解决动态统计问题的两把利刃

——剖析线段树与矩形切割

广东北江中学　　薛矛

【关键字】
线段树　矩形树　方块树　线段切割　矩形切割

【摘要】
本文从统计类型的问题出发，以更好地解决这类问题为目的，较详细地介绍了线段树的基本操作，改进和推广；矩形切割的思想以及具体的使用方法。并通过将线段树和矩形切割进行对比，分析了线段树和矩形切割的复杂度，优缺点等，提出了它们各自的适用范围，并总结出何时使用最合适。

【目录】

【正文】

一、引言

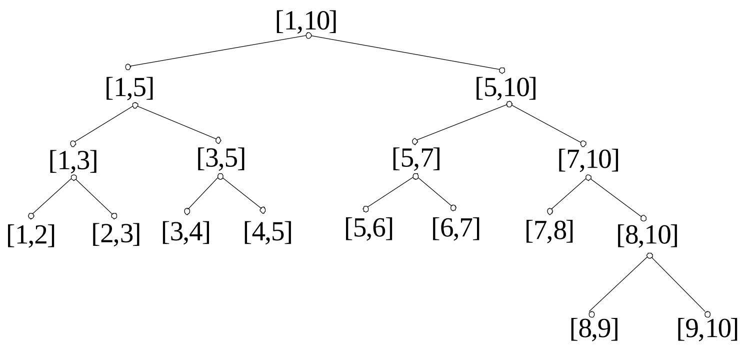
我们在做练习和比赛中，经常能碰见统计类型的题目。题目通过输入数据给程序提供事物信息，并要求程序能比较高效地求出某些时刻，某种情况下，事物的状态是怎样的。这类问题往往比较简单明了，也能十分容易地写出模拟程序。但较大的数据规模使得模拟往往不能满足要求。于是我们就要寻找更好的方法。本文将介绍解决此类问题的两种方法——线段树与矩形切割。

二、线段树

线段树已经不是一个陌生的名词了，相信大家也对线段树比较熟悉，这里只做简要的介绍。

2.1 线段树的结构

线段树是一棵二叉树，其结点是一条“线段”——[a,b]，它的左儿子和右儿子分别是这条线段的左半段和右半段，即[a, $\lfloor (a+b)/2 \rfloor$]和[$\lfloor (a+b)/2 \rfloor$,b]。线段树的叶子结点是长度为 1 的单位线段[a,a+1]。下图就是一棵根为[1,10]的线段树：



易证一棵以[a,b]为根的线段树结点数是 2*(b-a)-1。由于线段树是一棵平衡树，因此一棵以[a,b]为根结点的线段树的深度为 $\log_2(2*(b-a))$。

线段树中的结点一般采取如下数据结构：

```
Type TreeNode=Record
                  a,b,Left,Right:Longint
              End
```

其中 a,b 分别表示线段的左端点和右端点，Left，Right 表示左儿子和右儿子的编号。因此我们可以用一个一维数组来表示一棵线段树：

Tree:array[1..Maxn] of TreeNode;

a,b,Left,Right 这 4 个域是描述一棵线段树所必须的 4 个量。根据实际需要，我们可以增加其它的域，例如增加 Cover 域来计算该线段被覆盖的次数，bj 域用来表示结点的修改标记（后面将会提到）等等。

2.2 线段树的建立

我们可以用一个简单的过程建立一棵线段树。

```
Procedure MakeTree(a,b)
Var Now:Longint
Begin
   tot ← tot + 1
   Now ← tot
   Tree[Now].a ← a
   Tree[Now].b ← b
   If a +1 < b then
       Tree[Now].Left ← tot + 1
       MakeTree(a, ⌊(a+b)/2⌋)
       Tree[Now].Right ← tot + 1
       MakeTree(⌊(a+b)/2⌋ , b)
End
```

2.3 线段树中的线段插入和删除

增加一个 Cover 的域来计算一条线段被覆盖的次数，即数据结构变为：

```
Type
TreeNode=Record
                 a,b,Left,Right,Cover:Longint
              End
```

因此在 MakeTree 的时候应顺便把 Cover 置 0。

2.3.1 线段的插入

插入一条线段[c,d]

```
Procedure Insert(Num)
Begin
   If (c<Tree[Num].a)and(Tree[Num].b<d) then
         Tree[Num].Cover ← Tree[Num].Cover + 1
   Else
         If c<⌊(Tree[Num].a+Tree[Num].b)/2⌋ then
               Insert(Tree[Num].Left)
         If d>⌊(Tree[Num].a+Tree[Num].b)/2⌋ then
               Insert(Tree[Num].Right)
End
```

2.3.2 线段的删除

删除一条线段[c,d]

```
Procedure Delete(Num)
Begin
  If (c<Tree[Num].a)and(Tree[Num].b<d) then
      Tree[Num].Cover ← Tree[Num].Cover - 1
  Else
      If c<⌊(Tree[Num].a+Tree[Num].b)/2⌋ then
          Delete(Tree[Num].Left)
      If d>⌊(Tree[Num].a+Tree[Num].b)/2⌋ then
          Delete(Tree[Num].Right)
End
```

2.4 线段树的简单应用

掌握了线段树的建立，插入和删除这 3 条操作，就能用线段树解决一些最基本的统计问题了。例如给出一系列线段[a,b]（0<a<b<10000)覆盖在数轴上，然后求该数轴上共有多少个单位长度[k,k+1]被覆盖了。我们便可以在读入一系列线段[a,b]的时候，同时调用过程 Insert(1)。等所有线段都插入完后，就可以进行统计了：

```
Procedure Count(Num)
Begin
  If Tree[Num].Cover>0 then
      Number ← Number + (Tree[Num].b-Tree[NUM].a)
  Else
      Count(Tree[Num].Left)
      Count(Tree[Num].Right)
End
```

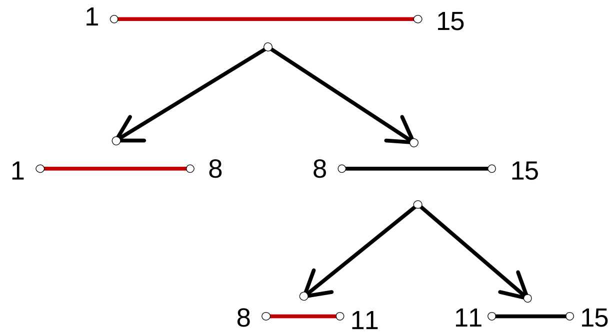
像这样的基本静态统计问题，线段树是可以很方便快捷地解决的。但是我们会留意到，如果处理一些动态统计问题，比如说一些需要用到删除和修改的统计，困难就出现了。

『例 1』在数轴上进行一系列操作。每次操作有两种类型，一种是在线段[a,b]上涂上颜色，另一种将[a,b]上的颜色擦去。问经过一系列的操作后，有多少条单位线段[k,k+1]被涂上了颜色。

这时我们就面临了一个问题——线段的删除。但线段树中线段的删除只能是把已经放入的线段删掉，例如我们没有放置[3,6]这条线段，删除[3,6]就是无法做到的了。而这道题目则不同，例如在[1,15]上涂了颜色，我们可以把[4,9]上的颜色擦去，但线段树中只是插入了[1,15]这条线段，要删除[4,9]这条线段显然是做不到的。因此，我们有必要对线段树进行改进。

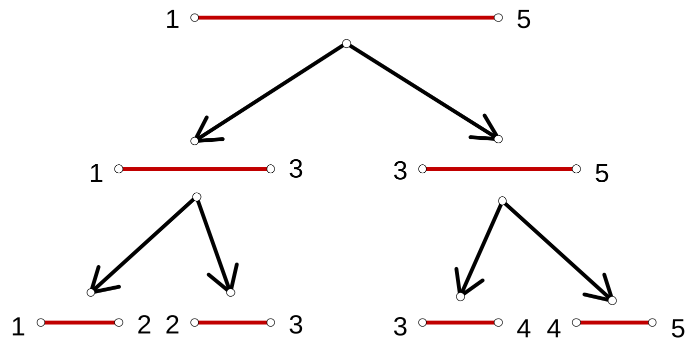
2.5 线段树的改进

用回刚刚那个例子。给[1,15]涂上色后，再把[4,9]的颜色擦去。很明显[1,15]这条线段已经不复存在，只剩下[1,4]和[9,15]，所以我们必须对线段树进行修改，才能使它符合改变了的现实。我们不难想到把[1,15]这条线段删去，再插入线段[1,4]和[9,15]。但事实上并非如此简单。如下图



若先前我们已经插入了线段[8,11]，[1,8]。按上面的做法，只把[1,15]删去，然后插入[1,4]，[9,15]的话，[1,8]，[8,11]这两条线段并没有删去，但明显与实际不符了。于是[1,8]，[8,11]也要修改。这时疑问就来了。若以线段[1,15]为根的整棵线段树中的所有结点之前都已经插入过，即我们曾经这样涂过颜色：[1,2],[2,3]，……，[14,15],[1,3],[3,5],……,[13,15],[1,5]，…………，[1,15]。然后把[1,15]上的颜色擦去。那么整个线段树中的所有结点的状态就都与实际不符了，全都需要修改。修改的复杂度就是线段树的结点数，即 2*(15-1)=28。如果不是[1,15]这样的小线段，而是[1,30000]这样的线段，一个擦除动作就需要 O(59998)的复杂度去修改，显然效率十分低（比直接模拟的 O(30000)还差）。

为了解决这个问题，我们给线段树的每一个结点增加一个标记域（以下用 bj 来表示标记域）。增加一个标记域有什么用呢？如下图：



以[1,5]为根的整棵线段树的全部结点都已涂色。现把[1,5]上的颜色擦去。则整棵线段树的结点的状态都与实际不符了。可是我们并不一定要对所有结点都进行修改，因为有些结点以后可能根本不会有被用到的时候。例如我们做完擦去[1,5]的操作之后，只是想询问[3,5]是否有涂上颜色。那么我们对[1,2]，[2,3],[1,3],[3,4],[4,5]等线段的修改就变成无用功了。为了避免无用功的出现，我们引入标记域 bj。具体操作如下：

1、擦去线段[a,b]之后，给它的左儿子和右儿子都做上标记，令它们的 bj=-1。
2、每次访问一条线段，首先检查它是否被标记，若其 bj=-1，则进行如下操作：

① 将该线段的状态改为未被覆盖，并把该线段设为未被标记，bj=0。
② 把该线段的左右儿子都设为被标记，bj=-1。

对线段[1,5]进行了这样的操作后就不需要对整棵线段树都进行修改了。原理很简单。以线段[3,4]为例。若以后有必要访问[3,4]，则必然先访问到它的父亲[3,5]，而[3,5]的 bj=-1，因此进行①、②的操作后，[3,5]的状态变为未被覆盖，并且把他的标记传递给了他的儿子——[3,4]和[4,5]。接着访问[3,4]的时候，它的bj=-1，我们又把[3,4]的状态变为未被覆盖。可见，标记会顺着访问[3,4]的路一直传递到[3,4]，使得我们知道要对[3,4]的状态进行修改，避免了错误的产生。同时，当我们需要用到[3,4]的时候才会进行修改，如果根本不需要用它，修不修改都无所谓了，并不会影响程序的正确性。因此这种方法在保持了正确性的同时有避免了无意义的操作，提高了程序的效率。

进行标记更新的代码如下：

```
Procedure Clear(Num)
Begin
   Tree[Num].Cover ← 0
   Tree[Num].bj ← 0
   Tree[Tree[Num].Left].bj ← -1
   Tree[Tree[Num].Right].bj ← -1
End
```

每次访问线段[a,b]之前，首先检查它是否被标记，如果是则调用过程 Clear 进行状态修改。这样做只是在访问的时候顺便进行修改，复杂度是 O(1)，程序效率依然很高。

于是，引入标记域后，本题中插入和删除的过程大致如下：

插入过程 Insert

1、若该线段被标记，则调用 Clear 过程

2、若线段状态为被涂色，则退出过程（线段已被涂色，无需再插入它或它的子线段）

3、若涂色的区域覆盖了该线段，则该线段的状态变为被涂色，并退出过程

4、若涂色的区域与该线段的左半截的交集非空，则调用左儿子的插入过程

5、若涂色的区域与该线段的右半截的交集非空，则调用右儿子的插入过程

删除过程 Delete

1、若该线段被标记，则退出过程（该线段已被赋予被擦除的“义务”，无需再次赋予）

2、若擦除的区域覆盖了该线段，则该线段的状态变为未被涂色，并将其左右儿子都做上标记，退出过程

3、若该线段的状态为被涂色，则
① 该线段状态变为未被涂色
② 将其左右儿子做上标记
③ 插入线段[a,c]和[d,b]

线段[a,b]状态为被涂色，而擦除[c,d]相当于把[a,b]整段擦除，再插入[a,c]（若 a<c）和[d,b]（若 d<b）

4、若该线段的状态为未被涂色，则

①若擦除区域与该线段的左半截的交集非空，则调用左儿子的擦除过程
②若擦除区域与该线段的右半截的交集非空，则调用右儿子的擦除过程
｛程序请参见**附录**｝

归纳一下标记域的思想及如何使用。***<u>如果我们对整条线段[a,b]进行操作的话</u>***，我们就可以只是给[a,b]的左右儿子做上标记，而无需对以[a,b]为根的整棵子树中的所有结点进行修改。原理就是对下面的所有结点[c,d]，都有[c,d]⊂[a,b]，因此[a,b]状态的改变也就代表了[c,d]状态的改变。

本着这个思想，标记域的使用形式并不是固定的，而是多样的，具体形式如何要视题目而定，但只要理解了它的思想，总能想到如何确定作标记的方式，维持线段树的高效。例如下面这一题：

『例 2』Byteotian 州铁道部决定赶上时代，为此他们引进了城市联网。假设城市联网顺次连接着 n 个城市，从 1 到 n 编号(起始城市编号为 1，终止城市编号为 n)。每辆火车有 m 个座位且在任何两个车站之间运送更多的乘客是不允许的。电脑系统将收到连续的预订请求并决定是否满足他们的请求。当火车在被请求的路段上有足够的空位时，就通过这个请求，否则不通过。通过请求的一部分是不允许的。通过一个请求之后，火车里的空位数目将得到更新。请求应按照收到的顺序依次处理。

任务：计算哪些请求可以通过，哪些请求不能通过。

输入文件

第一行是三个被空格隔开整数 n, m 和 r ($1<=n<=60\ 000$, $1<=m<=60\ 000$, $1<=r<=60\ 000$)。数字分别表示：铁路上的城市个数，火车内的座位数，请求的数目。接下来 r 行是连窜的请求。第 i+1 行描述第 i 个请求。描述包含三个整数 $k1$、$k2$ 和 v ($1<=k1<k2<=n$, $1<=v<=m$)。它们分别表示起点车站的编号，目标车站的编号，座位的需求数。

输出文件

输出 r 行，每行一个字符。’T’表示可以通过；’F’表示不能通过。

样例输入

4 6 4
1 4 2
1 3 2
2 4 3
1 2 3

样例输出

T
T
N
N

这道题需要判断请求是否能被满足，即涉及到线段状态的询问。当要求被满足的时候要减去相应线段上的座位数，因此涉及到线段的动态修改。于是我们同样可以引入标记域来维持动态修改算法的高效。

由于该题的特点在于询问上。为了询问能够高效，我们用 Seat 这个域来记录线段[a,b]上的座位数，因此在建立线段树的时候，所有节点的 Seat 初始状态为 *m*。为了能实时反映线段[a,b]上剩下的座位数，当[a,b]的左儿子或右儿子的状态发生改变时，我们通过[a,b].Seat = Min{左儿子.Seat , 右儿子.Seat}来对线段[a,b]的座位数进行更新，即取[a,b]整条线段上所有座位数中的最小值作为线段[a,b]的座位数。

对于线段状态的修改，仍然可以引入标记域。如果我们要把结点[a,b]上的座位数都减去 v，就满足了***对整条线段[a,b]进行操作***的要求，因此可以将其左右儿子的 bj 都减去 v。而在访问每条线段前先检查其标记是否为 0，若不为 0，则执行 Clear 过程进行更新。

```
Procedure Clear(Num)
Begin
   Tree[Num].Seat ←Tree[Num].Seat + Tree[Num].bj
   Tree[Tree[Num].Left].bj ←Tree[Tree[Num].Left].bj + Tree[Num].bj
   Tree[Tree[Num].Right].bj ←Tree[Tree[Num].Right].bj + Tree[Num].bj
   Tree[Num].bj ← 0
End
```

判断请求能否通过的函数以及座位状态修改的过程如下：

判断请求能否通过的函数 Can(返回值为布尔类型**)**

1、若线段标记不为 0，执行 Clear 过程进行更新

2、若请求区域覆盖了该线段，则：

　　若座位数大于等于要求值，返回 True，否则返回 False

3、若请求区域跨越该线段的中点，则返回值 = **Can**(左儿子) ∩ **Can**(右儿子)

4、若请求区域在该线段中点的左方，则返回值 = **Can**(左儿子)

5、若请求区域在该线段中点的右方，则返回值 = **Can**(右儿子)

座位状态修改的过程 Delete

1、若线段标记不为 0，执行 Clear 过程进行更新

2、若请求区域覆盖了该线段，则将该线段的座位数减去请求的座位数目 v，并将左右儿子的标记 bj 都减去 v。退出过程。

3、若请求区域与该线段左半截有交集，则调用左儿子的 Delete 过程

　　否则调用左儿子的 Clear 过程进行更新

4、若请求区域与该线段右半截有交集，则调用右儿子的 Delete 过程

　　否则调用右儿子的 Clear 过程进行更新

5、取左右儿子的座位数中较小的那个作为该线段的座位数

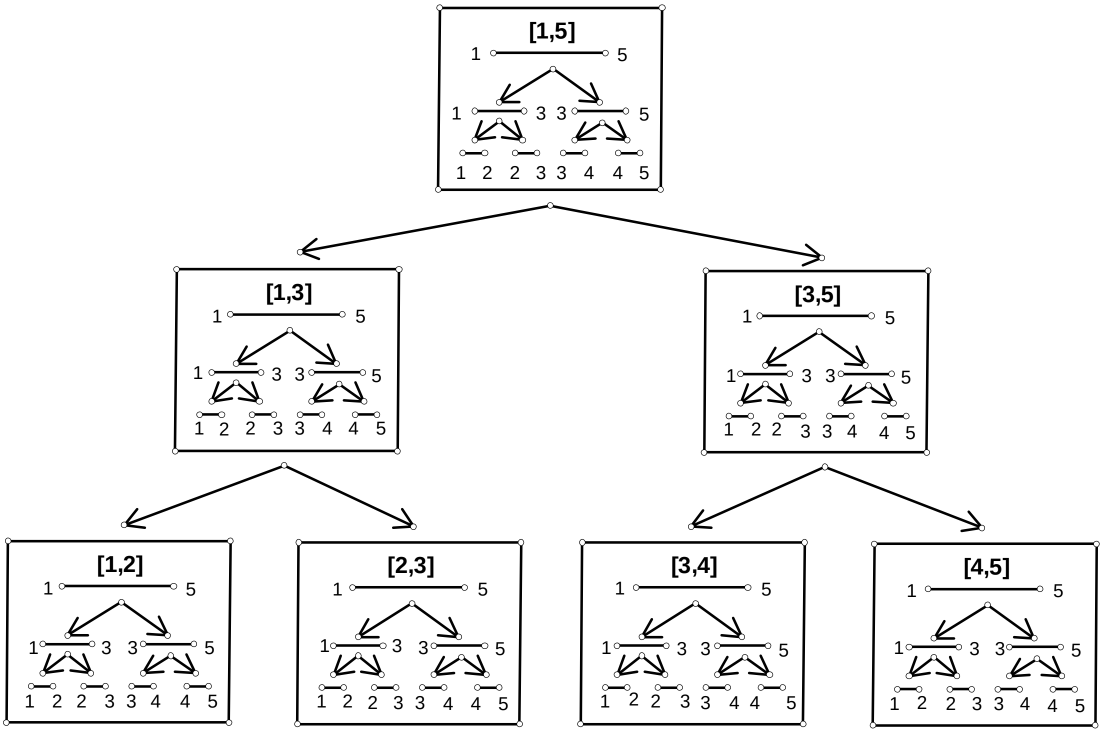
> 调用 Clear 过程，是因为第 5 步中进行座位数更新时需要用到左右儿子的座位数。而左右儿子的座位数并不一定符合实际情况（即它们的 bj 可能不为 0），不更新就有可能产生错误。

{程序请参见**附录**}

2.6 线段树的推广

线段树处理的是线性统计问题，而我们往往会遇到一些平面统计问题和空间统计问题。因此我们需要推广线段树，使它变成能解决平面问题的“矩形树”和能解决空间问题的“方块树”。

将一维线段树改成二维线段树，有两种方法。一种就是给原来线段树中的每个结点都加多一棵线段树，即“树中有树”。如下图：



例如在主线段树的结点[1,3]中，线段[3,5]表示的就是矩形(1,3,3,5) **{注：本文用(x1,y1,x2,y2)表示左下角顶点坐标为(x1,y1),右上角顶点坐标为(x2,y2)的矩形}**。容易算出，用这种方法构造一棵矩形(x1,y1,x2,y2)的线段树需要的空间为 $O((2\times(x2-x1)-1)\times(2\times(y2-y1)-1))$，即空间复杂度为 $O(Long\_x\times Long\_y)$，其中 Long_x，Long_y 分别表示矩形的长和宽。相应地，时间复杂度为 $O(n\times Log_2(Long\_x)\times Log_2(Long\_y))$。其中 n 为操作数。由于这种线段树有两层，处理起来较麻烦。

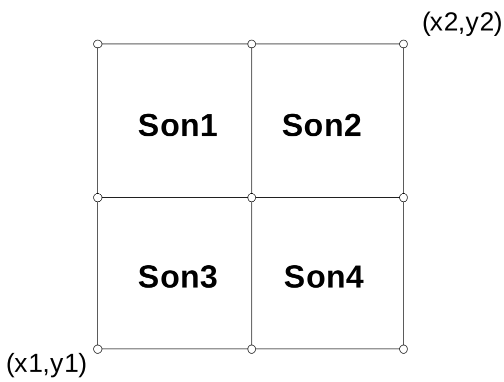
另一种方式是直接将原来线段树结点中的线段变成矩形。即每个结点代表一个矩形。因此矩形树用的是四分的思想，每个矩形分割为 4 个子矩形。矩形(x1,y1,x2,y2)的 4 个儿子分别为

$(x1,y1,\lfloor (x1+x2)/2\rfloor,\lfloor (y1+y2)/2\rfloor)$、

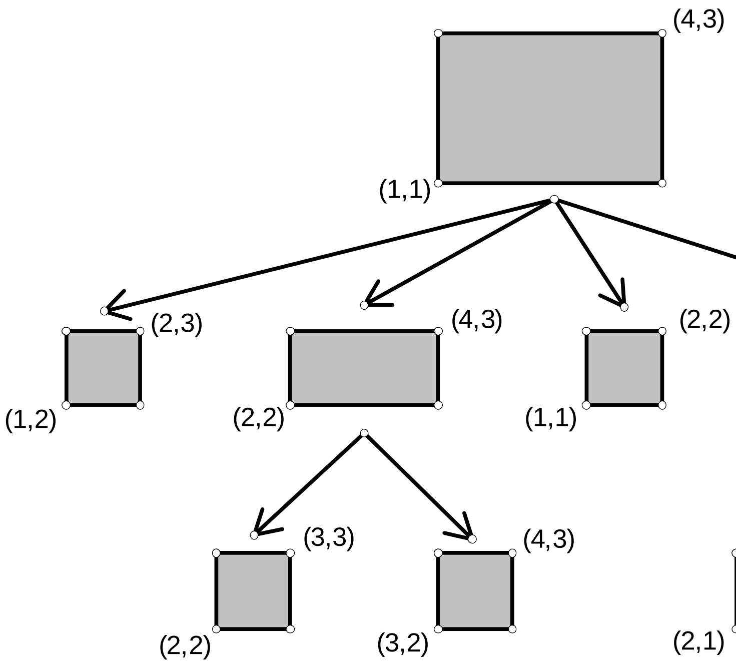
$(\lfloor (x1+x2)/2\rfloor,y1,x2,\lfloor (y1+y2)/2\rfloor)$、

$(x1,\lfloor (y1+y2)/2\rfloor,\lfloor (x1+x2)/2\rfloor,y2)$、

$(\lfloor (x1+x2)/2\rfloor,\lfloor (y1+y2)/2\rfloor,x2,y2)$



例如下图就是一棵以矩形(1,1,4,3)为根的矩形树：



易知，以(x1,y1,x2,y2)为根的矩形树的空间复杂度也是O(Long_x×Long_y)。但由于它只有一层，处理起来比第一种方法方便。而且在这种矩形树中，标记思想依然适用。而第一种方法中，标号思想在主线段树上并不适用，只能在第二层线段树上使用。但是这种方法的时间复杂度可能会达到O(n×Long_x)。比起第一种来就差了不少。

对于多维的问题，第一种方法几乎不可能使用。因此我们可以仿照第二种方法。例如对于 n 维的问题。我们构造以$(a_1,a_2,a_3,\ldots,a_n,b_1,b_2,b_3,\ldots,b_n)$为根的线段树，其中$(a_1,a_2,a_3\ldots,a_n)$表示的是左下角的坐标，$(b_1,b_2,b_3,\ldots,b_n)$表示的是右上角的坐标。构造的时候用的就不是二分，四分了，而是 2^n 分，构造出一棵 2^n 叉树。结点的个数变为 $2^n\times(b_1-a_1)\times(b_2-a_2)\times\ldots\ldots\times(b_n-a_n)$。

2.7 线段树小结

作为解决统计类问题的利器，线段树在改进和推广之后，做到了高效地解决更多的问题。因其适用范围广和实现上的方便，线段树不失为一个优秀的方法。但线段树还是有一些缺陷的，下文将在与矩形切割进行比较的时候提及。

三、矩形切割

矩形切割是一种处理平面上矩形的统计的方法。许多统计类的问题通过数学建模后都能转化为用矩形切割来解决。矩形切割的原型是线段切割。我们先来看看线段切割的思想。

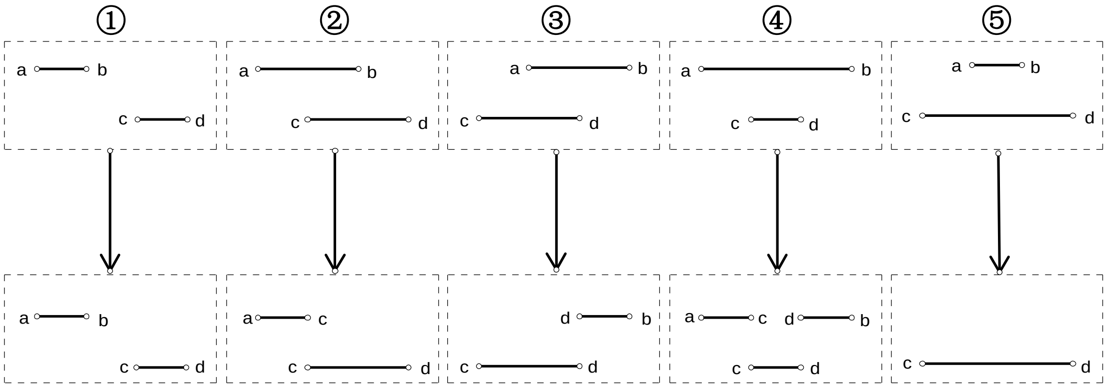
3.1 线段切割

用回『例 1』做例子。但是条件改变一下，就是涂色不一定是涂一种颜色，

而是可以涂多种颜色，同一线段上后涂的颜色会覆盖先涂的颜色。对每一种颜色都求出含有该种颜色的单位线段的条数。

题目要我们对每种颜色都求出被覆盖的单位线段的数目。如果所有的线段都是互不重叠的，那么我们只需把线段集合中同种颜色的所有线段的长度累加，就能得出该种颜色被覆盖的单位线段的数目了。但事实上线段之间会出现重叠的情况，因此我们引入线段切割的方法来对线段集合中的线段进行动态维护，使得所有线段两两不重叠。那么最后只需直接将线段的长度累加，就能得出答案。

其实线段切割的思想很简单。若线段集合中本来有一根线段[a,b]，现在加入一根新线段[c,d]。那么它们之间的位置关系可能有以下几种：



对于每一种位置关系，我们都可以通过切割线段[a,b]，并删除某些小段（因为这些小段已被[c,d]覆盖了），使得它与新线段[c,d]不重叠。

因此，当我们每次插入一条线段，就跟线段集合中的每一条线段[a,b]都判断一下是否出现重叠，若出现重叠则对[a,b]进行切割。判断重叠的方法为：若 a≥d 或者 c≥b，就不出现重叠，否则重叠。切割的方法就是：取线段[a,b]，[c,d]的交集[k1,k2]。若 a<k1，则加入线段[a,k1]；若 k2<b，则加入线段[k2,b]。删除线段[a,b]。

等全部线段插入并处理完后，由于所有的线段都不重叠，就能直接进行统计了。

3.1.1 线段的数据结构

```
Type Segment=Record
               a,b:Longint;
             End
```

通常可以增加一些域来描述线段的状态。如增加 Colour 域来表示线段的颜色。

3.1.2 判断线段相交的函数

```
Function Cross(a,b,c,d)
Begin
  If (.a>=d)or(c>=.b) then Cross ← false
  Else Cross ← true
End
```

3.1.3 切割线段的过程

```
Procedure Cut(Num,c,d)
Begin
   If Line[Num].a<c then Add(Line[Num].a,c)
   If d<Line[Num].b then Add(d,Line[Num].b)
   Delete(Num);
End
```

其中 Add 过程是将一条线段加到线段集合中的过程：

```
Procedure Add(a,b)
Begin
   tot ← tot + 1
   Line[tot].a ← a
   Line[tot].b ← b
End
```

其中 delete 过程是将一条线段删除的过程，可将线段集合中最后的一条线段移到要删除线段的位置上完成删除：

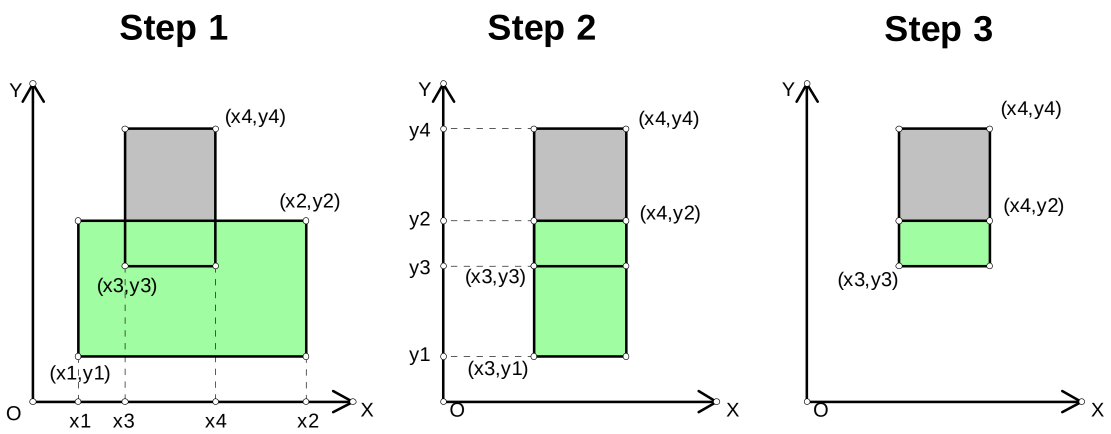
```
Procedure Delete(Num)
Begin
   Line[Num] ← Line[tot]
   tot ← tot -1
End
```

根据线段切割的思想，我们稍做推广，便能得出矩形切割的方法。

3.2 矩形切割

类似地，若矩形集合中已有矩形(x1,y1,x2,y2)，现加入矩形(x3,y3,x4,y4)。它们的位置关系可以有很多种（有 17 种之多），这里就不一一列举了。但无论它们的位置关系如何复杂，运用线段切割的思想来进行矩形切割，就会变得十分明了。

我们将矩形的切割正交分解，先进行 x 方向上的切割，再进行 y 方向的切割。以下图为例：



插入矩形(x3,y3,x4,y4)后，对矩形(x1,y1,x2,y2)进行切割。

Step 1:首先从 x 方向上切。把线段(x1,x2)切成(x1,x3)，(x4,x2)两条线段。于是相应地，我们就把两个矩形切了出来——(x1,y1,x3,y2)，(x4,y1,x2,y2)。把它们加到矩形集合中。去掉了这两个矩形后，我们要切的矩形就变为(x3,y1,x4,y2)。

Step 2:接着我们再进行 y 方向上的切割。把线段(y1,y2)切成(y1,y3)。相应地又得到一个矩形(x3,y1,x4,y2)。把它放入矩形集合。

Step 3:剩下的矩形为(x3,y3,x4,y2)，这个矩形已经被矩形(x3,y3,x4,y4)覆盖了，因此直接把它删掉。

我们可以归纳出**矩形切割的思想**：

1、先对被切割矩形进行 x 方向上的切割。取(x1,x2)，(x3,x4)的交集(k1,k2)

　① 若 x1<k1，则加入矩形(x1,y1,k1,y2)

　② 若 k2<x2，则加入矩形(k2,y1,x2,y2)

2、再对切剩的矩形***(k1,y1,k2,y2)*** 进行 y 方向上的切割。取(y1,y2)，(y3,y4)的交集(k3,k4)

　① 若 y1<k3，则加入矩形(***k1***,y1,***k2***,k3)

　② 若 k4<y2，则加入矩形(***k1***,k4,***k2***,y2)

3、把矩形(x1,y1,x2,y2)从矩形集合中删除。

切割过程的代码如下：

```
Procedure Cut(x1,y1,x2,y2,Direction)
Var k1,k2
Begin
  Case Direction of
    1:Begin
        k1 ← Max(x1,x3)   {计算线段(x1,x2)，(x3,x4)交集的左边界}
        k2 ← Min(x2,x4)   {计算线段(x1,x2)，(x3,x4)交集的右边界}
        if x1<k1 then Add(x1,y1,k1,y2)
        if k2<x2 then Add(k2,y1,x2,y2)
        Cut(k1,y1,k2,y2,Direction+1)
     End
    2:Begin
        k1 ← Max(y1,y3)
        k2 ← Min(y2,y4)
        if y1<k1 then Add(x1,y1,x2,k1)
        if k2<y2 then Add(x1,k2,x2,y2)
     End
  End
End
```

其中 **Add** 是加入矩形的过程。

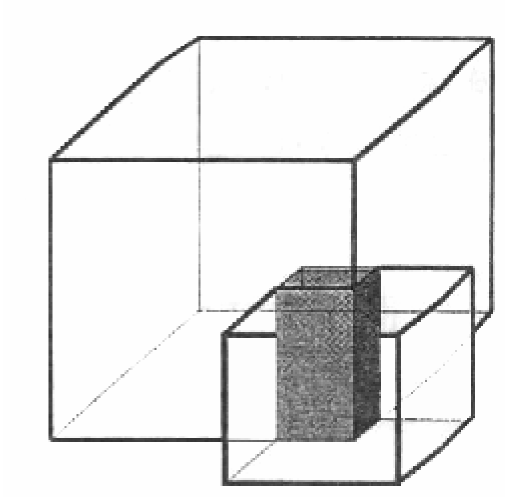
3.3 矩形切割的推广

本着矩形切割的思想，我们可以把矩形切割推广为立方体切割，甚至推广到

n 维空间中的切割。两个 n 维物体有重叠部分的充要条件就是***它们在 n 个方向上都存在交集***。就是说(x1,x2)和(x3,x4)有交集；(y1,y2)和(y3,y4)有交集；……。切割的方法也是类似的：先在 x 方向上切，然后在 y 方向上切，接着在 z 方向上切，……，一直到在第 n 个方向上切。

当 n 变大的时候，如果用这种方法来写程序，将会显得很复杂，甚至变得不可能。我们可以做些改动来简化代码，将一个 n 维"物体"用两个数组表示出来(a[1],a[2],a[3],……,a[n],b[1],b[2],b[3],……,b[n])。然后相应地改动一下 Add 过程，就可以不用分类讨论，直接改成一重循环，只需几行就能完成。由于比较简单，这里就不写出来了。

『例 3』卫星覆盖 [①]



卫星可以覆盖空间直角坐标系中一定大小的立方体空间，卫星处于该立方体的中心。其中(x,y,z)为立方体的中心点坐标，r 为此中心点到立方体各个面的距离（即 r 为立方体高的一半）。立方体的各条边均平行于相应的坐标轴。我们可以用一个四元组(x,y,z,r)描述一颗卫星的状态，它所能覆盖的空间体积$V=(2r)^3=8r^3$。

由于一颗卫星所能覆盖的空间体积是有限的，因此空间中可能有若干颗卫星协同工作。它们所覆盖的空间区域可能有重叠的地方，如下图所示（阴影部分表示重叠的区域）。

写一个程序，根据给定的卫星分布情况，计算它们所覆盖的总体积。

输入输出

输入文件是 Cover.in。文件的第一行是一个正整数 N（1<=N<=10O）：表示空间中的卫星总数。接下来的 N 行每行给出了一颗卫星的状态，用空格隔开的四个正整数 x,y,z,r 依次表示了该卫星所能覆盖的立方体空间的中心点坐标和半高，其中-1000<=x,y,z<=1000, 1<=r<=200。

输出文件是 Cover.out。文件只有一行,包括一个正整数，表示所有这些卫星所覆盖的空间总体积。

样例

Cover.in

3

0 0 0 3

1 –1 0 1

19 3 5 6

Cover.out

① NOI' 97 第二试第三题

1944

这题可以用立方体切割来做，思想也是一样，每读入一个立方体(x3,y3,z3,x4,y4,z4)，就和已有的立方体(x1,y1,z1,x2,y2,z2)判断是否有重叠，有的话就进行切割。所有的数据处理完后就可以将全部立方体的体积加起来，就能得出答案了。

应该注意的是新切割生成的立方体与立方体(x3,y3,z3,x4,y4,z4)是不会有重叠部分的。因此我们在读入矩形(x3,y3,z3,x4,y4,z4)之前，先把当前立方体集合中的立方体总数 tot 记录起来 tot1 ← tot，那么循环判断立方体重叠只需循环到 tot1 就行了，新生成的立方体就无需与立方体(x3,y3,z3,x4,y4,z4)判断是否重叠了。这样可以节省不少时间。

具体细节就不说了，程序参见**附录**

3.4 矩形切割的应用

如果我们引入矩形切割单单就是为了切矩形，那就没多大意义了，毕竟这样的题目不多见。其实矩形切割作为一个数学模型，常常可以在许多统计类的问题中使用。例如下面这题：

『例 4』War Field Statistical System [②]

2050 年，人类与外星人之间的战争已趋于白热化。就在这时，人类发明出一种超级武器，这种武器能够同时对相邻的多个目标进行攻击。凡是防御力小于或等于这种武器攻击力的外星人遭到它的攻击，就会被消灭。然而，拥有超级武器是远远不够的，人们还需要一个战地统计系统时刻反馈外星人部队的信息。这个艰巨的任务落在你的身上。请你尽快设计出这样一套系统。

这套系统需要具备能够处理如下 2 类信息的能力：

1、外星人向[x1，x2]内的每个位置增援一支防御力为 *v* 的部队。

2、人类使用超级武器对[x1，x2]内的所有位置进行一次攻击力为 *v* 的打击。系统需要返回在这次攻击中被消灭的外星人个数。

（注：防御力为 i 的外星人部队由 *i* 个外星人组成，其中第 *j* 个外星人的防御力为 *j*。）

输入格式

从文件 War.in 第一行读入 *n*，*m*。其中 *n* 表示有 *n* 个位置，*m* 表示有 *m* 条信息。以下有 *m* 行，每行有 4 个整数 *k*，*x1*，*x2*，*v* 用来描述一条信息 。*k* 表示这条信息属于第 *k* 类。*x1*，*x2*，*v* 为相应信息的参数。*k*=1 or 2。

注：你可以认为最初的所有位置都没有外星人存在。

规模：**0<*n*<=30000；0<*x1*<=*x2*<=*n*；0<*v*<=30000；0<*m*<=2000**

输出格式

结果输出到文件 War.out。按顺序输出需要返回的信息。

[②] NOI2003 前 OIBH 某次网上比赛试题

输入样例　　对应输出

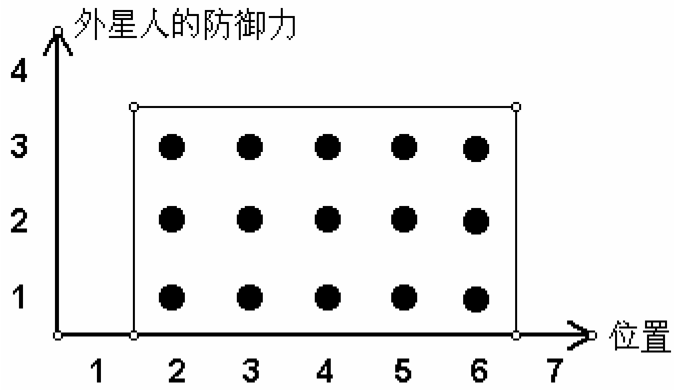
3 5	无
1 1 3 4	无
2 1 2 3	6
1 1 2 2	无
1 2 3 1	无
2 2 3 5	9

输出样例

6
9

这道题看上去与矩形切割好像并无多大关系。但仔细分析一下，这题可以转化为用矩形切割模型来解决。每次向[x1,x2]增添一支防御力为 v 的部队，因为每支防御力为 v 的部队是由 v 个外星人组成的，防御力依次为 1,2,3，……,v。如果我们在平面直角坐标系上来表示这种操作，则有：

若在位置[2,6]上增加一支防御力为 3 的部队，那么情况就如右图所示，其实等于加入了一个矩形(1,0,6,3)。因此在[x1,x2]上增添一支防御力为 v 的部队就等于增添一个矩形(x1-1,0,x2,v)。同理，在[x1,x2]上使用攻击力为 v 的武器，就等于把与矩形(x1-1,0,x2,v)有重叠部分的矩形都进行切割。所以这道题就变成了简单的矩形切割问题。



由于这道题要我们求的是每次使用武器所杀死的外星人数目。因此我们可以相应地根据这个改动一下做法。在增加部队，即插入矩形(x3,y3,x4,y4)的时候，并不需要与矩形集合中的矩形(x1,y1,x2,y2)判断是否重叠，因为对于这道题来说，重叠是没有关系的（一个格子可以站多个具有同样防御力的外星人）。而在使用武器的时候，我们像往常一样切割矩形，只是顺便做做统计罢了。

{程序请参见**附录。**}

由此我们可以看到，矩形切割并不是只是局限于解决几何类的问题，只要我们将题目数学建模后能运用矩形切割的思想，那么矩形切割也不失为一个好方法。

四、线段树与矩形切割的比较

同为解决动态统计问题利刃的线段树与矩形切割，区别不少。为了更快捷，更完美地解决问题，什么时候使用线段树较好，什么时候使用矩形切割更优，的确值得我们研究研究。

对两种方法进行比较，我们可以先从复杂度入手，毕竟这个要素是我们决定是否使用一种方法的决定性因素。

4.1 线段树的时空复杂度

线段树的时空复杂度在前面已经做了介绍。

4.1.1 线段树的空间复杂度

1. 线段树的空间复杂度是 O(Long_x)，其中 Long_x 为最长线段的长度。

2. 二维线段树是 O(Long_x×Long_y)，其中 Long_x，Long_y 分别为最大矩形的长、宽。

3. 三维线段树是 O(Long_x×Long_y×Long_z)，其中 Long_x，Long_y，Long_z 分别为最大方块的长、宽、高。

4.1.2 线段树的时间复杂度

1. 线段树的时间复杂度是 $O(n\times \mathrm{Log}_2(\mathrm{Long\_x}))$。

2. 二维线段树是 $O(n\times \mathrm{Log}_2(\mathrm{Long\_x})\times \mathrm{Log}_2(\mathrm{Long\_y}))$。

3. 三维线段树是 $O(n\times \mathrm{Log}_2(\mathrm{Long\_x})\times \mathrm{Log}_2(\mathrm{Long\_y})\times \mathrm{Log}_2(\mathrm{Long\_z}))$。

4.2 矩形切割的时空复杂度

矩形切割的时间复杂度是较浅显的。我们每次读入一个矩形，就必须跟矩形集合中的所有矩形进行比较，看看是否出现重叠。因此时间复杂度是 O(m*n)。其中 m 表示数据中的矩形数目，n 表示矩形集合中矩形数目。然而矩形集合中的矩形数目是会改变的，n 应该是矩形集合中矩形数目的峰值（即曾经在矩形集合中出现的矩形数目的最大值）。关于该峰值 n 的计算，就是计算空间复杂度的问题了。

4.2.1 矩形切割的空间复杂度

矩形切割的空间复杂度是由峰值 n 决定的，最多会出现多少个矩形，我们就开多大的数组。而 n 的计算却十分困难。因为在平面内放置矩形的情况不一样，切割出来的矩形个数和状况也就会不同。为此，我们可以先做一些数据，数据中的矩形是随机生成的。看看当数据中的矩形个数为 m 的时候，峰值 n 究竟会是多少。请看下表：

表 1

矩形数 m	100	500	1000	5000	10000	50000	100000
峰值 n	239	479	680	1015	1296	1741	2092

这些数据中的矩形是随机生成的。其中 m 是输入数据中的矩形个数。对于数据中的每个矩形(x1,y1,x2,y2)都有：0<=x1<x2<=60000，0<=y1<y2<=60000。其中对矩形集合中矩形数目的峰值 n，计算方法是：对同一个 m 值，生成 10 组数据，得出 10 个 n 值。取这些结果的平均值作为 n 的值。

在矩形个数较小的时候，如 m=100，峰值 n 达到了 239，是 m 的两倍多。但随着 m 增大的加快，峰值 n 的增加却比较缓慢。在 m=5000 时，n＝1015。m=10000 时，n=1296。相差不多。可见 n 与 m 并非是正比关系。也就是说，即使矩形个数猛增，矩形集合中矩形数目的最大值只是维持在一个较低的水平。因此对随机数据而言，空间复杂度是很小的。

然而这仅仅是对于随机数据。究竟在构造出来的数据中，峰值 n 可以达到多少呢？我们尝试构造这样的一种数据：

数据一共有 m 个矩形。前 k 个矩形如此放置：第一个矩形(1,1,x,y)是最大的。（其中应使 x,y 的值足够大，例如 x=y=100000000）。以后的每一个矩形都比前一个矩形缩小一点点。即如果前一个矩形是(x1,y1,x2,y2)，则下一个矩形为(x1+1,y1+1,x2-1,y2-1)。例如第 2 个矩形就是(2,2,x-1,y-1)，第 3 个矩形就是(3,3,x-2,y-2)。由于后一个矩形被前一个矩形完全覆盖，且没有边重叠，因此第 t+1 个矩形会将第 t 个矩形切割成 4 块。放入 k 个矩形之后，就有 4*(k-1)+1＝4k-3 个矩形了。例如**图 1** 就是 k=4 时的情况，共有 13 个矩形。之后的 m-k 个矩形如此放置：因为前面 k 个矩形中最后一个矩形是(k,k,x-k,y-k)，所以现在放一些这样的矩形：(k+1,0,k+2,y+1)，(k+3,0,k+4,y+1)，(k+5,0,k+6,y+1)，……。如**图 2** 所示。每放置这样的一个矩形，就会与 2k-1 个矩形产生重叠，切割后多出 2k-1 个矩形。又因为我们放置第一个矩形(1,1,x,y)的时候已假设 x,y 足够大，因此后 m-k 个矩形都能与 2k-1 个矩形发生重叠。因此会多出 $(m-k)\times(2k-1)=-2k^2+(2m+1)k-m$ 个矩形。加上先前的 4k-3 个矩形一共是 $-2k^2+(2m+5)k-(m+3)$ 个矩形。利用二次函数求最值可知当 $k=\frac{2m+5}{4}$ 时函数取到最大值 $\frac{m^2}{2}-\frac{3m}{2}+\frac{1}{8}$。空间复杂度达到了 $O(m^2)$！这样的复杂度显然是致命的。

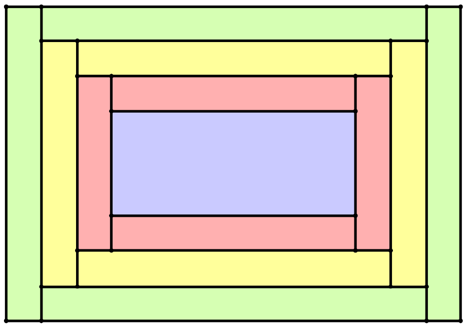


图1

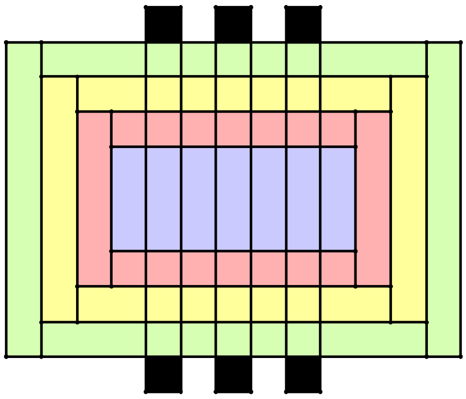


图2

可见，如果针对矩形切割算法的弱点刻意构造数据，复杂度将高到无法承受。如果并非针对性地构造极端数据，由表 1 的结果可以看出矩形切割还是很优秀的。

4.2.2 矩形切割的时间复杂度

知道了矩形切割的空间复杂度，时间复杂度就好办了。前面已经说了，时间

复杂度是 $O(m\times n)$。根据表 1，若是随机数据。在 m＝10000 的时候，n=1296。还是可以接受的。而当 m=50000 时，n=1741，就十分勉强了。而对于极端数据。时间复杂度是 $O(m^3)$，在 500 个矩形的时候就已经需要 1 亿多次的运算，效率是很低的。

4.3 线段树与矩形切割适用范围的比较

根据线段树和矩形切割的复杂度。我们就可以思考出它们的适用范围。

线段树的空间复杂度是固定的，即 $O(Long\_x)$。若其中的 Long_x 很大，即线段的端点取值范围很大，线段树的空间复杂度将会十分大甚至无法承受。特别是在矩形树和方块树中。例如方块的长宽高限定在 1000 以下。空间复杂度就已经达到了 $8*10^9$。根本无法承受。相对来说，矩形切割在这方面就十分有优势了。它存储一个矩形只需 4 个域，一个方块也只需 6 个域，完全不受 Long_x,Long_y 等边界的限制。矩形有多大对矩形切割的复杂度是没有影响的。例如例 3 中的边界范围限制-1000<=x,y,z<=1000 和例 4 中的 0<n<=30000；0<x1<=x2<=n；0<v<=30000 都决定了线段树是无法承受这种空间复杂度的。而对于矩形切割来说却是不在话下。

线段树的时间复杂度很小，只有 $O(n\times Log_2(Long\_x))$，因此对于操作数较多的题目线段树可以做到得心应手，效率很高。然而操作数一多，矩形切割的效率就不高了。而例 3 中立方体的数目最多才 100 个，因此就这题而言用矩形切割来做就显得十分优秀了。

在编程复杂度上，线段树和矩形切割都是很容易就能实现的。

因此我们可以得出结论：***<u>对边界范围小，操作数多的题目，我们选择线段树；对边界范围大，操作数少的题目，我们选择矩形切割。</u>***

五、总结

到此，我们已较深入地了解到线段树和矩形切割的方方面面。经过对它们的思想，基本操作，改进和推广等方面的思考与研究，我们更清晰地体会到了这两把解决统计问题的利刃。在对它们进行复杂度，优缺点，适用范围等各方面的比较的过程中，我们总结出了什么时候该用哪个方法，积累了一定的经验，也为以后更好地解决该类问题打下了一定的基础。

毕竟文章篇幅有限，本文也只是仅仅介绍了两个方法，涉及的范围并不广。但我想，发现和提出问题，思考并解决问题这种能力是无论在哪个领域哪个方面都应该提倡的。若本文能在为大家介绍了两种方法的同时，也能引起大家对这一方面的重视，激发大家对统计类问题更深入的研究和对其它问题更广泛的思考，我的目的就达到了。

【参考文献】

1、NOI97 试题

2、OIBH 网上比赛试题

【附录】

附录 1：例 1 的线段树程序　｛Sample1.pas｝

```
Program Sample1;
Const Maxn=120000;  ｛最多支持 120000 条线段，即支持 Long_x<=60000｝
Type TreeNode=Record
             a,b,Left,Right:Longint;
             Cover,bj:shortint; ｛记录线段是否被覆盖；线段的标记｝
           End;
Var i,j,k,m,n,tot,c,d,Ans:longint;
   Tree:array[0..Maxn] of TreeNode;
Procedure MakeTree(a,b:Longint); ｛建立线段树的过程｝
Var Now:longint;
Begin
  inc(tot);
  Now:=tot;
  Tree[Now].a:=a;
  Tree[Now].b:=b;
  if a+1<b then
  begin
    Tree[Now].Left:=tot+1;
    MakeTree(a,(a+b) shr 1);
    Tree[Now].Right:=tot+1;
    MakeTree((a+b) shr 1,b);
  end;
End;
Procedure Init;  ｛读入数据并预处理的过程｝
Begin
  assign(input,'sample1.in');
  reset(input);
  assign(output,'sample1.out');
  rewrite(output);
  readln(n,m);
  fillchar(Tree,sizeof(Tree),0);
  tot:=0;
  MakeTree(1,n);
End;
Procedure Clean(Num:Longint);  ｛更新标记的过程｝
Begin
  Tree[Num].Cover:=0;
  Tree[Num].bj:=0;
  Tree[Tree[Num].Left].bj:=-1;
```

```
  Tree[Tree[Num].Right].bj:=-1;
End;
Procedure Insert(Num,c,d:longint);  {涂色的过程}
Var Mid:longint;
Begin
  if Tree[Num].bj=-1 then Clean(Num); {若被标记则更新}
  if Tree[Num].Cover=1 then exit; {若线段已被涂色，退出过程}
  if (c<=Tree[Num].a)and(d>=Tree[Num].b) then
  begin
    Tree[Num].Cover:=1;
    exit;
  end;
  Mid:=(Tree[Num].a+Tree[Num].b) shr 1;
  if c<Mid then Insert(Tree[Num].Left,c,d);
  if d>Mid then Insert(Tree[Num].Right,c,d);
End;
Procedure Delete(Num,c,d:Longint);  {擦除颜色的过程}
Var Mid:Longint;
Begin
  if Tree[Num].bj=-1 then Exit; {若线段被标记，说明该线段已不复存在，无需再进
                                                         行删除，退出过程}
  if (c<=Tree[Num].a)and(d>=Tree[Num].b) then
  begin
    Tree[Num].Cover:=0;
    Tree[Tree[Num].Left].bj:=-1;
    Tree[Tree[Num].Right].bj:=-1;{把线段已被删除的信息传给左右儿子}
    exit;
  end;
  if Tree[Num].Cover=1 then  {若该线段是被涂了色的}
  begin
    Tree[Num].Cover:=0;
    Tree[Tree[Num].Left].bj:=-1;
    Tree[Tree[Num].Right].bj:=-1;     {先删除}
    if Tree[Num].a<c then Insert(Num,Tree[Num].a,c); {再插入}
    if d<Tree[Num].b then Insert(Num,d,Tree[Num].b);
  end
  else  {否则继续对左右儿子调用删除过程}
  begin
    Mid:=(Tree[Num].a+Tree[Num].b) shr 1;
    if c<Mid then Delete(Tree[Num].Left,c,d);
    if d>Mid then Delete(Tree[Num].Right,c,d);
  end;
End;
Procedure Calculate(Num:longint);{计算被覆盖的单位线段条数的过程}
```

```
Begin
  if Num=0 then exit; {父亲已是叶子结点了（叶子节点的儿子为 0），返回}
  if Tree[Num].bj=-1 then exit; {线段被标记，说明已不存在，返回}
  if Tree[Num].Cover=1 then
  begin
    inc(Ans,Tree[Num].b-Tree[Num].a);
    exit;
  end;
  Calculate(Tree[Num].Left);
  Calculate(Tree[Num].Right);
End;
Procedure Main;   {主过程}
Var i,k:longint;
Begin
  for i:=1 to m do
  begin
    readln(k,c,d);
    if k=1 then Insert(1,c,d)
    else Delete(1,c,d);
  end;
  Ans:=0;
  Calculate(1);
  Writeln(Ans);
  Close(output);
End;
Begin   {主程序}
  Init;
  Main;
End.
```

附录 2：例 2 的程序 {Kol.pas}

```
Program Kol;
Const Maxn=12000;  {最多支持 120000 条线段}
Type TreeNode=Record
                Seat,bj,Left,Right,a,b:Longint; {Seat 为座位数}
              End;
Var i,j,k,m,n,r,tot,k1,k2,v:Longint;
    Time:Longint;
    Tree:array[0..Maxn] of TreeNode;
Procedure MakeTree(a,b:longint); {建立线段树的过程}
Var Now:longint;
Begin
  inc(tot);
```

```
  Now:=tot;
  Tree[Now].a:=a;
  Tree[Now].b:=b;
  Tree[Now].Seat:=m;  {每条线段的座位数初始值为 m}
  if a+1<b then
  begin
    Tree[Now].Left:=tot+1;
    MakeTree(a,(a+b) shr 1);
    Tree[Now].Right:=tot+1;
    MakeTree((a+b) shr 1,b);
  end;
End;
Procedure Clear(Num:Longint); {更新标记的过程}
Begin
  if Tree[Num].bj<>0 then
  begin
    inc(Tree[Num].Seat,Tree[Num].bj);
    inc(Tree[Tree[Num].Left].bj,Tree[Num].bj);
    inc(Tree[Tree[Num].Right].bj,Tree[Num].bj);
    Tree[Num].bj:=0;
  end;
End;
Function Can(Num,c,d,v:longint):boolean;{判断请求是否可行的过程}
Var Mid:longint;
Begin
  Clear(Num); {先进行标记更新}
  if (c<=Tree[Num].a)and(Tree[Num].b<=d) then
  begin
    if Tree[Num].Seat>=v then Can:=true
    else Can:=false;
    Exit;
  end;
  Mid:=(Tree[Num].a+Tree[Num].b) shr 1;
  if (c<Mid)and(d>Mid) then
  Can:=(Can(Tree[Num].Left,c,d,v))and
       (Can(Tree[Num].Right,c,d,v))  {必须左右儿子都能满足请求}
  else if c<Mid then Can:=Can(Tree[Num].Left,c,d,v)
  else Can:=Can(Tree[Num].Right,c,d,v);
End;
   Procedure Delete(Num,c,d,v:Longint);{确定请求能被满足后删除座位的过程}
Var Mid:longint;
Begin
  Clear(Num); {先进行标记更新}
  if (c<=Tree[Num].a)and(Tree[Num].b<=d) then
```

```
  Begin
    Tree[Num].Seat:=Tree[Num].Seat-v;
    dec(Tree[Tree[Num].Left].bj,v);  {把座位数减少了 v 个的信}
    dec(Tree[Tree[Num].Right].bj,v); {息传递给左儿子和右儿子}
    Exit;
  End;
  Mid:=(Tree[Num].a+Tree[Num].b) shr 1;
  if c<Mid then Delete(Tree[Num].Left,c,d,v)
  else Clear(Tree[Num].Left); {对左儿子的标记进行更新}
  if Mid<d then Delete(Tree[Num].Right,c,d,v)
  else Clear(Tree[Num].Right); {对右儿子的标记进行更新}
  if     Tree[Tree[Num].Left].Seat<Tree[Tree[Num].Right].Seat     then
{取左右儿子的座位数的较小者作为当前线段的座位数}
  Tree[Num].Seat:=Tree[Tree[Num].Left].Seat
  else Tree[Num].Seat:=Tree[Tree[Num].Right].Seat;
End;
Procedure Main;  {主过程}
Begin
  for i:=1 to r do
  Begin
    readln(k1,k2,v);
    if Can(1,k1,k2,v) then
    Begin
      Writeln('T');
      Delete(1,k1,k2,v);
    End
    else Writeln('N');
  End;
  Close(output);
End;
Procedure Init;  {读入数据并预处理的过程}
Begin
  assign(input,'kol.in');
  reset(input);
  assign(output,'kol.out');
  rewrite(output);
  readln(n,m,r);
  tot:=0;
  Fillchar(Tree,sizeof(Tree),0);
  MakeTree(1,n);
End;
Begin  {主程序}
  Init;
  Main;
```

```
End.
```

附录 3：例 3 的程序　{Cover.pas}

```
Program Cover;
Const Maxn=10000;  {立方体集合最多能容纳 10000 个立方体}
Type Blocks=Record
             x1,y1,z1,x2,y2,z2:Longint; {描述方块的 6 个域}
           End;
Var i,j,k,m,n,x,y,z,r,tot:longint;
   Cubic:array[1..Maxn] of Blocks;
   Now:Blocks;
Procedure Init;  {读入数据并预处理的过程}
Begin
  assign(input,'cover.in');
  reset(input);
  assign(output,'cover.out');
  rewrite(output);
  readln(n);
  Fillchar(Cubic,sizeof(Cubic),0);
  tot:=0;
End;
Function Max(a,b:Longint):Longint; {比较两个数并返回较大者的函数}
Begin
  if a>b then Max:=a
  else Max:=b;
End;
Function Min(a,b:Longint):Longint; {比较两个数并返回较小者的函数}
Begin
  if a<b then Min:=a
  else Min:=b;
End;
Procedure Add(x1,y1,z1,x2,y2,z2:Longint);{加入立方体的过程}
Begin
  inc(tot);
  Cubic[tot].x1:=x1; Cubic[tot].y1:=y1; Cubic[tot].z1:=z1;
  Cubic[tot].x2:=x2; Cubic[tot].y2:=y2; Cubic[tot].z2:=z2;
End;
Procedure Cut(x1,y1,z1,x2,y2,z2,Direction:Longint);{切割}
Var k1,k2:longint;
Begin
  Case Direction of
   1:Begin {先在 x 方向切}
       k1:=Max(x1,Now.x1);  {计算线段[x1,x2],[Now.x1,Now.x2]}
```

```
        k2:=Min(x2,Now.x2);  {的交集[k1,k2]}
        if x1<k1 then Add(x1,y1,z1,k1,y2,z2);
        if k2<x2 then Add(k2,y1,z1,x2,y2,z2);
        Cut(k1,y1,z1,k2,y2,z2,Direction+1); {调用 y 方向的切割}
      End;
    2:Begin {然后在 y 方向切}
        k1:=Max(y1,Now.y1);
        k2:=Min(y2,Now.y2);
        if y1<k1 then Add(x1,y1,z1,x2,k1,z2);
        if k2<y2 then Add(x1,k2,z1,x2,y2,z2);
        Cut(x1,k1,z1,x2,k2,z2,Direction+1); {调用 z 方向的切割}
      End;
    3:Begin {接着在 z 方向切}
        k1:=Max(z1,Now.z1);
        k2:=Min(z2,Now.z2);
        if z1<k1 then Add(x1,y1,z1,x2,y2,k1);
        if k2<z2 then Add(x1,y1,k2,x2,y2,z2);
      End;
  End;
End;
Function Cross(x1,x2,x3,x4:longint):boolean; {判断线段[x1,x2], [x3,x4]
                                                                   是否相交}
Begin
  if (x1>=x4)or(x3>=x2) then Cross:=false
  else Cross:=true;
End;
Procedure Caculate;  {计算所有立方体的体积和}
Var i,Volume,k:longint;
Begin
  Volume:=0;
  for i:=1 to tot do
  begin
    k:=(Cubic[i].x2-Cubic[i].x1)*
        (Cubic[i].y2-Cubic[i].y1)*
        (Cubic[i].z2-Cubic[i].z1);
    Inc(Volume,k);
  end;
  Writeln(Volume);
End;

Procedure Main;  {主过程}
Var  i,j,tot1:Longint;
Begin
  for i:=1 to n do
```

```
  begin
    readln(x,y,z,r);
    Now.x1:=x-r;   Now.y1:=y-r;   Now.z1:=z-r;
    Now.x2:=x+r;   Now.y2:=y+r;   Now.z2:=z+r;
    j:=0;
    tot1:=tot; {保存当前队列的尾指针，则 tot1 之后的立方体都是新切割出来的}
    While j<tot1 do
    Begin
      inc(j);
      if (Cross(Cubic[j].x1,Cubic[j].x2,Now.x1,Now.x2))
      and(Cross(Cubic[j].y1,Cubic[j].y2,Now.y1,Now.y2))
      and(Cross(Cubic[j].z1,Cubic[j].z2,Now.z1,Now.z2)) then  {若两立方
体发生重叠，则进行切割}
      Begin
        Cut(Cubic[j].x1,Cubic[j].y1,Cubic[j].z1,
            Cubic[j].x2,Cubic[j].y2,Cubic[j].z2,1);
        Cubic[j]:=Cubic[tot1];
        Cubic[tot1]:=Cubic[tot];
        dec(tot);
        dec(tot1);
        dec(j);
      End;
    End;
        Add(Now.x1,Now.y1,Now.z1,Now.x2,Now.y2,Now.z2);{加入矩形 Now}
  end;
  Caculate;
  Close(output);
End;

Begin
  Init;
  Main;
End.
```

注意这段代码。删除第 j 个立方体，先将第 tot1 个立方体移到位置 j 上，再把当前队列末指针 tot 上的立方体移到位置 tot1 上。并且队列长度减 1，tot←tot-1；且 tot1←tot1-1，j←j-1

附录 4：例 4 的程序　{War.pas}

```
Program War;
Const Maxn=20000;  {矩形集合最多存放 20000 个矩形}
Type Rectangle=Record
                 x1,y1,x2,y2:Longint;
               End;
Var i,j,k,m,n,tot,x1,x2,v,Kill_Num:longint;
    Rect:array[1..Maxn] of Rectangle;
    Now:Rectangle;
```

```
Procedure Init;  {读入数据并预处理的过程}
Begin
  assign(input,'war.in');
  reset(input);
  assign(output,'war.out');
  rewrite(output);
  readln(n,m);
  Fillchar(Rect,sizeof(Rect),0);
  tot:=0;
End;
Function Max(a,b:Longint):Longint; {比较两个数并返回较大者的函数}
Begin
  if a>b then Max:=a
  else Max:=b;
End;
Function Min(a,b:Longint):Longint; {比较两个数并返回较小者的函数}
Begin
  if a<b then Min:=a
  else Min:=b;
End;
Procedure Add(x1,y1,x2,y2:Longint); {加入矩形的过程}
Begin
  inc(tot);
  Rect[tot].x1:=x1;
  Rect[tot].y1:=y1;
  Rect[tot].x2:=x2;
  Rect[tot].y2:=y2;
End;
Procedure Cut(x1,y1,x2,y2,Direction:Longint); {矩形切割的过程}
Var k1,k2:longint;
Begin
  Case Direction of
    1:begin  {先在 x 方向上切}
        k1:=Max(x1,Now.x1); {计算线段[x1,x2],[Now.x1,Now.x2]}
        k2:=Min(x2,Now.x2); {的交集[k1,k2]}
        if x1<k1 then Add(x1,y1,k1,y2);
        if k2<x2 then Add(k2,y1,x2,y2);
        Cut(k1,y1,k2,y2,Direction+1); {调用 y 方向上的切割}
      end;
    2:begin  {再在 y 方向上切}
        k1:=Max(y1,Now.y1); {计算线段[y1,y2],[Now.y1,Now.y2]}
        k2:=Min(y2,Now.y2); {的交集[k1,k2]}
        if y1<k1 then Add(x1,y1,x2,k1);
        if k2<y2 then Add(x1,k2,x2,y2);
```

```
      inc(Kill_Num,(x2-x1)*(k2-k1)); {统计杀掉的外星人数目}
    end;
  End;
End;
Function Cross(x1,x2,x3,x4:Longint):boolean;{判断线段[x1,x2]与[x3,x4]是
                                                              否相交的函数}
Begin
  if (x1>=x4)or(x3>=x2) then Cross:=false
  else Cross:=true;
End;
Procedure Add_Army; {加入外星人的过程}
Begin
  Add(Now.x1,Now.y1,Now.x2,Now.y2); {加入相应的矩形}
End;
Procedure Kill_Army;  {使用武器杀死外星人的过程}
Var i,tot1:Longint;
Begin
  Kill_Num:=0;  {统计被杀死的外星人数目的变量}
  tot1:=tot; {保存当前队列的尾指针，则 tot1 之后的矩形都是新切割出来的}
  i:=0;
  While i<tot1 do
  Begin
    inc(i);
    if (Cross(Rect[i].x1,Rect[i].x2,Now.x1,Now.x2))and
    (Cross(Rect[i].y1,Rect[i].y2,Now.y1,Now.y2)) then
    begin  {发生重叠，进行切割}
      Cut(Rect[i].x1,Rect[i].y1,Rect[i].x2,Rect[i].y2,1);
      Rect[i]:=Rect[tot1];
      Rect[tot1]:=Rect[tot];
      dec(tot);
      dec(tot1);
      dec(i);
    end;
  End;
  writeln(Kill_Num);  {输出杀死的外星人数目}
End;
Procedure Main;  {主过程}
Begin
  for i:=1 to m do
  begin
    readln(k,x1,x2,v);
    Now.x1:=x1-1;
    Now.y1:=0;
    Now.x2:=x2;
```

注意这段代码。删除第 j 个矩形，先将第 tot1 个矩形移到位置 j 上，再把当前队列末指针 tot 上的矩形移到位置 tot1 上。并且队列长度减 1，tot←tot-1；且 tot1←tot1-1，j←j-1

```
    Now.y2:=v;
    if k=1 then Add_Army
    else Kill_Army;
  end;
  Close(output);
End;
Begin
  Init;
  Main;
End.
```

附录 5：例 4 War Field Statistical System 原题

（注：文中对其数据规模做了些许改动）

问题描述

2050 年，人类与外星人之间的战争已趋于白热化。就在这时，人类发明出一种超级武器，这种武器能够同时对相邻的多个目标进行攻击。凡是防御力小于或等于这种武器攻击力的外星人遭到它的攻击，就会被消灭。然而，拥有超级武器是远远不够的，人们还需要一个战地统计系统时刻反馈外星人部队的信息。这个艰巨的任务落在你的身上。请你尽快设计出这样一套系统。

这套系统需要具备能够处理如下 2 类信息的能力：

1.外星人向[x1，x2]内的每个位置增援一支防御力为 v 的部队。
2.人类使用超级武器对[x1，x2]内的所有位置进行一次攻击力为 v 的打击。系统需要返回在这次攻击中被消灭的外星人个数。

注：防御力为 i 的外星人部队由 i 个外星人组成，其中第 j 个外星人的防御力为 j。

输入格式

从文件 c.in 第一行读入 n，m。其中 n 表示有 n 个位置，m 表示有 m 条信息。

以下有 m 行，每行有 4 个整数 k，x1，x2，v 用来描述一条信息 。k 表示这条信息属于第 k 类。x1，x2，v 为相应信息的参数。k=1 or 2。

注：你可以认为最初的所有位置都没有外星人存在。

规模：0<n<=1000；0<x1<=x2<=n；0<v<=1000；0<m<=2000

输出格式

结果输出到文件 c.out。按顺序输出需要返回的信息。

输入样例	对应输出	输出样例
3 5	无	6
1 1 3 4	无	9
2 1 2 3	6	
1 1 2 2	无	
1 2 3 1	无	
2 2 3 5	9	