
}
```

[cpp]

```cpp
//
struct Nodes{
    int C;//Cover
    int CL[4];//CoverLength[0~3]
    //CL[i]表示被覆盖了大于等于i次的线段长度，CL[0]其实就是线段总长
}ST[maxn<<2];
void PushUp(int rt){
    for(int i=1;i<=3;++i){
        if(ST[rt].C < i) ST[rt].CL[i]=ST[rt<<1].CL[i-ST[rt].C]+ST[rt<<1|1].CL[i-ST[rt].C];
        else ST[rt].CL[i]=ST[rt].CL[0];
    }
}
```


这里给出节点定义和PushUp().
更新节点信息的思路大概就是:
假设要更新CL[3],然后发现本节点被覆盖了2次，那么本节点被覆盖三次或以上的长度就等于子节点被覆盖了1次或以上的长度之和。
而CL[0]建树时就赋值，之后不需要修改。

（3）：如何维护扫描线过程中线段的数量

[cpp]

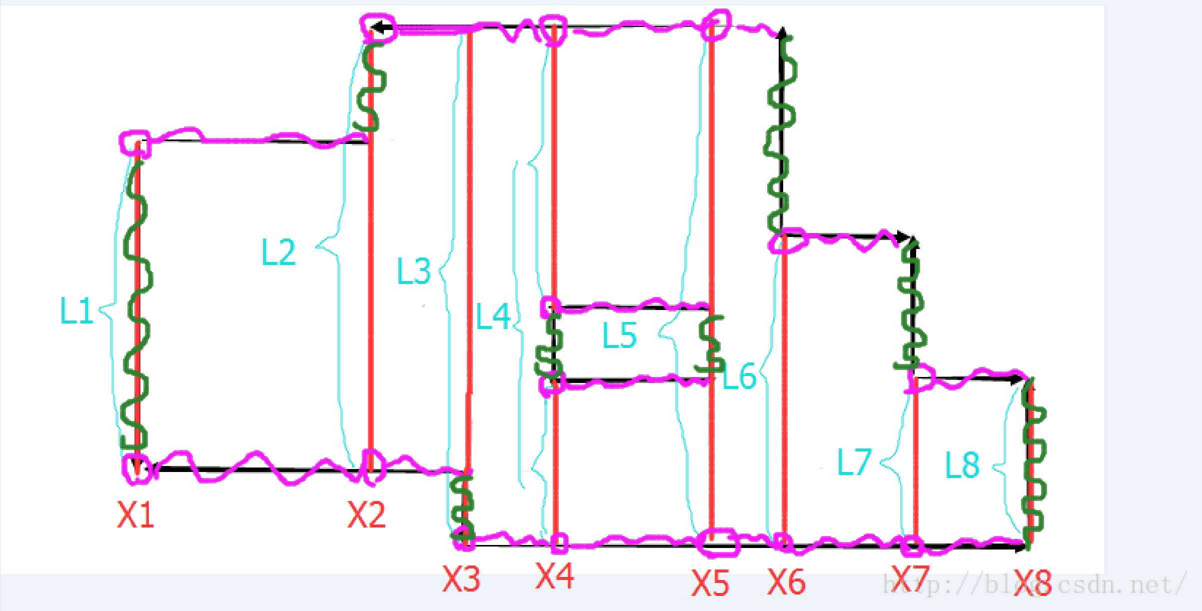
```cpp
//
struct Node{
    int cover;//完全覆盖层数
    int lines;//分成多少个线段
    bool L,R;//左右端点是否被覆盖
    Node operator +(const Node &B){//连续区间的合并
        Node C;
        C.cover=0;
        C.lines=lines+B.lines-(R&&B.L);
        C.L=L;C.R=B.R;
        return C;
    }
}K[maxn<<2];
```

[cpp]

```cpp
//
struct Node{
    int cover;//完全覆盖层数
    int lines;//分成多少个线段
    bool L,R;//左右端点是否被覆盖
    Node operator +(const Node &B){//连续区间的合并
        Node C;
        C.cover=0;
        C.lines=lines+B.lines-(R&&B.L);
        C.L=L;C.R=B.R;
        return C;
    }
}K[maxn<<2];
```

要维护被分成多少个线段，就需要记录左右端点是否被覆盖，知道了这个，就可以合并区间了。
左右两个区间合并时，若左区间的最右侧有线段且右区间的最左侧也有线段，那么这两个线段会合二为一，于是总线段数量会少1.

扫描线求重叠矩形周长：



这个图是在原来的基础上多画了一些东西，这次是要求周长。
所有的横向边都画了紫色，所有的纵向边画了绿色。

先考虑绿色的边，由图可以观察到，绿色边的长度其实就是L的变化值。
比如考虑X1,本来L是0,从0变到L1,所以绿色边长为L1.
再考虑X2,由L1变成了L2,所以绿色边长度为L2-L1,
于是，绿色边的长度就是L的变化值（注意上图中令L0=0,L9=0）。
因为长度是从0开始变化，最终归0.

再考虑紫色的边，要计算紫色边，其实就是计算L的线段是有几个线段组成的，每个线段会贡献两个端点（紫色圆圈）
而每个端点都会向右延伸出一条紫色边一直到下一个X值。

所以周长就是以上两部分的和。而两部分怎么维护，前面都讲过了，下面给出代码。

[cpp]

```cpp
//
struct Node{
    int cover;//完全覆盖层数
    int lines;//分成多少个线段
    bool L,R;//左右端点是否被覆盖
    int CoverLength;//覆盖长度
    int Length;//总长度
    Node(){}
    Node(int cover,int lines,bool L,bool R,int CoverLength):cover(cover),lines(lines),L(L),R(R),CoverLength(CoverLength){}
    Node operator +(const Node &B){//连续区间的合并
        Node C;
        C.cover=0;
        C.lines=lines+B.lines-(R&&B.L);
        C.CoverLength=CoverLength+B.CoverLength;
        C.L=L;C.R=B.R;
        C.Length=Length+B.Length;
        return C;
    }
}K[maxn<<2];
void PushUp(int rt){//更新非叶节点
    if(K[rt].cover){
        K[rt].CoverLength=K[rt].Length;
        K[rt].L=K[rt].R=K[rt].lines=1;
    }
    else{
        K[rt]=K[rt<<1]+K[rt<<1|1];
    }
```

```
}
```

```cpp
//
struct Node{
    int cover;//完全覆盖层数
    int lines;//分成多少个线段
    bool L,R;//左右端点是否被覆盖
    int CoverLength;//覆盖长度
    int Length;//总长度
    Node(){}
    Node(int cover,int lines,bool L,bool R,int CoverLength):cover(cover),lines(lines),L(L),R(R),CoverLength(CoverLength){}
    Node operator +(const Node &B){//连续区间的合并
        Node C;
        C.cover=0;
        C.lines=lines+B.lines-(R&&B.L);
        C.CoverLength=CoverLength+B.CoverLength;
        C.L=L;C.R=B.R;
        C.Length=Length+B.Length;
        return C;
    }
}K[maxn<<2];
void PushUp(int rt){//更新非叶节点
    if(K[rt].cover){
        K[rt].CoverLength=K[rt].Length;
        K[rt].L=K[rt].R=K[rt].lines=1;
    }
    else{
        K[rt]=K[rt<<1]+K[rt<<1|1];
    }
}
```

扫描的代码:

```cpp
int PreX=L[0].x;//前X坐标
int ANS=0;//目前累计答案
int PreLength=0;//前线段总长
int PreLines=0;//前线段数量
Build(1,20001,1);
for(int i=0;i<nL;++i){
    //操作
    if(L[i].c) Cover(L[i].y1,L[i].y2-1,1,20001,1);
    else Uncover(L[i].y1,L[i].y2-1,1,20001,1);
    //更新横向的边界
    ANS+=2*PreLines*(L[i].x-PreX);
    PreLines=K[1].lines;
    PreX=L[i].x;
    //更新纵向边界
    ANS+=abs(K[1].CoverLength-PreLength);
    PreLength=K[1].CoverLength;
}
//输出答案
printf("%d\n",ANS);
```

```cpp
int PreX=L[0].x;//前X坐标
int ANS=0;//目前累计答案
int PreLength=0;//前线段总长
int PreLines=0;//前线段数量
Build(1,20001,1);
for(int i=0;i<nL;++i){
    //操作
    if(L[i].c) Cover(L[i].y1,L[i].y2-1,1,20001,1);
    else Uncover(L[i].y1,L[i].y2-1,1,20001,1);
    //更新横向的边界
    ANS+=2*PreLines*(L[i].x-PreX);
    PreLines=K[1].lines;
```

```
    PreX=L[i].x;
    //更新纵向边界
    ANS+=abs(K[1].CoverLength-PreLength);
    PreLength=K[1].CoverLength;
}
//输出答案
printf("%d\n",ANS);
```

求立方体重叠3次或以上的体积：

这个首先扫描面，每个面内求重叠了3次或以上的面积，然后乘以移动距离就是体积。
面内扫描线，用线段树维护重叠了3次或以上的线段长度，然后用长度乘移动距离就是重叠了3次或以上的面积。
扫描面基本原理都跟扫描线一样，就是嵌套了一层而已，写的时候细心一点就没问题了。

八：可持久化（主席树）

可持久化线段树，也叫主席树。
可持久化数据结构思想，就是保留整个操作的历史，即，对一个线段树进行操作之后，保留访问操作前的线段树的能力。
最简单的方法，每操作一次，建立一颗新树。这样对空间的需求会很大。

而注意到，对于点修改，每次操作最多影响 $\lfloor \log_2(n-1) \rfloor + 2$ 个节点，于是，其实操作前后的两棵线段树，结构一样，

而且只有 $\lfloor \log_2(n-1) \rfloor + 2$ 个节点不同，其余的节点都一样，于是可以重复利用其余的点。

这样，每次操作，会增加 $\lfloor \log_2(n-1) \rfloor + 2$ 个节点。
于是，这样的线段树，每次操作需要O(log2(n))的空间。

题目：**HDU 2665 Kth number**　　题解
给定10万个数，10万个询问。
每个询问，问区间[L,R]中的数，从小到大排列的话，第k个数是什么。

这个题，首先对十万个数进行离散化，然后用线段树来维护数字出现的次数。
每个节点都存出现次数，那么查询时，若左节点的数的个数>=k，就往左子树递归，否则往右子树递归。
一直到叶节点，就找到了第k大的数。

这题的问题是，怎么得到一个区间的每个数出现次数。
注意到，数字的出现次数是满足区间减法的。
于是要求区间[L,R]的数，其实就是T[R]-T[L-1]　，其中T[X]表示区间[1,X]的数形成的线段树。
现在的问题就是，如何建立这10万棵线段树。

由之前的分析，需要O(n log2(n))的空间
下面是代码：

[cpp]

```cpp
//主席树
int L[maxnn],R[maxnn],Sum[maxnn],T[maxn],TP;//左右子树，总和，树根，指针
void Add(int &rt,int l,int r,int x){//建立新树，l,r是区间， x是新加入的数字的排名
    ++TP;L[TP]=L[rt];R[TP]=R[rt];Sum[TP]=Sum[rt]+1;rt=TP;//复制&新建
    if(l==r) return;
    int m=(l+r)>>1;
    if(x <= m) Add(L[rt],l,m,x);
    else       Add(R[rt],m+1,r,x);
}
int Search(int TL,int TR,int l,int r,int k){//区间查询第k大
    if(l==r) return l;//返回第k大的下标
    int m=(l+r)>>1;
    if(Sum[L[TR]]-Sum[L[TL]]>=k) return Search(L[TL],L[TR],l,m,k);
    else return Search(R[TL],R[TR],m+1,r,k-Sum[L[TR]]+Sum[L[TL]]);
}
```

[cpp]

```cpp
//主席树
int L[maxnn],R[maxnn],Sum[maxnn],T[maxn],TP;//左右子树，总和，树根，指针
```

```
void Add(int &rt,int l,int r,int x){//建立新树，l,r是区间， x是新加入的数字的排名
    ++TP;L[TP]=L[rt];R[TP]=R[rt];Sum[TP]=Sum[rt]+1;rt=TP;//复制&新建
    if(l==r) return;
    int m=(l+r)>>1;
    if(x <= m) Add(L[rt],l,m,x);
    else       Add(R[rt],m+1,r,x);
}
int Search(int TL,int TR,int l,int r,int k){//区间查询第k大
    if(l==r) return l;//返回第k大的下标
    int m=(l+r)>>1;
    if(Sum[L[TR]]-Sum[L[TL]]>=k) return Search(L[TL],L[TR],l,m,k);
    else return Search(R[TL],R[TR],m+1,r,k-Sum[L[TR]]+Sum[L[TL]]);
}
```

以上就是主席树部分的代码。
熟悉SBT的，应该都很熟悉这种表示方法。
L,R是伪指针，指向左右子节点。
特殊之处是，0 表示空树，并且 L[0]=R[0]=0.
也就是说，空树的左右子树都是空树。
而本题中，每一颗树其实都是完整的，刚开始有一颗空树。
但是刚开始的空树，真的需要用空间去存吗？
其实不需要，刚开始的空树有这些性质：
1.每个节点的Sum值为0
2.每个非叶节点的左右子节点的Sum值也是0

而SBT的空树刚好满足这个性质。而线段树不依赖L,R指针来结束递归。
线段树是根据区间l,r来结束的，所以不会出现死循环。

所以只需要把Sum[0]=0;那么刚开始就不需要建树了，只有每个操作的 $\lfloor \log_2(n-1) \rfloor + 2$ 个节点。

这个线段树少了表示父节点的int rt，因为不需要（也不能够）通过rt来找子节点了，而是直接根据L,R来找。

----------------------------- 补充 -------------------------------------

终于又找到一道可以用主席树的题目了：Codeforces 650D.Zip-line 题解
做这题之前需要会求普通的LIS问题（最长上升子序列问题）。

九：练习题

适合非递归线段树的题目：

Codeforces 612D The Union of k-Segments : 题解
题意：线段求交，给定一堆线段，按序输出被覆盖k次或以上的线段和点。
基础题，先操作，最后一次下推标记，然后输出，
维护两个线段树，一个线段覆盖，一个点覆盖。

Codeforces 35E Parade : 题解
题意：给定若干矩形，下端挨着地面，求最后的轮廓形成的折线，要求输出每一点的坐标。
思路：虽然是区间修改的线段树，但只需要在操作结束后一次下推标记，然后输出，所以适合非递归线段树。

URAL 1846 GCD2010 : 题解
题意：总共10万个操作，每次向集合中加入或删除一个数，求集合的最大公因数。（规定空集的最大公因数为1）

Codeforces 12D Ball : 题解
题意：
给N (N<=500000)个点，每个点有x,y,z (0<= x,y,z <=10^9)
对于某点(x,y,z)，若存在一点(x1,y1,z1)使得x1 > x && y1 > y && z1 > z 则点(x,y,z)是特殊点。
问N个点中，有多少个特殊点。

提示：排序+线段树

Codeforces 19D Points : 题解
题意：
给定最多20万个操作，共3种：
1.add x y ：加入(x,y)这个点
2.remove x y ：删除(x,y)这个点

3.find x y　　：找到在(x,y)这点右上方的x最小的点，若x相同找y最小的点，输出这点坐标，若没有，则输出-1.
提示：排序，线段树套平衡树

Codeforces 633E Startup Funding : 题解
这题需要用到一点概率论，组合数学知识，和二分法。
非递归线段树在这题中主要解决RMQ问题（区间最大最小值问题），由于不带修改，这题用Sparse Table求解RMQ是标答。
因为RMQ询问是在二分法之内求的，而Sparse Table可以做到O(1)查询，所以用Sparse Table比较好，总复杂度O(n*log(n))。
不过非递归线段树也算比较快的了，虽然复杂度是O(n*log(n)*log(n))，还是勉强过了这题。

扫描线题目：

POJ 1177 Picture:给定若干矩形求合并之后的图形周长　　题解

HDU 1255 覆盖的面积：给定平面上若干矩形,求出被这些矩形覆盖过至少两次的区域的面积.　题解

HDU 3642 Get The Treasury：给定若干空间立方体，求重叠了3次或以上的体积（这个是扫描面，每个面再扫描线）题解

POJ 2482 Stars in your window : 给定一些星星的位置和亮度，求用W*H的矩形能够框住的星星亮度之和最大为多少。　题解

递归线段树题目：

Codeforces 558E A Simple Task　题解
给定一个长度不超过10^5的字符串（小写英文字母），和不超过5000个操作。
每个操作 L R K 表示给区间[L,R]的字符串排序，K=1为升序，K=0为降序。
最后输出最终的字符串。

Codeforces 527C Glass Carving　:　题解
给定一个矩形，不停地纵向或横向切割，问每次切割后，最大的矩形面积是多少。

URAL1989 Subpalindromes　　题解
给定一个字符串(长度<=100000)，有10万个操作。
操作有两种：
1：改变某个字符。
2：判断某个子串是否构成回文串。

HDU 4288 Coder :　题解
题意：对一个集合进行插入与删除操作。要求询问某个时刻，集合中的元素从小到大排序之后，序号%5 ==3 的元素值之和。
这题其实不一定要用线段树去做的，不过线段树还是可以做的。

HDU 2795 BillBoard : 题解
题意：有一个板，h行，每行w长度的位置。每次往上面贴一张海报，长度为1*wi .
每次贴的时候，需要找到最上面的，可以容纳的空间，并且靠边贴。

Codeforces 374D Inna and Sequence : 题解
题意：给定百万个数a[m]，然后有百万个操作，每次给现有序列加一个字符（0或1），或者删掉已有序列中，第 a[0] 个，第a[1]个,...，第a[m]个。

Codeforces 482B Interesting Array:　题解
题意就是，给定n,m.
满足m个条件的n个数，或说明不存在。
每个条件的形式是，给定 Li,Ri,Qi ，要求　a[Li]&a[Li+1]&...&a[Ri] = Qi ;
Codeforces 474E Pillar（线段树+动态规划）：　题解
题意就是，给定10^5 个数（范围10^15）,求最长子序列使得相邻两个数的差大于等于 d。

POJ 2777 Count Color :　题解
给线段涂颜色，最多30种颜色，10万个操作。
每个操作给线段涂色，或问某一段线段有多少种颜色。
30种颜色用int的最低30位来存，然后线段树解决。

URAL 1019 Line Painting: 线段树的区间合并　题解
给一段线段进行黑白涂色，最后问最长的一段白色线段的长度。

Codeforces 633H Fibonacci-ish II : 题解
这题需要用到莫队算法（Mo's Algorithm）+线段树区间修改，不过是单边界的区间，写起来挺有趣。
另一种解法就是暴力，很巧妙的方法，高复杂度+低常数居然就这么给过了。

树套树题目：

ZOJ 2112 Dynamic Rankings 动态区间第k大　题解

做法：树状数组套主席树 或者 线段树套平衡树

Codeforces 605D Board Game : 题解

做法：广度优先搜索(BFS)　+　线段树套平衡树

Codeforces 19D Points : 题解

题意：

给定最多20万个操作，共3种：

1.add x y　　：加入(x,y)这个点

2.remove x y ：删除(x,y)这个点

3.find x y　　：找到在(x,y)这点右上方的x最小的点，若x相同找y最小的点，输出这点坐标，若没有，则输出-1.

提示：排序，线段树套平衡树