

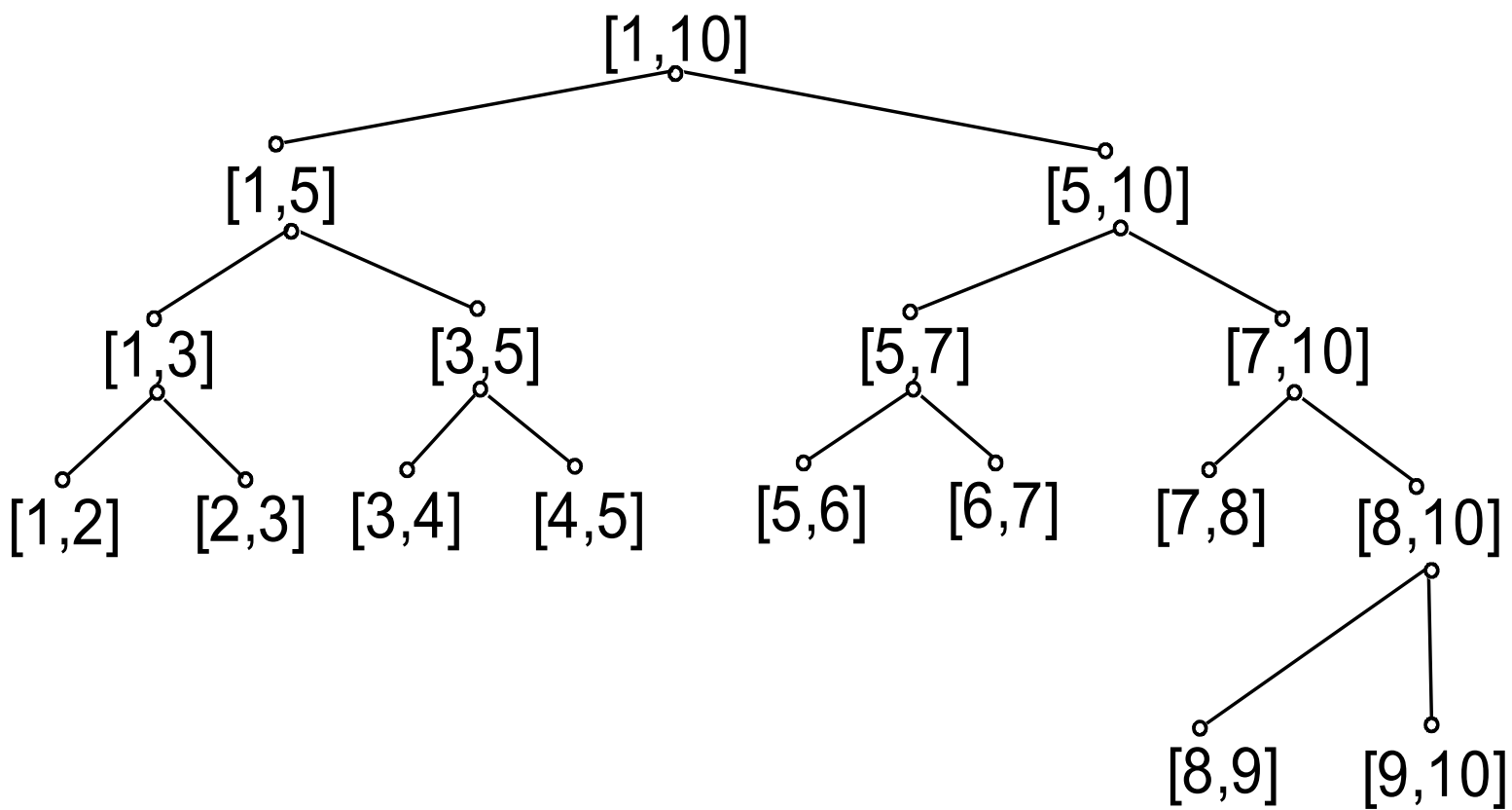
线段树

线段树

- 在一类问题中，我们需要经常处理可以映射在一个坐标轴上的一些固定线段，例如说映射在 OX 轴上的线段。由于线段是可以互相覆盖的，有时需要动态地取线段的并，例如取得并区间的总长度，或者并区间的个数等等。一个线段是对应于一个区间的，因此线段树也可以叫做区间树。

线段树的构造思想

- 线段树是一棵二叉树，树中的每一个结点表示了一个区间 $[a,b]$ 。每一个叶子节点表示了一个单位区间。对于每一个非叶结点所表示的结点 $[a,b]$ ，其左儿子表示的区间为 $[a,(a+b)/2]$ ，右儿子表示的区间为 $[(a+b)/2,b]$ 。

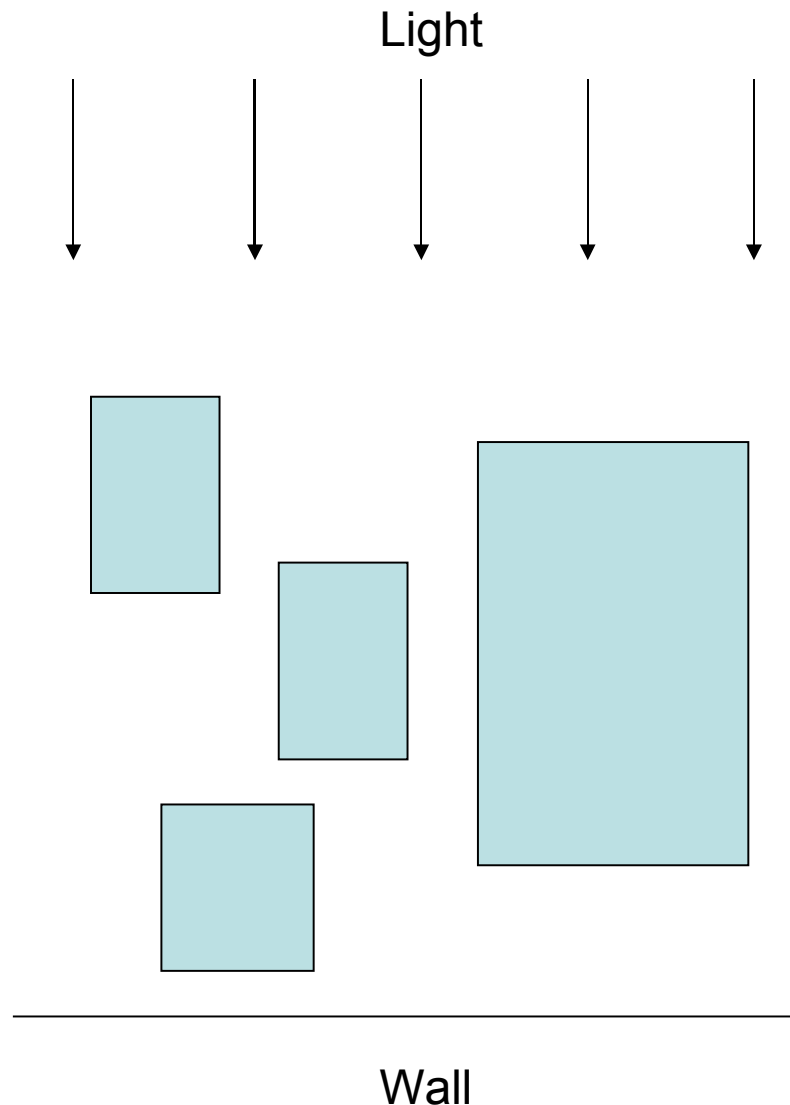


线段树的运用

- 线段树的每个节点上往往都增加了一些其他的域。在这些域中保存了某种动态维护的信息，视不同情况而定。这些域使得线段树具有极大的灵活性，可以适应不同的需求。

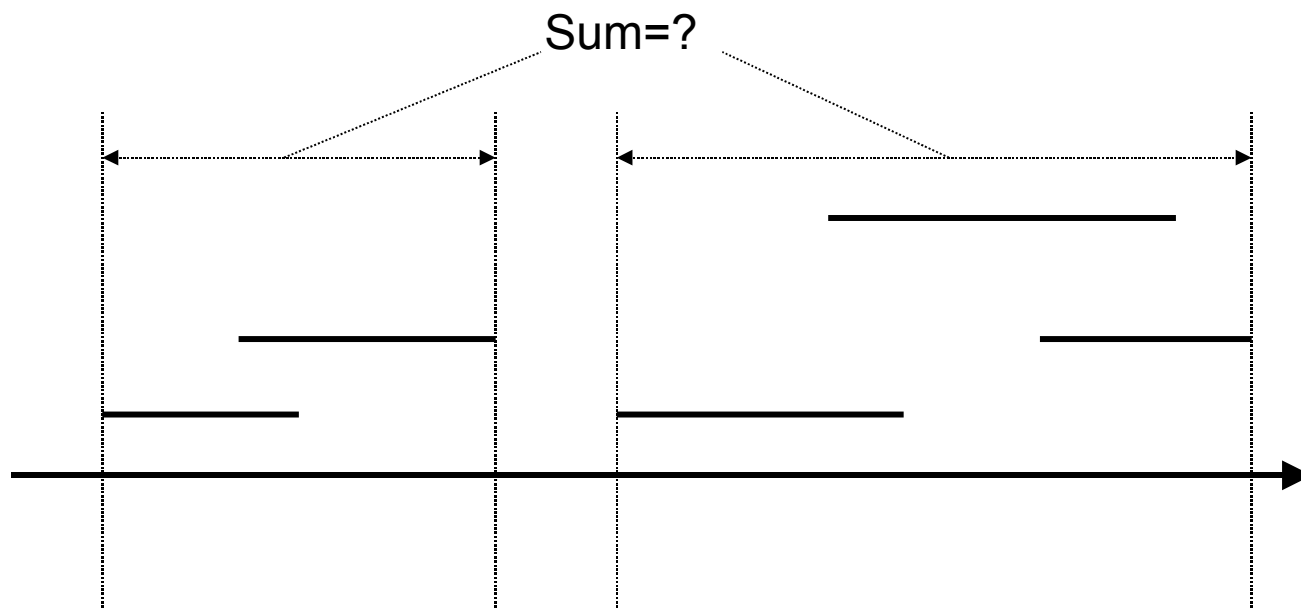
例 1

- 桌子上零散地放着若干个盒子，桌子的后方是一堵墙。如右图所示。现在从桌子前方射来一束平行光，把盒子的影子投射到了墙上。问影子的总宽度是多少？



分析

- 这道题目是一个经典的模型。在这里，我们略去某些处理的步骤，直接分析重点问题，可以把题目抽象地描述如下： x 轴上有若干条线段，求线段覆盖的总长度。



最直接的做法

- 设线段坐标范围为 $[\min, \max]$ 。使用一个下标范围为 $[\min, \max-1]$ 的一维数组，其中数组的第 i 个元素表示 $[i, i+1]$ 的区间。数组元素初始化全部为 0。对于每一条区间为 $[a, b]$ 的线段，将 $[a, b]$ 内所有对应的数组元素均设为 1。最后统计数组中 1 的个数即可。

示例

- 初始情况

0	0	0	0	0
---	---	---	---	---

- [1 , 2]

1	0	0	0	0
---	---	---	---	---

- [3 , 5]

1	0	1	1	0
---	---	---	---	---

- [4 , 6]

1	0	1	1	1
---	---	---	---	---

- [5 , 6]

1	0	1	1	1
---	---	---	---	---

4个
1

缺点

- 此方法的时间复杂度决定于下标范围的平方。
- 当下标范围很大时（ $[0, 10000]$ ），此方法效率太低。

离散化的做法

- 基本思想：先把所有端点坐标从小到大排序，将坐标值与其序号一一对应。这样便可以将原先的坐标值转化为序号后，对其应用前一种算法，再将最后结果转化回来得解。
- 该方法对于线段数相对较少的情况有效。

示例

- [10000,22000] [30300,55000] [44000,60000] [55000,60000]
- 排序得 10000 , 22000 , 30300 , 44000 , 55000 , 60000
- 对应得 1 , 2 , 3 , 4 , 5 , 6
- [1,2] [3,5] [4,6] [5,6]

示例

- 初始情况

0	0	0	0	0
---	---	---	---	---

- [1 , 2]

1	0	0	0	0
---	---	---	---	---

- [3 , 5]

1	0	1	1	0
---	---	---	---	---

- [4 , 6]

1	0	1	1	1
---	---	---	---	---

- [5 , 6]

1	0	1	1	1
---	---	---	---	---

4个
1

示例

- 10000 , 22000 , 30300 , 44000 , 55000 , 60000
- 1 , 2 , 3 , 4 , 5 , 6

1	2	3	4	5	6
1	0	1	1	1	
10000	22000	30300	44000	55000	60000

- $(22000-10000)+(60000-30300)=41700$

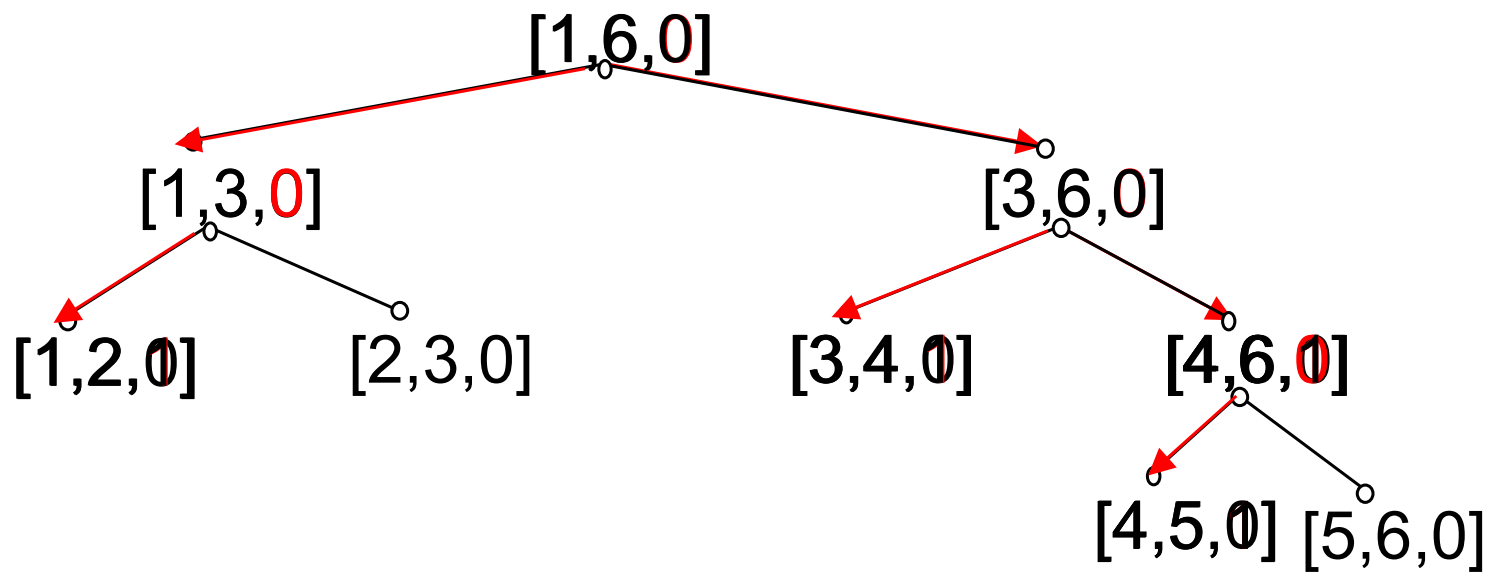
缺点

- 此方法的时间复杂度决定于线段数的平方。
 -
- 对于线段数较多的情况此方法效率太低。

使用线段树的做法

- 给线段树每个节点增加一个域 **cover**。**cover=1** 表示该结点所对应的区间被完全覆盖，**cover=0** 表示该结点所对应的区间未被完全覆盖。

加入 ~~[3,0]~~



程序实现

线段树的数据结构表示

- 1、动态数据结构
- 2、完全二叉树

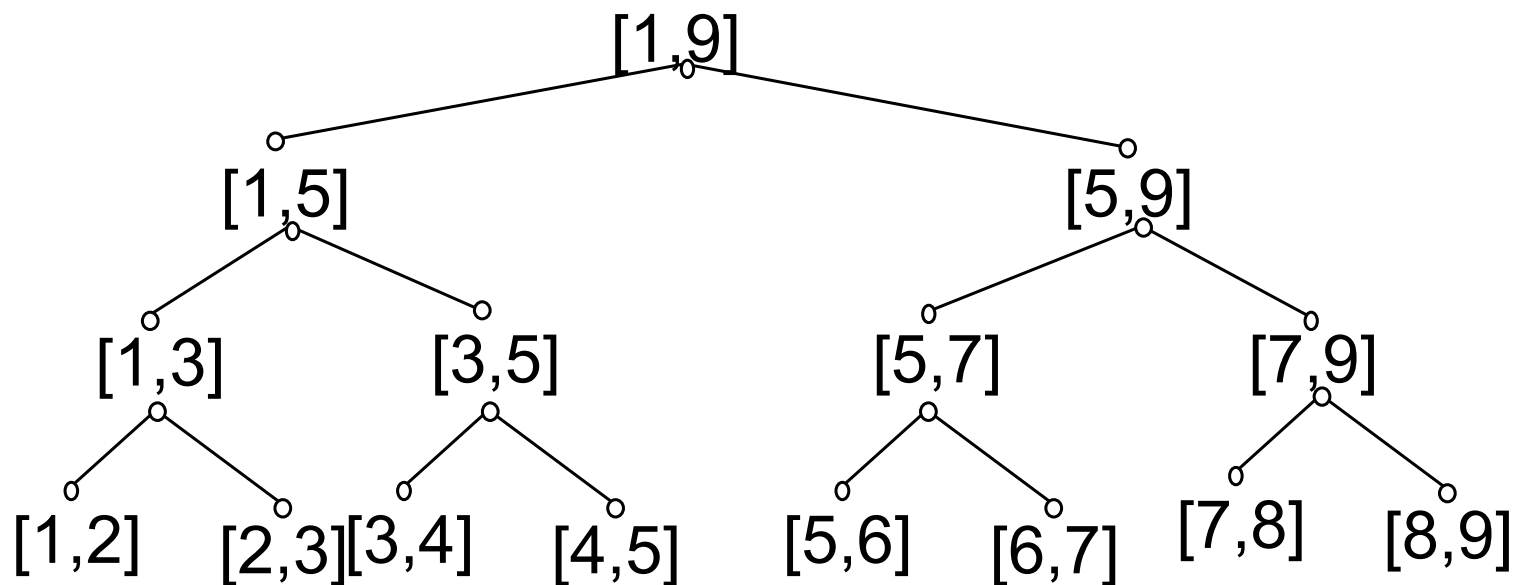
动态数据结构

- type
- pNode = ^TreeNode;
- TreeNode = record
- b, e: Integer;
- l, r: pNode;
- cover: Integer;
- end;

对应区间

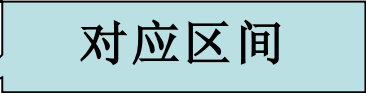
左右孩子

完全二叉树



完全二叉树

- type
- TreeNode = record
- b, e: Integer;
- cover: Integer;
- end;



对应区间

插入算法

- procedure Insert(p, a, b: Integer);
 - var
 - m: Integer;
 - begin
 - if Tree[p].cover = 0 then
 - begin
 - m := (Tree[p].b + Tree[p].e) div 2;
 - if (a = Tree[p].b) and (b = Tree[p].e) then
 - Tree[p].cover := 1
 - else if b <= m then Insert(p * 2, a, b)
 - else if a >= m then Insert(p * 2 + 1, a, b)
 - else begin
 - Insert(p * 2, a, m);
 - Insert(p * 2 + 1, m, b);
 - end;
 - end;
 - end;
- 未被完全覆盖
- 取中值
- 完全覆盖
- 在左边
- 在右边
- 二分

统计算法

- function Count(p: Integer): Integer;
- begin
- if Tree[p].cover = 1 then
- Count := Tree[p].e - Tree[p].b
- else if Tree[p].e - Tree[p].b = 1 then Count := 0
- else Count := Count(p * 2) + Count(p * 2 + 1);
- end;

被完全覆盖

是单位区间

二分递归求解


- 事实上，我们也可以不在每个节点中保存其表示范围，而是在递归调用时增加两个参数来加以表示。

另一种定义

- type
- TreeNode = record
- cover: Integer;
- end;

插入算法

- procedure Insert(p, l, r, a, b: Integer);
- var
- m: Integer;
- begin
- if Tree[p].cover = 0 then
- begin
- m := (l + r) div 2;
- if (a = l) and (b = r) then
- Tree[p].cover := 1
- else if b <= m then Insert(p * 2, l, m, a, b)
- else if a >= m then Insert(p * 2 + 1, m, r, a, b)
- else begin
- Insert(p * 2, l, m, a, m);
- Insert(p * 2 + 1, m, r, m, b);
- end;
- end;
- end;



**Tree[p].b 换成了
l, Tree[p].e 换成了
r, 递归时需要多加两
个参数, 其余都一样**

统计算法

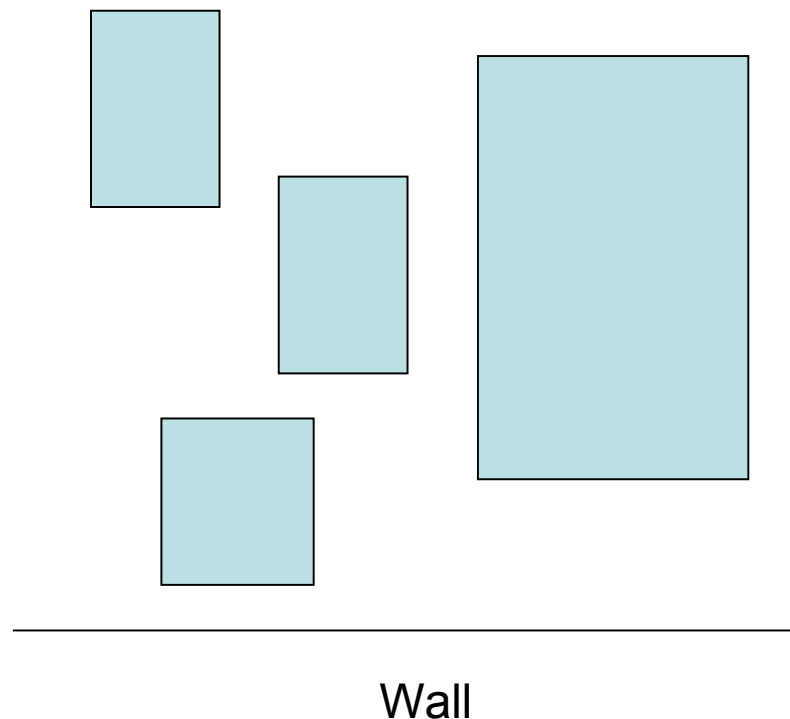
- function Count(p, l, r: Integer): Integer;
- begin
- if Tree[p].cover = 1 then
- Count := r - l
- else if r - l = 1 then Count := 0
- else Count := Count(p * 2, l, (l + r) div 2)
- + Count(p * 2 + 1, (l + r) div 2, r);
- end;



这个也一样

例 2

- 桌子上零散地放着若干个盒子，桌子的后方是一堵墙。如右图所示。问从桌子前方可以看到多少个盒子？假设人站得足够远。



分析

- 可以这样来看这道题： x 轴上有若干条不同线段，将它们依次染上不同的颜色，问最后能看到多少种不同的颜色？（后染的颜色会覆盖原先的颜色）
- 我们可以这样规定： x 轴初始是颜色 0，第一条线段染颜色 1，第二条线段染颜色 2，以此类推。

分析

- 原先构造线段树的方法不再适用，但是我们可以通过修改线段树的 **cover** 域的定义，使得这道题也能用线段树来解。
- 定义 **cover** 如下：**cover**=-1 表示该区间由多种颜色组成。**cover**>=0 表示该区间只有一种单一的颜色 **cover**。

插入算法

未被完全覆盖或者染色不同

- procedure Insert(p, l, r, a, b, c: Integer);
- var
- m: Integer;
- begin
- if Tree[p].cover <> c then
- begin
- m := (l + r) div 2;
- if (a = l) and (b = r) then Tree[p].cover := c
- else begin
- if Tree[p].cover >= 0 then
- begin
- Tree[p * 2].cover := Tree[p].cover;
- Tree[p * 2 + 1].cover := Tree[p].cover;
- Tree[p].cover := -1;
- end;
- if b <= m then Insert(p * 2, l, m, a, b, c)
- else if a >= m then Insert(p * 2 + 1, m, r, a, b, c)
- else begin
- Insert(p * 2, l, m, a, m, c);
- Insert(p * 2 + 1, m, r, m, b, c);
- end;
- end;
- end;
- end;

为什么？
有可能越界吗？

统计算法

- 使用一个数组 **Flag**，初始化为 0。遍历线段树，对于每种颜色 **c** 对 **Flag[c]** 赋值 1。最后统计 **Flag** 中 1 的个数即可。（注意颜色 0 应该排除在外，可以在最后减 1）

统计算法

- procedure Count(p, l, r: Integer);
- begin
- if Tree[p].cover \geq 0 then Flag[Tree[p].cover] := 1
- else if $r - l > 1$ then
- begin
- Count(p * 2, l, $(l + r) \text{ div } 2$);
- Count(p * 2 + 1, $(l + r) \text{ div } 2$, r);
- end;
- end;

例 3

- 把例 2 稍加改动，规定：线段的颜色可以相同。连续的相同颜色被视作一段。问 x 轴被分成多少段。



分析

- 仍然定义 **cover** 如下: **cover=-1** 表示该区间由多种颜色组成。 **cover>=0** 表示该区间只有一种单一的颜色 **cover** 。

插入算法

- 插入算法不变

统计算法

最左边的颜

色

最右边的颜

色

最左颜色 = 最右颜色 =
本身

非底色则统计数加 1

连接处颜色相同并
且非底色，则总数
减 1

```
• function Count(p, l, r: Integer; var lc, rc: Integer): Integer;  
• var  
•   result, tl, tr: Integer;  
• begin  
•   if Tree[p].cover >= 0 then  
•   begin  
•     lc := Tree[p].cover;  
•     rc := Tree[p].cover;  
•     if Tree[p].cover > 0 then Count := 1  
•     else Count := 0;  
•   end  
•   else if r - l > 1 then  
•   begin  
•     result := Count(p * 2, l, (l + r) div 2, lc, tl) + Count(p * 2 + 1, (l + r) div 2, r, tr, rc);  
•     if (tl = tc) and (tl > 0) then  
•       result := result - 1;  
•     Count := result;  
•   end;  
• end;
```

例 4

- x 轴上有若干条不同线段，问某个单位区间 $[x, x+1]$ 上重叠了多少条线段？



分析

- 为线段树每个节点增加一个 **Count** 域。表示所对应区间上重叠的线段数。
- 思考线段树的构造方法：当某线段能够完整覆盖某个结点所对应的区间时，则不再二分。因此要统计某个单位区间上重叠的线段总数，必须把从叶结点到根结点路径上所有结点的 **count** 域累加。

插入算法

- procedure Insert(p, l, r, a, b: Integer);
- var
- m: Integer;
- begin
- m := (l + r) div 2;
- if (a = l) and (b = r) then Tree[p].count := Tree[p].count + 1
- else begin
- if b <= m then Insert(p * 2, l, m, a, b)
- else if a >= m then Insert(p * 2 + 1, m, r, a, b)
- else begin
- Insert(p * 2, l, m, a, m);
- Insert(p * 2 + 1, m, r, m, b);
- end;
- end;
- end;

统计算法

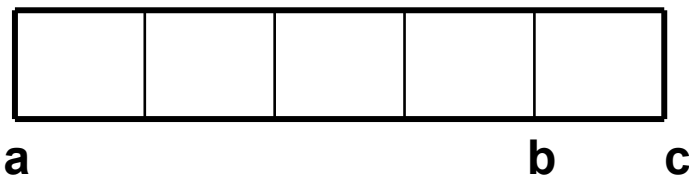
- function Count(p: Integer): Integer;
- var
- result: Integer;
- begin
- result := 0;
- while p > 0 do
- begin
- result := result + Tree[p].count;
- p := p div 2;
- end;
- Count := result;
- end;

例 5

- 一行 N 个方格，开始每个格子中的数都是 0。现在动态地提出一些问题和修改：提问的形式是求某一个特定的子区间 $[a,b]$ 中所有元素的和；修改的规则是指定某一个格子 x ，加上或者减去一个特定的值 A 。现在要求你能对每个提问作出正确的回答。 $1 \leq N \leq 1024$ ，提问和修改的总数可能达到 60000 条。

用线段树解

- 为线段树每个节点增加一个 **Count** 域。表示所对应区间内元素之和。
- 每次修改一个格子，需要修改从叶结点到根结点路径上所有结点的值。
- 特别注意：题目中的区间是以元素为端点，因此 $[a,b]$ 和 $[b,c]$ 存在重合，这和我们之前讨论的区间定义不同。我们这里忽略预处理过程，直接使用之前的区间定义。



定义

- type
- TreeNode = record
- count: Integer;
- end;

插入算法

- procedure Modify(p, delta: Integer);
- begin
- repeat
- Tree[p].count := Tree[p].count + delta;
- p := p div 2;
- until p = 0;
- end;

统计算法

- function Count(p, l, r, a, b: Integer): Integer;
- var
- m: Integer;
- begin
- if (l = a) and (r = b) then Count := Tree[p].count
- else begin
- m := (l + r) div 2;
- if b <= m then Count := Count(p * 2, l, m, a, b)
- else if a >= m then Count := Count(p * 2 + 1, m, r, a, b)
- else Count := Count(p * 2, l, m, a, m)
- + Count(p * 2 + 1, m, r, m, b);
- end;
- end;

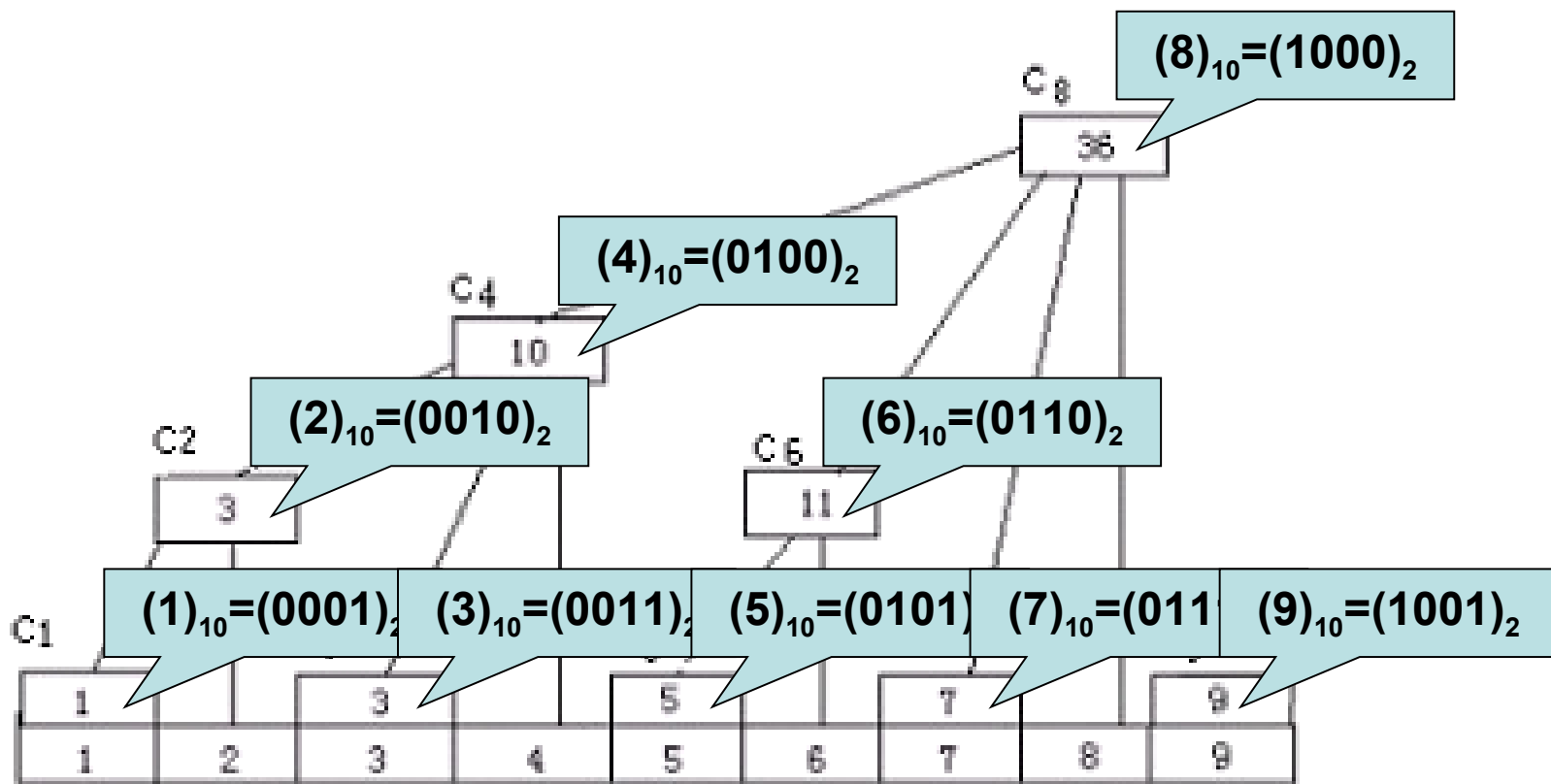
介绍另一种算法

- 对于序列 a ，我们设一个数组 C ，定义
- $C[i] = a[i - 2^k + 1] + \dots + a[i]$ ， k 为 i 在二进制下末尾 0 的个数。

图例

c 数组

a 数组



如何计算 x 对应的 2^k ?

- $2^k = x$ and $(x \gg k)$ k 为 x 在二进制数下末尾 0 的个数

- 以 6 为例

- $(6)_{10} = (0110)_2$

- $\text{xor } 6-1 = (5)_{10} = (0101)_2$

- $(0011)_2$

- $\text{and } (6)_{10} = (0110)_2$

- $(0010)_2$

- function Lowbit(x: Integer): Integer;
- begin
- Lowbit := x and (x xor (x – 1));
- end;

如何计算某个区间 $[a,b]$ 的和 $\text{sum}(a, b)$

- 我们这里所说的区间以元素为端点

a			b	c
---	--	--	---	---

- 把这个问题转化成为求 $\text{sum}(1,b)-\text{sum}(1,a-1)$
- 如何求 $\text{sum}(1,x)$?

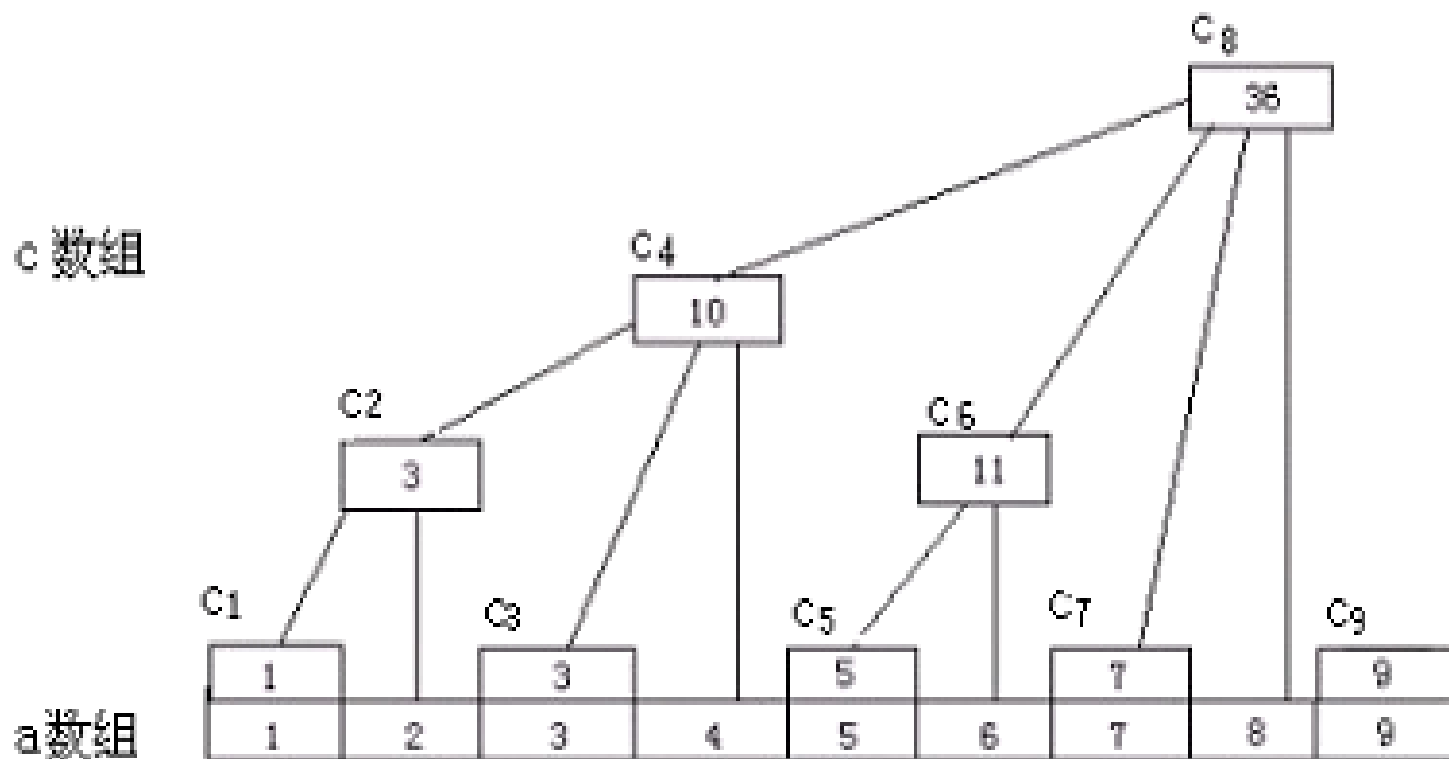
求和算法

- function Sum(x: Integer): Integer;
- var
- result: Integer;
- begin
- result := 0;
- while x > 0 do
- begin
- result := result + C[x];
- x := x - Lowbit(x);
- end;
- Sum := result;
- end;

如何修改一个元素的值

- 设要修改的元素是 $a[p]$
- 任意 x 满足 $x \geq p > x - \text{Lowbit}(x)$ 的 $C[x]$ 均要修改。

如何确定哪些 $C[x]$ 需要修改?



修改算法

- procedure Modify(p, delta: Integer);
- begin
- while p <= n do
- begin
- C[p] := C[p] + delta;
- p := p + Lowbit(p);
- end;
- end;

复杂度分析

- 很容易得出 **Sum** 和 **Modify** 的复杂度均为 $\log_2 n$

例 6

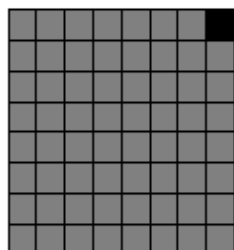
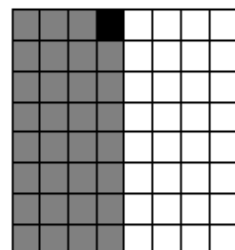
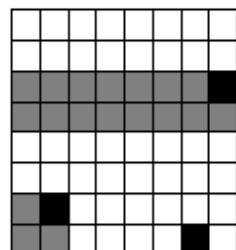
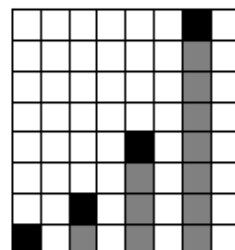
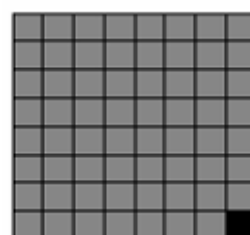
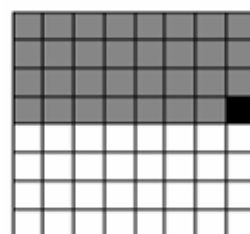
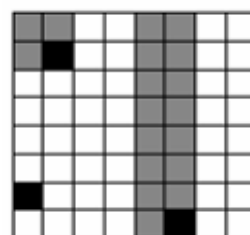
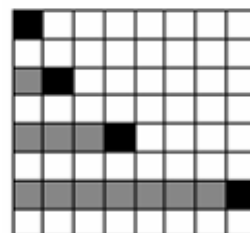
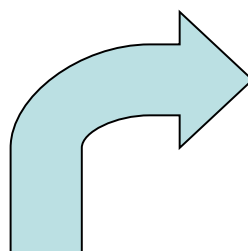
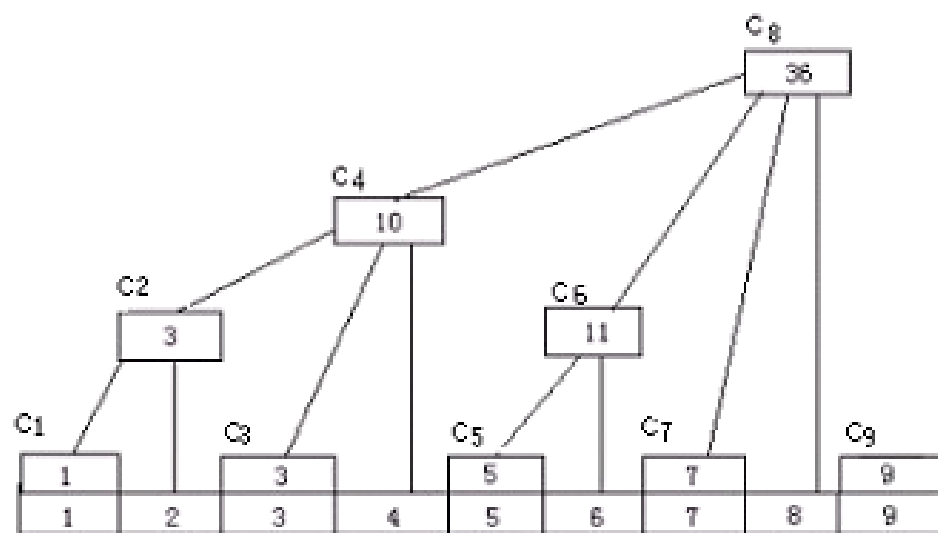
- 在一个 $N \times N$ 的方格中，开始每个格子里的数都是 0。现在动态地提出一些问题和修改：提问的形式是求某一个特定的子矩阵 $(x1,y1)-(x2,y2)$ 中所有元素的和；修改的规则是指定某一个格子 (x,y) ，在 (x,y) 中的格子元素上加上或者减去一个特定的值 A 。现在要求你能对每个提问作出正确的回答。 $1 \leq N \leq 1024$ ，提问和修改的总数可能达到 60000 条。

Mobile

- 例 6 实际上是例 5 的推广，从一维扩展到了二维。

c 数组

a 数组



求和算法

- function Sum(x, y: Integer): Integer;
- var
- result, t: Integer;
- begin
- result := 0;
- while x > 0 do
- begin
- t := y;
- while t > 0 do
- begin
- result := result + C[x, t];
- t := t - Lowbit(t);
- end;
- x := x - Lowbit(x);
- end;
- Sum := result;
- end;

修改算法

- procedure Modify(x, y, delta: Integer);
- var
- t: Integer;
- begin
- while x <= n do
- begin
- t := y;
- while t <= n do
- begin
- C[x, t] := C[x, t] + delta;
- t := t + Lowbit(t);
- end;
- x := x + Lowbit(p);
- end;
- end;

思考

- 1、能否把这种算法应用到前面的例题上去？
- 2、如果用线段树解 **Mobile** ，应该怎么做？

两种算法的比较

- 线段树对题目的适应性强，但是要求有较高的处理技巧。
- 后一种算法适用类型单一，但是算法巧妙，其效率也比线段树高。