二分法与统计问题

淮阴中学 李睿

[关键字]

线段树 二叉树 二分法

[摘要]

我们经常遇到统计的问题。这些问题的特点是，问题表现得比较简单，一般是对一定范围内的数据进行处理，用基本的方法就可以实现，但是实际处理的规模却比较大，粗劣的算法只能导致低效。为了解决这种困难，在统计中需要借助一些特殊的工具，如比较有效的数据结构来帮助解决。本文主要介绍的是分治的思想结合一定的数据结构，使得统计的过程存在一定的模式，以到达提高效率的目的。首先简要介绍线段树的基础，它是一种很适合计算几何的数据结构，同时也可以扩充到其他方面。然后将介绍 IOI2001 中涉及的一种特殊的统计方法。接着将会介绍一种与线段树有所不同的构造模式，它的形式是二叉排序树，将会发现这种方法是十分灵活的，进一步，我们将略去对它的构造，在有序表中进行虚实现。

目录

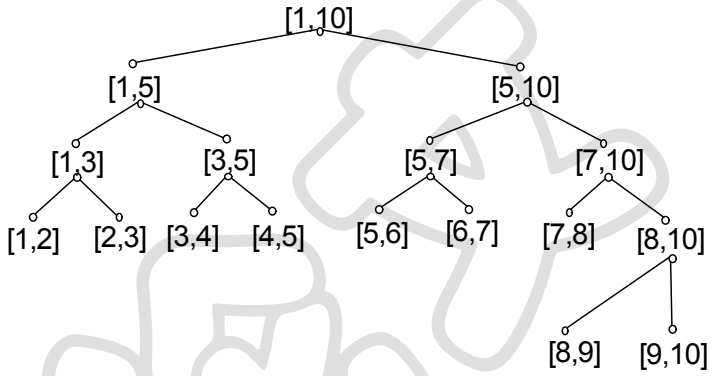
[正文]

一 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。

1.1 线段树的构造思想

线段树处理的是一定的固定线段，或者说这些线段是可以对应于有限个固定端点的。处理问题的时候，首先抽象出区间的端点，例如说 N 个端点 ti(1≤i≤N)。那么对于任何一个要处理的线段（区间）[a,b]来说，总可以找到相应的 i,j，使得 ti=a,tj=b,1≤i≤j≤N。这样的转换就使得线段树上的区间表示为整数，通过映射转换，可以使得原问题实数区间得到同样的处理。下图显示了一个能够表示[1，10]的线段树：



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]。每一个叶子节点上 a+1=b,这表示了一个初等区间。对于每一个内部结点 b-a>1，设根为[a,b]的线段树为 T(a,b),则进一步将此线段树分为左子树 T(a,(a+b)/2),以及右子树 T((a+b)/2,b),直到分裂为一个初等区间为止。

线段树的平分构造，实际上是用了二分的方法。线段树是平衡树，它的深度为 lg(b-a)。

如果采用动态的数据结构来实现线段树，结点的构造可以用如下数据结构：

```
Type
    Tnode=^Treenode;
    Treenode=record
        B,E:integer;
        Count:integer;
        LeftChild,Rightchild:Tnode;
    End;
```

其中 B 和 E 表示了该区间为[B,E]；Count 为一个计数器，通常记录覆盖到此区间的线段的个数。LeftChild 和 RightChild 分别是左右子树的根。

或者为了方便，我们也采用静态的数据结构。用数组 B[]，E[]，C[]，LSON[]，

RSON[]。设一棵线段树的根为 v。那么 B[v],E[v]就是它所表示区间的界。C[v]仍然用来作计数器。LSON[v]，RSON[v]分别表示了它的左儿子和右儿子的根编号。

注意，这只是线段树的基本结构。通常利用线段树的时候需要在每个结点上增加一些特殊的数据域，并且它们是随线段的插入删除进行动态维护的。这因题而异，同时又往往是解题的灵魂。

1.2 线段树处理数据的基本方法

线段树的最基本的建立，插入和删除的过程，以静态数据结构为例。

建立线段树（a,b）:

设一个全局变量 n，来记录一共用到了多少结点。开始 n=0

```
procedure  BUILD(a,b)
begin
    n ← n+1
    v ← n
    B[v]←a
    E[v]←b
    C[v]←0
    if  b – a>1 then
        begin
            LSON[v]←n+1
            BUILD(a,⌊(a+b)/2⌋)
            RSON[v]←n+1
            BUILD(⌊(a+b)/2⌋ ,b)
        end
end
```

将区间[c,d]插入线段树 T(a,b),并设 T(a,b)的根编号为 v：

```
procedure INSERT(c,d;v)
begin
    if c≤B[v] and E[v]≤d then C[v] ← C[v]+1
    else if  c<⌊(B[v]+E[v])/2⌋ then INSERT(c,d;LSON[v]);
        if  d>⌊(B[v]+E[v])/2⌋ then INSERT(c,d;RSON[v]);
end
```

对于此算法的解释：如果[c，d]完全覆盖了当前线段，那么显然该结点上的

基数（即覆盖线段数）加 1。否则，如果[c，d]不跨越区间中点，就只对左树或者右树上进行插入。否则，在左树和右树上都要进行插入。注意观察插入的路径，一条待插入区间在某一个结点上进行“跨越”，此后两条子树上都要向下插入，但是这种跨越不可能多次发生。插入区间的时间复杂度是 O(logn)。

在线段上树删除一个区间与插入的方法几乎是完全类似的：

将区间[c,d]删除于线段树 T(a,b),并设 T(a,b)的根编号为 v：

```
procedure DELETE(c,d;v)
begin
    if c≤B[v] and E[v]≤d then C[v] ← C[v]-1
    else if  c<⌊(B[v]+E[v])/2⌋ then DELETE(c,d;LSON[v]);
        if  d>⌊(B[v]+E[v])/2⌋ then DELETE(c,d;RSON[v]);
end
```

特别注意：只有曾经插入过的区间才能够进行删除。这样才能保证线段树的维护是正确的。例如，在先前所示的线段树上不能插入区间[1，10]，然后删除区间[2，3]，这显然是不能得到正确结果的。

线段树的作用主要体现在可以动态维护一些特征，例如说要得到线段树上线段并集的长度，增加一个数据域 M[v]，讨论：
如果 C[v]>0,M[v] = E[v]-B[v];
C[v]=0 且 v 是叶子结点，M[v]=0；
C[v]=0 且 v 是内部结点，M[v]=M[LSON[v]]+M[RSON[v]];

只要每次插入或删除线段区间时，在访问到的结点上更新 M 的值，不妨称之为 UPDATA，就可以在插入和删除的同时维持好 M。求整个线段树的并集长度时，只要访问 M[ROOT]的值。这在许多动态维护的题目中是非常有用的，它使得每次操作的维护费用只有 logn。

类似的，还有求并区间的个数等等。这里不再深入列举。

1.3 应用的优势

线段树的构造主要是对区间线段的处理，它往往被应用于几何计算问题中。比如说处理一组矩形问题时，可以用来求矩形并图后的轮廓周长和面积等等，比普通的离散化效率更高。这些应用可以在相关资料中查到。这里不作深入。

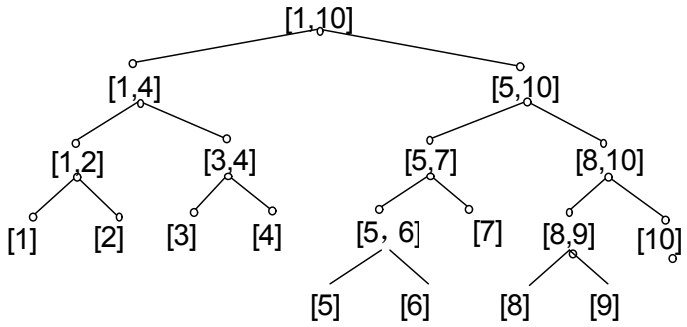
1.4 转化为对点的操作

线段树处理的是区间线段的问题，有些统计问题处理的往往是点的问题。而点也是可以理解为特殊的区间的。这时往往将线段树的构造进行变形，也就是说

可以转化为记录点的结构。

变形：

将线段树上的初等区间分裂为具体的点，用来计数。即不存在(a,a+1)这样的区间，每个点分裂为 a 和 a+1。同时按照区间的中点分界时，点要么落在左子树上，要么落在右子树上。如下图：



原线段数记录基数的 C[v]这时就可以用来计算落在定区间内的点个数了。原搜索路径也发生了改变，不存在“跨越”的情况。同时插入每个点的时候都必须深入到叶结点，因此一般来说都要有 logn 的复杂度。

应用这样的线段树一方面是方便计数。另一方面由于它实际上是排序二叉树，所以容易找出最大和最小来。下面就看一个找最大最小的例子。

[例一]PROMOTION 问题（POI0015）

问题大意：

一位顾客要进行 n（1≤n≤5000）天的购物，每天他会有一些帐单。每天购物以后，他从以前的所有帐单中挑出两张帐单，分别是面额最大的和面额最小的一张，并把这两张帐单从记录中去掉。剩下的帐单留在以后继续统计。输入的数据保证，所有 n 天的帐单总数不超过 1000000，并且每份帐单的面额值是 1 到 1000000 之间的整数。保证每天总可以找到两张帐单。

解决方法：

本题明显地体现了动态维护的特性，即每天都要插入一些面额随机的帐单，同时还要找出最大和最小的两张。不妨建立前面所说的线段树，这棵线段树的范围是[1，1000000]，即我们把所有面额的帐单设为一个点。插入和删除一份帐单是显然的。如何找到最大的帐单呢？显然，对于一个树 v 来说，如果 C[LSON[v]]>0，那么树 v 中的最小值一定在它的左子树上。同样，如果 C[RSON[v]]>0，它的最大值在右子树上；否则，如果 C[LSON[v]]=0，那么最大最小的两份帐单都在右子树上。所以线段树的计数其实为我们提供了线索。显然对于一个特定面额来说。它的插入，删除，查找路径是相同的，长度为树的深度，即 log1000000=20。如果总共有 N 张帐单，那么考虑极限时的复杂度为 N*20+n*20*2。这比普通排序的实现要简单得多。

本题还可以采取巧妙的办法，线段树不一定要存帐单的具体面额。由于我们对 1000000 种面额都进行了保存，所以线段树显得比较庞大。采取一种方法：我们用 hash 来保存每一种面额的帐单数目，然后对于一个具体的帐单，例如面额为 V，我们在线段树中保存 V/100 的值，也就是说，我们把连续的 100 种面额的帐单看成是一组。由于 V 的范围是[1..1000000]，所以线段树中有 10000 个点。

在找最大的数的时候，首先找到最小的组，然后在 hash 里对这个组进行搜索，显然这个搜索的规模不会超过 100。由于线段树变小了，所以树的深度只有 14 左右，整个问题的复杂度极限为 N*14+n*14*100*2，对于问题的规模来说，仍然是高效率的。但这样做比前种方法在一定程度上节省了空间。同时实际上也提醒了我们对线段树应该加以灵活的应用。

二 一种解决动态统计的静态方法

2.1 问题的提出

[例二] IOI2001 MOBILES

> 在一个 N*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≤N≤1024,提问和修改的总数可能达到 60000 条。

正如在摘要中所说的，这类题目的意思非常简单明了，而且用几个小循环就可以解决。但是面对可能将要处理的规模，我们却望而却步了，因为简单的实现效率实在太低了。本题的一种完美解决方法用到了一种特殊的结构定义。问题是二维的，注意到降格的思想，我们对一维的问题进行讨论，然后只要稍微进行推广。

一维的序列求和问题是这样的：设序列的元素存储在 a[]中，a 的下标是 1..n 的正整数，需要动态地更新某个 a[x]的值，同时要求出 a[x1]到 a[y1]这一段所有元素的和。这个问题与 MOBILES 问题实际上提法是一样的。

2.2 数据结构的构造和设想

本题利用前面讲的线段树实际上就可以得到高效地解决。因为我们知道计算 a[x1]到 a[y1]这一段所有元素的和，可以用 sum(1,y1)-sum(1,x1-1)，即用部分和求差的技术。而求 sum(1,x)这种形式在线段树上是容易快速得到的。并且修改元素的值的方法也类似。这里不详细说明。我们希望再构造一种特殊的形式，因为它的实现比线段树要来得简单得多。同时这种思想也是非常有趣和巧妙的。

对于序列 a[]，我们设一个数组 C，其中 $C[i]=a[i-2\text{^}k+1]+\cdots+a[i]$（k 为 i 在二进制下末尾 0 的个数）。于是我们的操作显然与这个特定的 C 有着特殊的联系。那么在这个用来记录的数组中，C[K]到底是怎样的表现呢？举一个例子，C[56]，将 56 写成二进制的形式为 111000，那么 C[56]表示的最小的数是 110001，即 49， C[56]表示的是 a[49]到 a[56]的所有元素的和。可以发现 C 的每个元素表示是无具体规律的，例如对 C[7]，它只能表示 a[7]的值。

2.3 此种数据结构的维护

也许你已经注意到了，对 C 的定义非常奇特，似乎看不出什么规律。下面将具体研究 C 的特性，考察如何在其中修改一个元素的值，以及如何求部分和，之后我们会发现，C 的功用是非常的巧妙的。

如何计算 C[x]对应的 2^k？k 为 x 在二进制数下末尾 0 的个数。

由定义可以看出，这一计算是经常用到的。有无简单的操作可以得到这个结果呢？我们可以利用这样一个计算式子：

2^k=x and (x xor (x-1))

这里巧妙地利用了位操作，只需要进行两步的简单计算。其证明只要联系位操作的具体用法以及 x 的特征就可以得到。

在下面的叙述中我们把这个计算式子用函数 **LOWBIT(x)**来表示。

修改一个 a[x]的值

在前面提出的问题中，我们其实要解决的是两个问题：修改 a[x]的值，以及求部分和。我们已经借用 C 来表示 a 的一些和，所以这两个问题的解决，就是要更新 C 的相关量。对于一个 a[x]的修改，只要修改所有与之有关系，即能够包含 a[x]的 C[i]值，那么具体哪些 C[i]是能够包含 a[x]的呢？举一个数为例，如 x=1001010，从形式上进行观察，可以得到：

p1= 1001010
p2= 1001100
p3= 1010000
p4= 1100000
p5=1000000

这里的每一个 pi 都是能够包含 x 的，也就是说，任意的 C[pi]的值，包含 a[x]的值。这一串数到底有什么规律呢?可以发现：

$$P1 = x$$
$$Pi < Pi + 1$$
$$Pi + 1 = Pi + LOWBIT(Pi)$$

从观察上容易看出这是正确的，从理论上也容易证明它是正确的。这些数是否包括了所有需要修改的值呢？从二进制数的特征上考虑，可以发现对于任意的 Pi<Y<Pi+1，C[Y]所包含的值是 a[Pi　+1]+…+a[Y]。C[Y]是不可能包含 a[x]的。

再注意观察 P 序列的生成，我们每次其实是在最后一个 1 上进位得到下一个数。所以 P 序列所含的数最多为 lgn，这里 n 是 a 表的长度，或者说是 C 表的长度。因为我们记录的值是 C[1]—C[n]，当 P 序列中产生的数大于 n 时，我们已经不需要继续这个过程了。在很多情况下对 a[x]进行修改时，涉及到的 P 序列长度要远小于 logn。对于一般可能遇到的 n 来说，都是几步之内就可以完成的。修改一个元素 a[x]，使其加上 A，变成 a[x]+A，可以有如下的过程：

```
procedure UPDATA(x,A)
begin
    p←x
    while (p<=n) do
     begin
                C[p]←C[p]+A
                p←p+LOWBIT(p)
     end
end
```

计算一个提问[x,y]的结果：

我们下面来解决求部分和的问题。根据以往的经验，把这个问题转化成为求sum(1,y)-sum(1,x-1)。那么如何根据 C 的值来求一个 sum（1,x)呢？容易得到如下过程。

```
function   SUM(x)
begin
    ans ← 0
    p ← x
    while (p>0) do
        begin
            ans←ans+C[p]
            p←p-LOWBIT(p)
        end
    return ans
end
```

这个过程与 UPDATA 十分类似，很容易理解。同时，它的复杂度显然是 lgn。每次解答一个提问时，只要执行 SUM 两次，然后相减。所以一次提问需要的操作次数为 2logn。

通过上两步的分析，我们发现，动态维护数组以及求和过程的复杂度通过 C 的巧妙定义都降到了 logn。这个结果是非常令人惊喜和满意的。

2.4 应用的分析

对于我们提出的一维问题，用前面介绍的两个函数就可以轻易地解决了。注意我们所需要消耗的内存仅是一个很单一的数组，它的构造比起线段树来说要简单得多（很明显，这个问题也可以用前面的线段树结构来解决）。

只要把这个一维的问题很好地推广到二维，就可以解决 IOI2001 的 MOBILES 问题。如何推广呢？注意在 MOBILES 问题中我们要修改的是 a[x,y]的值，那么模仿一维问题的解法，可以将 C[x,y]定义为：

$$C[x,y]=\sum a[i,j] \text{其中} x-LOWBIT(x)+1\le i\le x, y-LOWBIT(y)+1\le i\le y$$

其具体的修改和求和过程实际上是一维过程的嵌套。这里省略其具体描述。可以发现，经过这次推广，算法的复杂度为 $\log^2 n$。而就空间而言，我们仅将一维数组简单地变为二维数组，推广的耗费是比较低的。

可以尝试类似地建立二维线段树来解决这个问题，它的复杂性要比这种静态的方法高得多。

在 IOI2001 的竞赛规则中，将 MOBILES 一题的内存限制在 5Mb。用本节介绍的方法，只需要 4Mb 的 C 数组以及一些零散的变量。而如果用蛮力建立第一节中的线段树，其解决问题的瓶颈是可想而知的。

这种特殊的统计方法对于本题很有优势，同时它推广到高维时比较方便，是

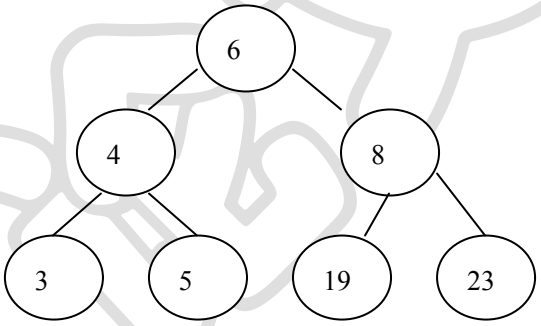
前面所涉及的线段树不可比的。但本节的统计方法也存在缺陷，它似乎不太容易推广到其他问题。仔细研究过线段树会知道它能够支持很多特殊的统计问题。这些将会通过实践体现出来。下面再介绍一些其他的实现方法。

三 在二叉排序树上实现统计

前面已经讲过，线段树经过左右分割以后实际上具有二叉排序树的性质。另一方面，前面也说明过，线段树的建立方式非常适用于处理线段，对于点的问题，可以推广应用，例如例一，但是总有些大材小用的感觉。一方面，在线段树上需要设立过多的指针来指向左子树和右子树；另一方面，结点用于表示区间，处理点的时候，不需要保留这样的区间。线段树上的一个结点分裂为两个半区间的时候实际上是通过一个中间点来分割的，那么在点的统计问题中，只要保留这样的分割点就可以了。

3.1 构造可用于统计的静态二叉排序树

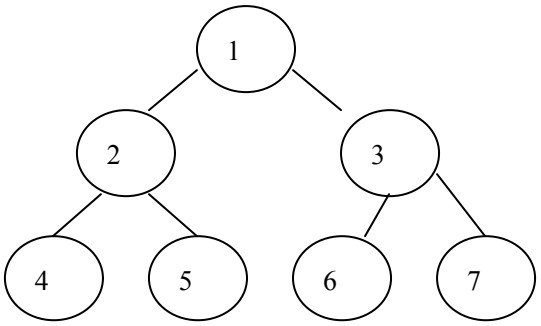
对于处理点的问题，只要建立一棵二叉排序树，使得要处理的点在这棵树上都能够找到相应的节点。同时由二叉树的性质：左子树上的所有点的值都比根小，右子树上的所有的点的值都比根大，我们利用这一点把线段树的优点继承过来。首先要对所有要处理的点建立一棵可用以静态统计的二叉排序树作为模板。例如对于集合{3，4，5，8，19，23，6}，可以建立一棵包含 7 个点的二叉排序统计树：



注意到每个节点上所标的就是它对应的点值。

建立二叉统计树的第一步，是把所有要处理的点坐标离散化，形成一个排序的映射，例如我们称为 X 映射，并且设其中一共有 n 个不同的对象。例如在上例中，X={3，4，5，6，8，19，23}，n=7。

现在要把 X 映射中的点值填入到树中，使它有上面的构造。这里我们选择静态结构作为对二插树的支持。将二叉树的结点从上到下，从左到右进行编号，并令根结点的编号为 1。即上图中对应的编号应该是：



这与静态堆的实现是十分类似的。对于任何一个编号为 i 的结点，它的左儿子编号自然为 i*2，右二子编号为 i*2+1。现在要把 X 的映射填入到数组 V 中去。V[1..n]应该保存相应位置上的点值。

注意到对 V 对应的二叉树进行中序遍历的结果就对应 X 中的映射，所以可以通过递归的方法建立 V：

```
p ← 0   作为 X 映射中的指针
procedure   BUILD(ID:integer)   ID 是 V 结点的下标
begin
      if   (ID*2≤n) then BUILD(ID*2);
      p←p+1
      V[ID]=X[p]
      if   (ID*2+1≤n) then BUILD(ID*2+1);
end
在主程序中调用 BUILD(1)
```

这个过程即对 V 先序遍历。

如果对二叉树先序遍历的过程熟悉，也可以不采用递归过程。只要从先序遍历的第一个结点开始，每次找到它的后继。第一个结点显然是二叉树最下面一层的最左边的结点。如果有 n 个结点，首先通过下面这个过程找到第一个结点：

```
function first:integer
begin
      level  ←1, tot = 2
      while (tot-1<n) do
            begin
                  level←level+1
                  tot←tot*2
            end
      return s div 2
end
```

然后我们可以通过下面这个过程对 V 赋值：

```
procedure BUILD
begin
    now ← first
    p←0
    for i ← 1 to n do
    begin
        p←p+1,V[now]←X[p]
        if (now*2+1≤n) then
          begin
              now←now*2+1
              while (now*2≤n) now←now*2
          end
        else
          begin
            while (now 是奇数) now←now div 2
            now←now div 2
          end
    end
end
```

从构造的方法可以看出，这是一棵近似满二叉树，因此它也是一棵平衡树，它的深度为 logn。

3.2 进行统计的方法分析

在这棵树中，对于任何将要处理的一个点，它具有值 value，我们根据 value 很容易在树中找到相应的结点。例如我们要动态维护点的个数，类似例一中提到的，我们在树的每个结点上设一个 SUM，表示以该结点为根的二叉树上的点的总数。最初 SUM[i]=0。插入一个点有如下过程：

```
procedure INSERT(value)
begin
    now ← 1
    repeat
        SUM[now]←SUM[now]+1
        if (V[now]=value) break
        if (V[now]>value) now←now*2
          else now←now*2+1
    until false
end
```

我们可以在 logn 时间内动态维护 SUM,其过程与 value 的查找是同步的。

这个 SUM 的设立比较普通。有些特殊的设定，就比较有大的作用。比如我们在每个结点上设一个 LESS,表示值小于等于根结点值的总数，即根上的点以及左子树上的点的总数。那么插入时有：

```
procedure INSERT1(value)
begin
    now ← 1
    repeat
        if (value<=V[now]) then
                    LESS[now]←LESS[now]+1
        if (V[now]=value) break
        if (V[now]>value) now←now*2
          else now←now*2+1
    until false
end
```

这个过程与前一个大同小异。实际上 LESS[I]=SUM[I]-SUM[I*2+1]。举这个例子在于说明利用二叉排序树的结构，是很容易结合具体的问题进行变化的。

我们对其变化，甚至也可以利用来解决例二。只要将刚才 LESS 的定义作一点变化，令它为根及其左树上所有点上的权和。如果要在 a[x]上增加 A。可以很容易得到：

```
procedure INSERT2(x,A)
begin
    now ← 1
    repeat
        if (x<=V[now]) then
                    LESS[now]←LESS[now]+A
        if (V[now]=x) break
        if (V[now]>x) now←now*2
          else now←now*2+1
    until false
end
```

同样也很方便，另外如果要求 SUM(1,x)的值，只要根据这样一个函数：

```
function  SUM(x):longint
begin
    ans ← 0
    now ←1
    repeat
        if (V[now]<=x) ans←ans+LESS[now]
        if (V[now]=x) break
        if (V[now]>x) now←now*2
          else now←now*2+1
    until false
    return ans
end
```

可以发现这几个过程基本相似，这种实现对例二解决的效率并不亚于第二节

中介绍的方法，而且它对内存的消耗也是 1 个单一的数组，可以很容易地推广到二维解决 MOBILES 的问题，最后主要的内存消耗也是 4Mb 的静态数组，而它的效率也是较高的。用二叉排序树来实现，其思路和线段树是一样的，因为二者本质上是相似的。

这种方法经常被应用到离散化的统计问题中，尤其是平面问题的统计。

3.3 一个较复杂的例子

[例三]采矿(KOP)

金矿的老师傅年底要退休了。经理为了奖赏他的尽职尽责的工作，决定送他一块长方形地。长度为 S，宽度为 W。老师傅可以自己选择这块地。显然其中包含的采金点越多越好。你的任务就是计算最多能得到多少个采金点。如果一个采金点的位置在长方形的边上，它也应当被计算在内。

任务：

读入采金点的位置。计算最大的价值。

输入：

文件 KOP.IN 的第一行是 S 和 W，(*1<=s,w<=10 000*);他们分别平行于 OX 坐标和 OY 坐标，指明了地域的尺寸。接下来一行是整数 n (*1<=n<=15 000*)，表示采金点的总数。然后是 n 行，每行两个整数，给出了一个采金点的坐标。坐标范围是(*-30 000<=x,y<=30 000*)。

输出：

一个整数，最多的采金点数。

样例输入

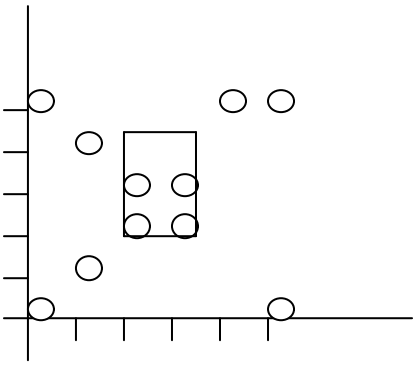
```
1 2
12
0 0
1 1
2 2
3 3
4 5
5 5
4 2
1 4
0 5
5 0
2 3
3 2
```

样例输出

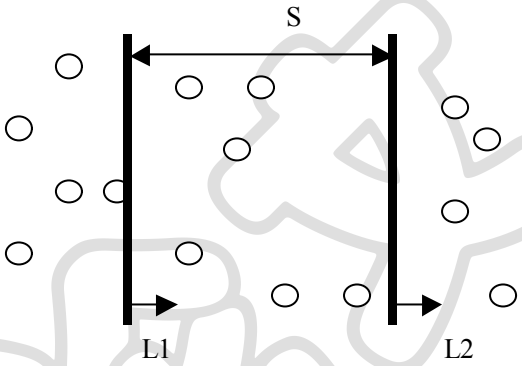
```
4
```

分析：

题目中的样例实际上对应了下图：



这是一个针对点进行扫描的问题。容易想到离散化，例如用两根线来进行扫描，使得两根线之间的区域在 X 坐标上相差不超过 S。然后再统计这一个带状区域中的每一个宽度为 W 的矩形。如下图：

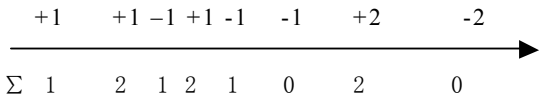


图中是两条扫描线 L1 和 L2。L2 在 L1 前面动，通过调整，使得 L1 到 L2 的距离不大于 S。这时中间带状区域的点成为进一步研究的对象。可以发现，每个点进出要处理的带状区域各一次。

对于带状区域中的所有点，由于他们的横坐标差不会大于 S，所以我们可以忽略所有的横坐标，仅考虑他们的纵坐标。例如在一个带状区域内有 5 个点的纵坐标分别是{5，3，9，1，9}，w=2，很自然地，考虑将这几个坐标排序成{1，3，5，9，9}，然后可以通过类似横坐标的扫描方法来求得宽度为 w 的矩形。但是在本题中，这种方法不是特别好。进一步考虑对纵坐标的特殊处理：

对于每一种坐标 y，建立成两个点事件 (y, +1) , (y+w+1, −1)

例如将前面的点标成 (1,+1),(4,-1),(3,+1),(6,-1),(5,+1),(8,-1),(9,+1),(12,-1), (9,+1),(12,-1), 一共是 10 个点事件，再将他们按照 y 的坐标排序，得 (1,+1),(3,+1),(4,-1), (5,+1)， (6,-1), (8,-1), (9,+1), (9,+1),(12,-1), (12,-1)我们把后面的标号反映在一个 y 坐标的映射上，然后从低到高求和，如下图：



注意坐标下的求和，这些和中最大的一个就是该带状区域中一个包含最多点数的矩形。每个位置上的和记录的是此位置之前所有标号的和。

通过这步巧妙的转换，我们可以用前面的二叉排序树来实现 y 坐标的点事件处理。同时前面已经说过，每一个点进出带状区域仅各一次，因此我们要利用树的统计实现：在插入或者删除一个点事件之后，能够维持坐标下Σ的值；能够在很短时间内得到Σ中最大的一个值。

得到大致算法如下：

将所有的点事件映射到 y 坐标中，最多有 n=15000 个点，所以可能有 30000 个不同的坐标，将这些值建立一棵可用以统计的二叉排序树，即 BUILD。

在树上的每个结点，要设立两个值。一个是 SUM，记录以该点为根的子树上所有点上的值的和，开始时 SUM=0。另一个是 MAXSUM，记录以该点为根的子树上最大的Σ，注意这里的定义是以该结点为根的，也就是在子树上的值。可以通过下面这个函数来完成一个点事件的插入，并且维护 SUM 和 MAXSUM 的特性。其中 k 是一个标号，其值为+1，或者-1。

```
procecure INSERT(y,k)
begin
     now ← 1
     repeat
       if V[now]=y break
       if V[now]<y now←now*2+1
         else now←now*2
     until false;
     repeat
           SUM[now]←SUM[now]+k
           MAXSUM[now]←Max{MAXSUM[now*2],
                           SUM[now]-SUM[now*2+1],
                           SUM[now]-SUM[now*2+1]+MAXSUM[now*2+1]
                           }
           now ← now div 2
     until now = 0
end
```

仔细分析这个过程，它有两个部分，首先向下找到 y 所在的结点，然后再向上，对路径上的每个点的相关量进行修改。SUM 的计算不需要解释。MAXSUM 的计算实际上是一个动态规划的过程。因为是要取得连续和的最大值，所以有 3

种情况：1 最大值在左树上；2 最大值正好包含根结点；3 最大值在右树上。Max 函数括号中的 3 项分别对应了这 3 种情况下的取值。

在完成了 INSERT 操作后，MAXSUM[1]就是当前的一个最优解。INSERT 算法的执行复杂度为树高，即 logn

INSERT 函数是解决本题的核心。有了这个过程以后，很容易完成整个算法，只要先将所有的点按照横坐标先排序，然后利用两条扫描线 L1 和 L2 进行扫描。根据前面的分析，n 个点进出扫描的带状区域各一次，每一次对应两个点事件，因此总共是 4n 次 INSERT 操作。整个算法的时间复杂为 O(nlgn)。算法的编程实现也不复杂。

通过这个例子可以深刻感受到利用二叉树的好处。二叉树与前面的线段树在本质上是相似的，只是这种实现更加方便和简洁了。当然，对于这一类题目，首先要将问题进行一些转化，使之可以有效地利用二叉树。本例中正是利用了点事件这一概念，才找到了有效的算法。

四 虚二叉树

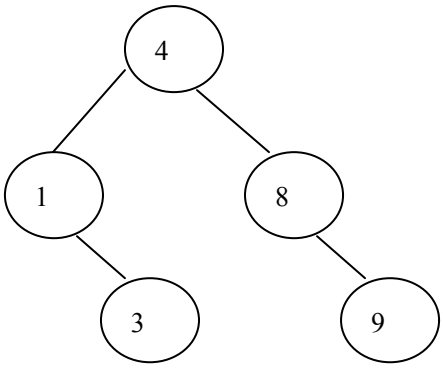
从第三节的介绍中可以看到二叉树是一种非常实用的静态处理方法。但在使用之前必须构造出一个空的二叉树。实际上结合二分法，我们可以不用专门构造出二叉树的结构。在本节中，就介绍怎样略去这种构造。由于失去了实实在在的二叉树结构，这里就将它称之为虚二叉树的实现。

4.1 虚二叉树实现的形态

虽然不去构造这样的二叉树，但是我们必然仍然要用到它的平衡树性质，以及计算上的简便性。这里的想法是源于：对于任何一个有序表，在对其进行二分查找时，实际上就等于在一个二叉树上进行查找。这里我们依然假设查找的对象在表中总是存在的，例如对如下这个二分查找的程序段：

```
function BinSearch(x):integer
begin
    l←1,r←n
    while (l<=r) do
        begin
            m←  (l+r) div 2
            if (V[m]=x) return m
            if (V[m]>x) r←m-1
                else l←m+1
        end
end
```

其中 m 每次都一个区间[l,r]的中间，这与前面所讲的线段树或者二叉树构造都是近似的。对于一个表{1，3，4，8，9}的二分查找，等价于在如下图的二叉排序树上进行查找：



或者用另外一种解释说，m 取中间的值，即为一棵树的树根，它左边的所有元素即[l,m-1]构成了其左子树上的形态，而[m+1,r]构成了右子树上的形态。

虽然这棵树不是近似满二叉树，没有第三节中所构造的那样有非常美的构造，但是它也是一棵平衡二叉树。因为仅由二分查找的性质可得，最多 logn 次查找就可以找到这个结点，即对于某个固定值的修改仅跟 logn 个结点有一定的关系。

在第三节中，我们先构造出一棵树，然后在每个结点上进行修改。这种思路的好处是我们可以通过下标得到父亲儿子的关系，结构明了。

在本节中，通过对二分查找的考察，实际上得出，一个排好了序的线形表，就对应一棵树，这棵树的形态并没有真正构造出来，但在二分查找的过程中它是固定的。

在第三节中，我们把所有需要修改的量都保留在根节点上。把这种方法迁移到本节中，每棵子树的根节点实际上就是我们在二分查找中 m 对应的值。这样我们就省去了构造。

举插入节点的例子，来说明这种虚实现的方法，设 LESS 表示根及其左树上所有结点的个数：

```
procedure INSERT(x)
begin
    l←1,r←n
    while (l<=r) do
        begin
            m=(l+r) div 2
            if   x<=V[m] LESS[m]←LESS[m]+1
            if   x =V[m] break
            if   x<V[m] then r  ←m –1
                 else l←m+1
        end
end
```

只要在头脑中想象一棵虚拟的排序二叉树，就不难理解这个过程的具体含义了。

虚实现实际上省去了构造，因而它在实现上进一步简单化了。在效率上我们依然保证了它的复杂度不超过 logn。

虚实现中的 V 实际上就是一个有序的映射表，它仍然处理动态维护特性的问题，在实现之前，首先仍将所有要处理的对象进行排序。这一点与前面处理问题的方法是类似的。

4.2 一个具体的例子

[例 4]在一个平面直角坐标系上有 n 个点，n 最多为 15000，要求每个点的左下方有多少个点，也就是说对于每个点 xi,yi，求满足 xj≤xi 并且 yj≤yi 的 j 的个数。

分析：

我们已经对于这种类似形式的问题讨论了多次。现在我们要用虚实现来解决这个算法，使得它的时间复杂度为 O(nlogn)。

首先将点排序，排序的规则是 x 优先，在 x 优先的情况下，按照 y 优先。这样可以使得对于任意的(xi≤xj ,yi≤yj),有 i≤j。然后我们可以不考虑 x 坐标了，因为在排好序的表中，x 总是有序，我们只考察 y 的包含情况。

把 y 坐标排序，建立映射关系，放在 V 数组中，V 中的 y 坐标从小到大排序。我们只要按照先前点排序的顺序依次把各个 y 插入到计数用的 LESS 中去，同时就可以得到一个点左下方的值。把前面的函数改成如下形式：

```
function INSERT-AND-GET(y):integer
begin
    l←1,r←n
    ans←0
    while (l<=r) do
        begin
            m=(l+r) div 2
            if   y<=V[m] LESS[m]←LESS[m]+1
            if   y>=V[m] ans←ans+LESS[m]
            if   y =V[m] break
            if   y<V[m] then r  ←m –1
                 else l←m+1
        end
    return ans
end
```

这个函数的实现不需要过多解释，它的复杂度为 O(logn)。

4.3 最长单调序列的动态规划优化问题

最长单调序列是动态规划解决的经典问题。现在以求最长下降序列（严格下降）为例，来说明怎样用 O(nlogn)来解决它。设问题处理的对象是序列 a[1..n]。整个动态规划算法是这样实现的：

```
procedure longest-decreasing-subsequence
begin
    ans ←0
    for  i←1 to n do
      begin
          j←1,k←ans
          while (j≤k) do
              begin
                  m ← (j+k) div 2
                  if b[m]>a[i] j←m+1
                     else k←m-1
              end
          if j>ans   ans←ans+1
          b[j]←a[i]
      end
    return ans
end
```

这一程序非常短小精悍，其中的奥妙还是不少的。为了理解这个过程，还是从最基本的解决方法开始分析。

首先我们都知道求最长下降序列的算法：

$$a[0]=\infty$$

$$M[0]=0$$

$$M[i]=MAX\{M[j]+1,0\le j<i\text{并且}a[j]>a[i]\}$$

$$P[i]=j(j\text{是上式中取}MAX\text{时的}j\text{值})$$

$$ans=MAX\{M[i]\}$$

在这个公式中 P 表示了决策。专门考虑这个 P，可能有多个决策都可以得到 M[i]得到最大值，这些决策都是等价的。那么我们当然可以对 P 进行特殊的限制，即，在所有等价的决策 j 中，P 选择 a[j]最大的那一个。

P 的选择跟我们得到结果并没有任何关系，但是希望对 P 的解释说明这样的问题：对于第 x 个数来说，它可以组成长度为 M[x]的最长下降序列，它的子问题是在 a[1..x-1]中的一个长度为 M[x]-1 的最长下降序列，并且这个序列的最后一个数大于 a[x]。我们让 P 选择这些所有可能解中末尾数最大的，也就是说在处理完 a[1..x-1]之后，对于所有长度为 M[x]-1 的下降序列，P[x]的决策只跟其中末尾最大的一个有关，其余的同样长度的序列我们不需要关心它了。

由此想到，用另外一个动态变化的数组 b，当我们计算完了 a[x]之后，a[1..x]中得到的所有下降序列按照长度分为 L 个等价类，每一个等价类中只需要一个作为代表，这个代表在这个等价类中末尾的数最大，我们把它记为 b[j],1≤j≤L。b[j]是所有长度为 j 的下降序列中，末尾数最大的那个序列的代表。

由于我们把 a[1..x-1]的结果都记录在了 b 中，那么处理当前的一个数 a[x]，我们无需和前面的 a[j]（1≤j≤x-1）作比较，只需要和 b[j]（1≤j≤L)进行比较。

对于 a[x]的处理，我们简单地说明。

首先，如果 a[x]<b[L]，也就是说在 a[1..x-1]中只存在长度为 1 到 L 的下降序

列，其中 b[L]是作为长度为 L 的序列的代表。由于 a[x]<b[L]，显然把 a[x]接在这个序列的后面，形成了一个长度为 L+1 的序列。a[x]显然也可以接在任意的 b[j](1≤j<L)后面，形成长度为 j+1 的序列，但必然有 a[x]<b[j+1]，所以它不可能作为 b[j+1]的代表。这时 b[L+1]=a[x],即 a[x]作为长度为 L+1 的序列的代表，同时 L 应该增加 1。

另一种可能是 a[x]>=b[1]，显然这时 a[x]是 a[1..x]中所有元素中最大的，它仅能构成长度为 1 的下降序列，同时它又必然是最大的，所以它作为 b[1]的代表,b[1]=a[x]。

如果前面的情况都不存在,我们肯定可以找到一个 j,2≤j≤L,有 b[j-1]>a[x],b[j]≤a[x]，这时分析，a[x]必然接在 b[j-1]后面，新成一个新的长度为 j 的序列。这是因为，如果 a[x]接在任何 b[k]后面，1≤k<j-1，那么都有 b[k+1]>a[x]，a[x]不能作为代表。而对于任何的 b[k]，其中 j≤k≤L,b[k]<=a[x]，a[x]不能延长这个序列。由于 a[x]≥b[j],所以我们就将 b[j]更新为 a[x]。

在任何一种情况完成之后，b[1..L]显然是个下降的序列，但它并不表示长度为 L 的下降序列，这点不可混淆。

这几种情况实际上都可以归结为：处理 a[x]，令 b[L+1]为无穷小，从左到右找到第一个位置 j，使 b[j]≤a[x],然后则只要将 b[j]=a[x],如果 j=L+1,则 L 同时增加。x 处以前对应的最长下降序列长度为 M[x]=j。

这样的程序段先描述为：

```
procedure longest-decreasing-subsequence'
begin
    L ←0
    for   x←1 to n do
        begin
            b[L+1]←无穷小
            j←1
            while (b[j]>a[x]) j←j+1
            b[j]←a[x]
            if j>L then L←j
        end
    return L
end
```

注意程序段中的斜体部分，它容易退化，当 a 本身是下降序列时，它退化为 $O(n^2)$的算法。

这时我们就要利用本节提到的二分法了。斜体部分是找到最小的 j，满足 b[j]≤a[x],由于 b[1..L]一定是一个单调下降的有序序列，我们只需要用二分查找找到这个位置。其原理就等同于在二叉树上进行查找。于是就有了我们一开始给出的经典程序段。它的关键部分是：

```
j←1,k←ans
while (j≤k) do
    begin
        m ← (j+k) div 2
        if b[m]>a[i] j←m+1
          else k←m-1
    end
if j>ans   ans←ans+1
b[j]←a[i]
```

以上解决了最长下降序列的长度，其实解决上升序列，或者最长不上升序列，只要将算法中的不等号略做修改。相信在领略了此方法的原理后，你不难做出这种修改。

本题还有有趣的地方，在用 b 求得最长下降序列长度的同时，我们也完成了对 a 序列用最少的不下降序列进行覆盖的构造。换句话说，我们可以通过这个方法来说明一个序列的不下降最小覆盖数等于最长下降序列的长度。这是一个有趣的命题。

为什么说我们完成了构造呢？只要注意到我们每次处理 a[x]的时候，都把 b[j]更新为 a[x]，它的构造意义即是说，a[x]接在 b[j]的那个代表的后面，即 b[j]代表所在覆盖路径上的下个点是 a[x]。同时当 L 增加的时候，我们单独开辟了一条新的覆盖路径，a[x]作为这条路径上的头一结点。

也许本题的动态规划说明显得烦琐。如果将动态规划的方程一点一点地变形，也是能够得到最后的方案的。本文中试图直接说明它的正确性，所以没有采用间接的方法。

[小结]

本文围绕的是统计问题的解决。可以发现我们讨论的问题中大多数是平面离散化一类的问题，解决这些办法的基本思想实际上是利用二分的方法，利用特殊的数据结构实现来达到提高效率的目的。这几节中提到的方法基本相似，在各节中具体讨论了不同的数据结构实现方法，因此也可以说本文是以技巧为主的。特别地，可以发现，一三四节中的例题都可以用几种方法来实现，我们当然应当选择比较容易和简便的方法。一般来说，线段树适于几何问题，其具体的实例在相关参考资料中描述得很具体，本文仅对这方面的知识作了基本介绍，以求达到一定的启示。第二节讲到的方法可以说浓缩了高级的技巧。而本文重点所在是第三和第四节，这两种静态的方法更适合于大多数解决点统计的问题。最后补充了一个动态规划的例子，它实际上是虚实现的一个体现，也提醒我们在多数有序问题中，采用二分法可能会起到点睛的作用。

参考书目

[1]IOI99 中国集训队论文 陈宏 《数据结构的选择和效率》
[2]《计算几何—算法分析与设计》清华大学出版社
[3]《算法与数据结构（第二版）》电子工业出版社
[4]USACO training gate <Dynamic Programming>资料
[5]IOI2001 试题分析（第二版）