[Range Minimum Query and Lowest Common Ancestor[翻译]](http://www.cnblogs.com/drizzlecrj/archive/2007/10/23/933472.html)

2007-10-23 09:38 by 农夫三拳, 6076 阅读, 13 评论, [收藏](http://www.cnblogs.com/drizzlecrj/archive/2007/10/23/933472.html), [编辑](http://www.cnblogs.com/drizzlecrj/admin/EditPosts.aspx?postid=933472)

**Range Minimum Query and Lowest Common Ancestor**

    【原文见  <http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor>】

**作者：      By danielp  
                                                                                                                    Topcoder  Member  
                                                                                                                翻译：      农夫三拳@seu**[Introduction](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Introduction)  
 [Notations](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Notations)  
 [Range Minimum Query (RMQ)](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))  
     [Trivial algorithms for RMQ](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Trivial_algorithms_for_RMQ)  
     [A <O(N), O(sqrt(N))> solution](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#A O(N), O(sqrt(N)) solution)  
     [Sparse Table (ST) algorithm](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Sparse_Table_(ST)_algorithm)  
     [Segment Trees](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Segment_Trees)  
 [Lowest Common Ancestor (LCA)](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Lowest Common Ancestor (LCA))  
     [A <O(N), O(sqrt(N))> solution](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#A O(N), O(sqrt(N)) solution)  
     [Another easy solution in <O(N logN, O(logN)>](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Another easy solution in O(N logN, O(logN))  
     [Reduction from LCA to RMQ](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Reduction from LCA to RMQ)  
 [From RMQ to LCA](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#From RMQ to LCA)  
 [An <O(N), O(1)> algorithm for the restricted RMQ](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#A O(N), O(1) algorithm for the restricted RMQ)  
 [Conclusion](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Conclusion)   
  
**Introduction**

      在一棵树中查找一对结点的最近公共祖先(LCA)的问题在20世纪末期已经被仔细的研究过了，并且它现在已经成为算法中图论的基本算法了。这个问题之所以有趣并不是因为处理它的算法很有技巧，而是因为它在字符串处理和生物学计算中的广泛应用，例如，当LCA和后缀树或者其他树形结构在一起使用时。[Harel and Tarjan](http://www.topcoder.com/tc?module=LinkTracking&link=http://siamdl.aip.org/getabs/servlet/GetabsServlet?prog=normal%26id=SMJCAT000013000002000338000001%26idtype=cvips%26gifs=Yes&refer=)是首先深入研究这个问题的人，他们得出：在对输入树LCA进行线性处理后，查询可以在常数时间内得到答案。他们的工作已经得到了广泛的延伸，这篇教程将展示一些有趣的方法，而它们还也可以用在其他的问题上。

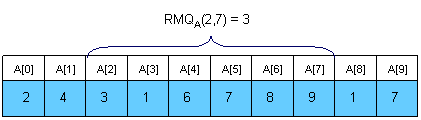
      让我们考虑一个不太抽象的LCA的例子：生命树。地球上当前的居住者是由其他物种进化而来已经是一个不争的事实。这种进化结构可以表示成一棵树，其中节点表示物种，而它的孩子结点表示从该物种直接进化得到的物种。现在通过在树中查找一些结点的LCA把具有相似特征的物种划分成组，我们可以找出两个物种共同的祖先，并且我们可以知道它们所拥有的相似特征是来自于那个祖先。

      Range Minimum Query(RMQ)被用在数组中用来查找两个指定索引中具有最小值的元素的位置。我们后面将会看到LCA问题可以归约成一个带限制的RMQ问题，其中相邻的数组元素相差1。

      尽管如此, RMQ并不是仅仅和LCA一起用的。当他们在和后缀数组（一个新的数据结构，它支持和后缀树同等效率的字符串查询，但是使用更少的内存且编码很简单）一起使用时，在字符串处理中扮演着相当重要的角色。

      在这篇教程中，我们将首先讨论RMQ。我们将给出解决这个问题的多种方法--有一些速度比较慢但是容易编码，而其他的则更快。在第二部分我们将讨论LCA和RMQ之间的关系。首先我们先回顾一下不使用RMQ来解决LCA的两个简单方法；然后我们将指出RMQ和LCA问题其实是等价的；并且，最后，我们将看到RMQ问题怎样规约成它的限制版本，并且对于这个特殊情况给出一个最快的算法。

**Notations**      假设一个算法预处理时间为 **f(n)**，查询时间为**g(n)**。这个算法复杂度的标记为**<f(n), g(n)>**。我们将用**RMQA(i, j)**来表示数组中索引i和j之间最小值的位置。 **u**和**v**的离树T根结点最远的公共祖先用**LCAT(u, v)**表示。  
  
**Range Minimum Query(RMQ)**      给定数组**A[0, N-1]**找出给定的两个索引间的最小值的位置。



**Trivial algorithms for RMQ**

       对每一对索引**(i, j)**，将**RMQA(i, j)**存储在**M[0, N-1][0, N-1]**表中。普通的计算将得到一个**<O(N3), O(1)>** 复杂度的算法。尽管如此，通过使用一个简单的动态规划方法，我们可以将复杂度降低到**<O(N2), O(1)>**。预处理的函数和下面差不多：

**http://www.cnblogs.com/Images/OutliningIndicators/None.gifvoid process1(int M[MAXN][MAXN], int A[MAXN], int N)**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif      int i, j;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (i =0; i < N; i++)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        M[i][i] = i;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (i = 0; i < N; i++)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        for (j = i + 1; j < N; j++)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            if (A[M[i][j - 1]] < A[j])**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                M[i][j] = M[i][j - 1];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            else**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                M[i][j] = j;**

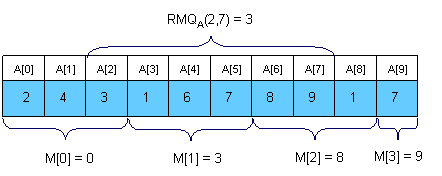
**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}**

这个普通的算法相当的慢并且使用 **O(N2)**的空间，对于大数据它是无法工作的。

**An <O(N), O(sqrt(N))> solution**

  一个比较有趣的点子是把向量分割成**sqrt(N)**大小的段。我们将在**M[0,sqrt(N)-1]**为每一个段保存最小值的位置。

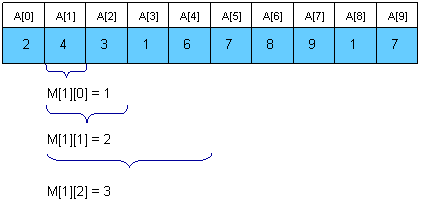
**M**可以很容易的在**O(N)**时间内预处理。下面是一个例子：



       现在让我们看看怎样计算**RMQA(i, j)**。想法是遍历所有在区间中的**sqrt(N)**段的最小值，并且和区间相交的前半和后半部分。为了计算上图中的**RMQA(2,7)**,我们应该比较**A[2]**, **A[M[1]]**, **A[6]** 和**A[7]**，并且获得最小值的位置。可以很容易的看出这个算法每一次查询不会超过**3 \* sqrt(N)**次操作。

      这个方法最大的有点是能够快速的编码（对于TopCoder类型的比赛），并且你可以把它改成问题的动态版本（你可以在查询中间改变元素）。  
  
**Sparse Table (ST) algorithm**

     一个更好的方法预处理**RMQ** 是对**2k**的长度的子数组进行动态规划。我们将使用数组**M[0, N-1][0, logN]**进行保存，其中**M[i][j]**是以**i** 开始，长度为 **2j**的子数组的最小值的索引。下面是一个例子



为了计算**M[i][j]**我们必须找到前半段区间和后半段区间的最小值。很明显小的片段有这**2j - 1** 长度，因此递归如下：

http://www.topcoder.com/i/education/lca/RMQ_007.gif

预处理的函数如下：

http://www.cnblogs.com/Images/OutliningIndicators/None.gifvoid process2(int M[MAXN][LOGMAXN], int A[MAXN], int N)

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif  {

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif      int i, j;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif  //initialize M for the intervals with length 1

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif      for (i = 0; i < N; i++)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          M[i][0] = i;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif  //compute values from smaller to bigger intervals

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif      for (j = 1; 1 << j <= N; j++)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          for (i = 0; i + (1 << j) - 1 < N; i++)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif              if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                  M[i][j] = M[i][j - 1];

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif              else

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                  M[i][j] = M[i + (1 << (j - 1))][j - 1];

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif  }  

http://www.cnblogs.com/Images/OutliningIndicators/None.gif

一旦我们预处理了这些值，让我们看看怎样使用它们去计算**RMQA(i, j)**。思路是选择两个能够完全覆盖区间**[i..j]**的块并且找到它们之间的最小值。设**k = [log(j - i + 1)]**.。为了计算**RMQA(i, j)** 我们可以使用下面的公式

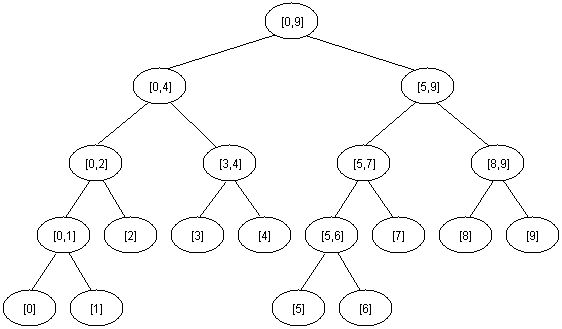
http://www.topcoder.com/i/education/lca/RMQ_005.gif

So, the overall complexity of the algorithm is **<O(N logN), O(1)>**。  
  
**Segment trees**

     为了解决RMQ问题我们也可以使用线段树。线段树是一个类似堆的数据结构，可以在基于区间数组上用对数时间进行更新和查询操作。我们用下面递归方式来定义线段树的**[i, j]**区间：

* 第一个结点将保存区间**[i, j]**区间的信息
* 如果i<j 左右的孩子结点将保存区间**[i, (i+j)/2]**和**[(i+j)/2+1, j]**的信息

      注意具有**N**个区间元素的线段树的高度为**[logN] + 1**。下面是区间[0,9]的线段树：



线段树和堆具有相同的结构，因此我们定义**x**是一个非叶结点，那么左孩子结点为**2\*x**,而右孩子结点为**2\*x+1**。

 使用线段树解决RMQ问题，我们应该使用数组 **M[1, 2 \* 2[logN] + 1]**，这里**M[i]**保存结点i区间最小值的位置。初始时**M**的所有元素为**-1**。树应当用下面的函数进行初始化(**b**和**e**是当前区间的范围):

**http://www.cnblogs.com/Images/OutliningIndicators/None.gifvoid initialize(int node, int b, int e, int M[MAXIND], int A[MAXN], int N)**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifif (b == e)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifM[node] = b;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    else**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif     {**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//compute the values in the left and right subtrees**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        initialize(2 \* node, b, (b + e) / 2, M, A, N);**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        initialize(2 \* node + 1, (b + e) / 2 + 1, e, M, A, N);**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//search for the minimum value in the first and**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//second half of the interval**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        if (A[M[2 \* node]] <= A[M[2 \* node + 1]])**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            M[node] = M[2 \* node];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        else**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            M[node] = M[2 \* node + 1];**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}**

**http://www.cnblogs.com/Images/OutliningIndicators/None.gif**

      上面的函数映射出了这棵树建造的方式。当计算一些区间的最小值位置时，我们应当首先查看子结点的值。调用函数的时候使用 **node = 1**, **b = 0**和**e  = N-1**。

      现在我们可以开始进行查询了。如果我们想要查找区间**[i, j]**中的最小值的位置时，我们可以使用下一个简单的函数：

http://www.cnblogs.com/Images/OutliningIndicators/None.gif  int query(int node, int b, int e, int M[MAXIND], int A[MAXN], int i, int j)

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifint p1, p2;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif 

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif 

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//if the current interval doesn't intersect

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//the query interval return -1

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    if (i > e || j < b)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return -1;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif 

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//if the current interval is included in

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//the query interval return M[node]

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    if (b >= i && e <= j)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return M[node];

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif 

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//compute the minimum position in the

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//left and right part of the interval

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    p1 = query(2 \* node, b, (b + e) / 2, M, A, i, j);

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    p2 = query(2 \* node + 1, (b + e) / 2 + 1, e, M, A, i, j);

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif 

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//return the position where the overall

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//minimum is

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    if (p1 == -1)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return M[node] = p2;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    if (p2 == -1)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return M[node] = p1;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    if (A[p1] <= A[p2])

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return M[node] = p1;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    return M[node] = p2;

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif 

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

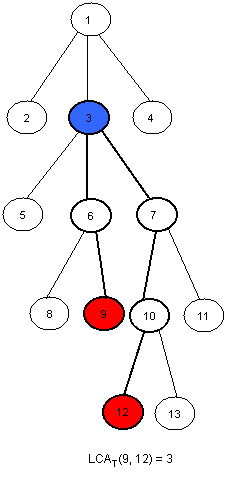
http://www.cnblogs.com/Images/OutliningIndicators/None.gif

     你应该使用**node = 1**, **b = 0**和**e = N - 1**来调用这个函数，因为分配给第一个结点的区间是**[0, N-1]**。

     可以很容易的看出任何查询都可以在**O(log N)**内完成。注意当我们碰到完整的in/out区间时我们停止了，因此数中的路径最多分裂一次。用线段树我们获得了**<O(N), O(logN)>**的算法。线段树非常强大，不仅仅是因为它能够用在RMQ上，

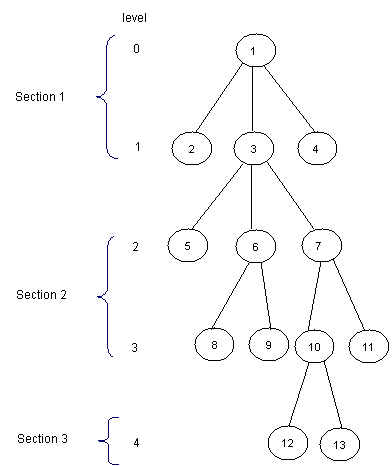
还因为它是一个非常灵活的数据结构，它能够解决动态版本的RMQ问题和大量的区间搜索问题。  
  
**Lowest Common Ancestor (LCA)**

      给定一棵树**T**和两个节点**u**和**v**，找出**u**和**v**的离根节点最远的公共祖先。下面是一个例子（这篇教程中所有的例子中树的根结点均为1）:

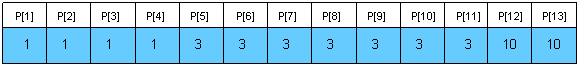


**An <O(N), O(sqrt(N))> solution**

      将输入分成同等大小的部分来解决RMQ问题是一个很有趣的方法。这个方法对LCA问题同样适用。大致思想是将树分成**sqrt(H)**个部分，其中**H**是树的高度。因此第一个段将包含**0**到**sqrt(H)-1**层，第二个段则包含**sqrt(H)**到**2\*sqrt(H)-1**层，依次下去。下面给出了样例中的树是如何被分割的：



      现在，对于每一个结点，我们应该知道每一个段的在上一层中的祖先。我们将预处理这些值，并将他们存储在**P[1, MAXN]**中。下面是对于样例中的树的P数组内容(为了简化，对于在第一个段中的所有结点**i**,**P[i]=1**):



      注意对于每一个段中的上面一部分,**P[i]=T[i]**。我们可以使用深度优先搜索对**P**进行预处理(**T[i]**是树中i结点的父亲结点，**nr**为**[sqrt(H)]**，**L[i]**是结点**i**所处的层的编号):

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gifvoid dfs(int node, int T[MAXN], int N, int P[MAXN], int L[MAXN], int nr)  {**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifint k;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//if node is situated in the first**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//section then P[node] = 1**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//if node is situated at the beginning**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//of some section then P[node] = T[node]**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//if none of those two cases occurs, then**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//P[node] = P[T[node]]**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifif (L[node] < nr)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif         P[node] = 1;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifelse**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        if(!(L[node] % nr))**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif              P[node] = T[node];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        else**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif              P[node] = P[T[node]];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.giffor each son k of node**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif      dfs(k, T, N, P, L, nr);**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}**

**http://www.cnblogs.com/Images/OutliningIndicators/None.gif**

 现在，我们可以很容易的进行查询了。为了找到**LCA(x,y)**，我们首先找出它所在的段，然后再用普通的方法计算它。下面是代码：

http://www.cnblogs.com/Images/OutliningIndicators/None.gif int LCA(int T[MAXN], int P[MAXN], int L[MAXN], int x, int y)

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//as long as the node in the next section of

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//x and y is not one common ancestor

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we get the node situated on the smaller

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//lever closer

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    while (P[x] != P[y])

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          if (L[x] > L[y])

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifx = P[x];

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          else

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                y = P[y];

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//now they are in the same section, so we trivially compute the LCA

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifwhile (x != y)

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          if (L[x] > L[y])

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif             x = T[x];

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif          else

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif             y = T[y];

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif      return x;

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

http://www.cnblogs.com/Images/OutliningIndicators/None.gif

      这个函数最多执行**2 \* sqrt(H)**次操作。通过使用这个方法，我们得到了**<O(N), O(sqrt(H))>**的算法，这里**H**指的是树的高度。在最坏的情况下**H=N**，因此总的复杂度为**<O(N), O(sqrt(N))>**。这个算法的主要好处是易于编码(Division1中的程序员应该在15分钟内完成这段代码)。  
  
**Another easy solution in <O(N logN, O(logN)>**

    如果我们对这个需要一个更快的解决方法，我们可以使用动态规划。首先我们构建一张表**P[1,N][1,logN]**，这里**P[i][j]**指的是结点**i**的第**2j**个祖先。为了计算这个值，我们可以使用下面的递归：

http://www.topcoder.com/i/education/lca/LCA_005.gif

预处理的函数如下：

http://www.cnblogs.com/Images/OutliningIndicators/None.gif void process3(int N, int T[MAXN], int P[MAXN][LOGMAXN])  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    int i, j;  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we initialize every element in P with -1  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (i = 0; i < N; i++)  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        for (j = 0; 1 << j < N; j++)  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            P[i][j] = -1;  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//the first ancestor of every node i is T[i]  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (i = 0; i < N; i++)  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        P[i][0] = T[i];  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//bottom up dynamic programing  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.giffor (j = 1; 1 << j < N; j++)  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif       for (i = 0; i < N; i++)  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif           if (P[i][j - 1] != -1)  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifP[i][j] = P[P[i][j - 1]][j - 1];  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif

这个过程将花费**O(N logN)** 的时间和空间。现在让我们看看如何查询。用**L[i]**来表示节点**i**在树中所处的层数。可以看到，如果**p**和**q**在树中的同一层中，我们可以使用一个类二分查找的方法进行搜索。因此，对于**2**的**j**次方(界于**log[L[p]**和**0**之间，降序)，如果**P[p][j] != P[q][j]**，那么可以知道**LCA(p, q)**必然在更高的层中，因此我们继续搜索**LCA(p = P[p][j], q = P[q][j])**。最后，**p**和**q**都有了相同的祖先，因此返回**T[p]**。让我们看看如果**L[p] != L[q]**的情况。 不妨假设**L[p] < L[q]**。我们可以使用类似的二分搜索方法来查找与**q**在同一层次的**p**的祖先，然后我们在用下面所描述的方法计算**LCA**。整个函数如下:

**http://www.cnblogs.com/Images/OutliningIndicators/None.gifint query(int N, int P[MAXN][LOGMAXN], int T[MAXN],**

**http://www.cnblogs.com/Images/OutliningIndicators/None.gifint L[MAXN], int p, int q)**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    int tmp, log, i;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//if p is situated on a higher level than q then we swap them**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifif (L[p] < L[q])**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.giftmp = p, p = q, q = tmp;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we compute the value of [log(L[p)]**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (log = 1; 1 << log <= L[p]; log++);**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    log--;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we find the ancestor of node p situated on the same level**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//with q using the values in P**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (i = log; i >= 0; i--)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        if (L[p] - (1 << i) >= L[q])**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            p = P[p][i];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    if (p == q)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return p;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we compute LCA(p, q) using the values in P**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.giffor (i = log; i >= 0; i--)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        if (P[p][i] != -1 && P[p][i] != P[q][i])**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            p = P[p][i], q = P[q][i];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

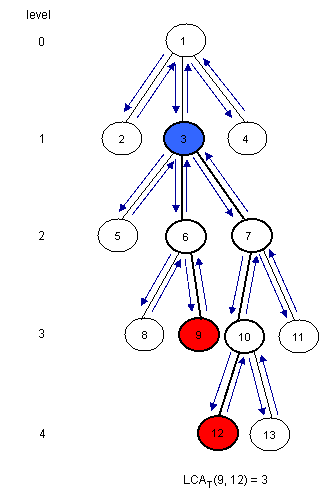
**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    return T[p];**

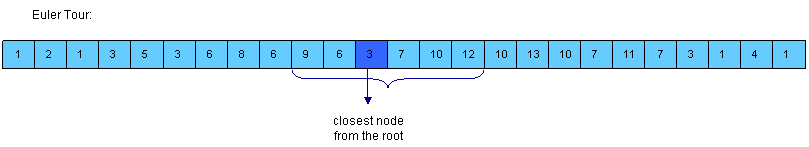
**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}**

**http://www.cnblogs.com/Images/OutliningIndicators/None.gif**

      现在，我们可以看到这个函数最多需要执行**2\*log(H)**次的操作，这里的H是树的高度。在最坏情况下**H=N**,因此总的时间复杂度为**<O(NlogN),O(logN)>**。这个方案非常易编码，并且它比前一个要快。  
  
**Reduction from LCA to RMQ**

     现在，让我们看看怎样用RMQ来计算LCA查询。事实上，我们可以在线性时间里将LCA问题规约到RMQ问题，因此每一个解决RMQ的问题都可以解决LCA问题。让我们通过例子来说明怎么规约的:

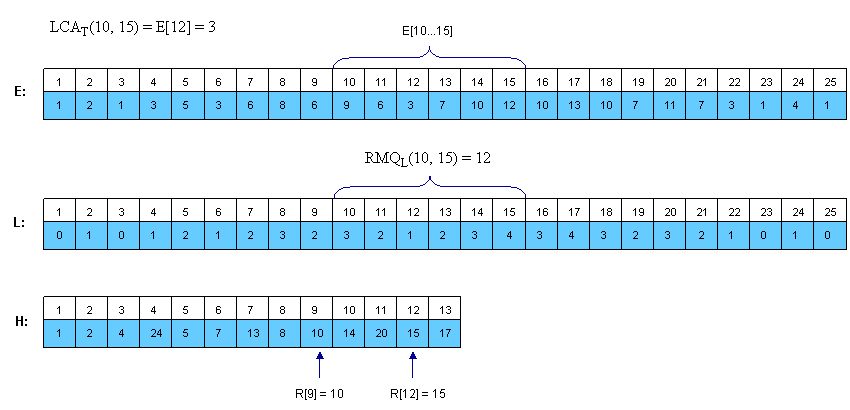


[](http://www.topcoder.com/i/education/lca/LCA_007.gif)

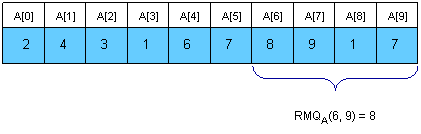
                                                                    点击放大图片  
      注意**LCAT(u, v)**是在对**T**进行dfs过程当中在访问**u**和**v**之间离根结点最近的点。因此我们可以考虑树的欧拉环游过程**u**和**v**之间所有的结点，并找到它们之间处于最低层的结点。为了达到这个目的，我们可以建立三个数组:  
  
**E[1, 2\*N-1]**  - 对**T**进行欧拉环游过程中所有访问到的结点;**E[i]**是在环游过程中第**i**个访问的结点  
**L[1,2\*N-1]** -  欧拉环游中访问到的结点所处的层数;**L[i]**是**E[i]**所在的层数  
**H[1, N] - H[i]**是**E**中结点**i**第一次出现的下标(任何出现i的地方都行，当然选第一个不会错)

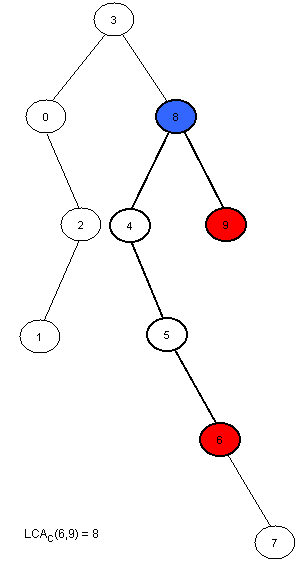
    假定**H[u]<H[v]**(否则你要交换**u**和**v**)。可以很容易的看到**u**和**v**第一次出现的结点是**E[H[u]..H[v]]**。现

在，我们需要找到这些结点中的最低层。为了达到这个目的，我们可以使用**RMQ**。因此 **LCAT(u, v) = E[RMQL(H[u], H[v])]** (记住RMQ返回的是索引)，下面是**E,L,H**数组:

[](http://www.topcoder.com/i/education/lca/LCA_008.gif)  
点击放大图片

注意L中连续的元素相差为1。  
  
**From RMQ to LCA**  
      我们已经看到了LCA问题可以在线性时间规约到RMQ问题。现在让我们来看看怎样把RMQ问题规约到LCA。这个意味着我们实际上可以把一般的RMQ问题规约到带约束的RMQ问题(这里相邻的元素相差1)。为了达到这个目的，我们需要使用笛卡尔树。  
      对于数组**A[0,N-1]**的笛卡尔树**C(A)**是一个二叉树，根节点是**A**的最小元素，假设**i**为**A**数组中最小元素的位置。当**i**>0时，这个笛卡尔树的左子结点是**A[0,i-1]**构成的笛卡尔树，其他情况没有左子结点。右结点类似的用**A[i+1,N-1]**定义。注意对于具有相同元素的数组**A**，笛卡尔树并不唯一。在这篇教程中，将会使用第一次出现的最小值，因此笛卡尔树看作唯一。可以很容易的看到**RMQA(i, j) = LCAC(i, j)**。  
下面是一个例子:





现在我们需要做的仅仅是用线性时间计算**C(A)**。这个可以使用栈来实现。初始栈为空。然后我们在栈中插入**A**的元素。在第**i**步,**A[i]**将会紧挨着栈中比**A[i]**小或者相等的元素插入,并且所有较大的元素将会被移除。在插入结束之前栈中**A[i]**位置前的元素将成为**i**的左儿子，**A[i]**将会成为它之后一个较小元素的右儿子。在每一步中，栈中的第一个元素总是笛卡尔树的根。  
如果使用栈来保存元素的索引而不是值，我们可以很轻松的建立树。下面是上述例子中每一步栈的状态:

|  |  |  |
| --- | --- | --- |
| Step | Stack | Modifications made in the tree |
| 0 | 0 | 0 is the only node in the tree. |
| 1 | 0 1 | 1 is added at the end of the stack. Now, 1 is the right son of 0. |
| 2 | 0 2 | 2 is added next to 0, and 1 is removed (A[2] < A[1]). Now, 2 is the right son of 0 and the left son of 2 is 1. |
| 3 | 3 | A[3] is the smallest element in the vector so far, so all elements in the stack will be removed and 3 will become the root of the tree. The left child of 3 is 0. |
| 4 | 3 4 | 4 is added next to 3, and the right son of 3 is 4. |
| 5 | 3 4 5 | 5 is added next to 4, and the right son of 4 is 5. |
| 6 | 3 4 5 6 | 6 is added next to 5, and the right son of 5 is 6. |
| 7 | 3 4 5 6 7 | 7 is added next to 6, and the right son of 6 is 7. |
| 8 | 3 8 | 8 is added next to 3, and all greater elements are removed. 8 is now the right child of 3 and the left child of 8 is 4. |
| 9 | 3 8 9 | 9 is added next to 8, and the right son of 8 is 9. |

     注意**A**中的每个元素最多被增加一次和最多被移除一次。因此上述算法的时间复杂度为**O(N)**。下面是树的处理函数：

**http://www.cnblogs.com/Images/OutliningIndicators/None.gifvoid computeTree(int A[MAXN], int N, int T[MAXN])**

**http://www.cnblogs.com/Images/OutliningIndicators/None.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    int st[MAXN], i, k, top = -1;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we start with an empty stack**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//at step i we insert A[i] in the stack**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    for (i = 0; i < N; i++)**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//compute the position of the first element that is**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//equal or smaller than A[i]**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        k = top;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        while (k >= 0 && A[st[k]] > A[i])**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            k--;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we modify the tree as explained above**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif       if (k != -1)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            T[i] = st[k];**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif       if (k < top)**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            T[st[k + 1]] = i;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//we insert A[i] in the stack and remove**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//any bigger elements**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        st[++k] = i;**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        top = k;**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//the first element in the stack is the root of**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif//the tree, so it has no father**

**http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    T[st[0]] = -1;**

**http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}**

**http://www.cnblogs.com/Images/OutliningIndicators/None.gif**

**An<O(N), O(1)> algorithm for the restricted RMQ**

      现在我们知道了一般的RMQ问题可以使用LCA归约成约束版本。这里，数组中相邻的元素差值为1.我们可以使用一个更快的**<O(N), O(1)>**的算法。下面我们将在数组**A[0,N-1]**上解决RMQ问题，这里**|A[i]-A[i+1]|=1,i=[1,N-1]**。我们将把**A**转换为一个二元的有着**N-1**个元素的数组，其中**A[i]=A[i]-A[i+1]**。很显然A中的元素只有可能是**+1**或者**-1**。注意原来的**A[i]**的值现在是**A[1],A[2],...,A[i]**的和加上原来的**A[0]**。尽管如此，下面我们根本不需要原来的值。

      为了解决这个问题的约束版本，我们需要将**A**分成**l = [(log N) / 2]**的大小块.让**A'[i]**为**A**中第**i**块的最小值,**B[i]**为**A**中最小块值的位置。A和B的长度均为**N/l**。现在我们利用第一节中讨论的ST算法预处理A'数组。这个将花费**O(N/l \* log(N/l))=O(N)**的时间和空间。经过预处理之后，我们可以在O(1)时间内在很多块上进行查询。具体的查询过程和上面说过的一样。注意每个块的长度为**l=[(logN)/2]**，这个非常的小。同样，要注意A是一个二元数组。二元数组的总的元素的大小**l**满足**2l=sqrt(N)**。因此，对于每一个二元数组中的块**l**，我们需要在表**P**中查询每一对索引的RMQ。这个可以在**O(sqrt(N)\*l2)=O(N**) 的时间和空间内解决。为了索引表**P**，可以预处理**A**中的每一个块并且将其存储在数组**T[1,N/l]**中。块的类型可以成为一个二进制数如果把-1替换成0，把+1替换成1。  
  
    现在，对于询问 **RMQA(i, j)**我们有两种情况:

 **i**和**j**在同一个块中,因此我们使用在**P**和**T**中计算的值

 **i**和**j**在不同的块中，因此我们计算三个值:从**i**到**i**所在块的末尾的**P**和**T**中的最小值，所有**i**和**j**中块中的通过与处理得到的最小值以及从**j**所在块**i**和**j**在同一个块中,因此我们使用在**P**和**T**中计算的值**j**的**P**和**T**的最小值；最后我们我们只要计算三个值中最小值的位置即可。  
**Conclusion**  
      RMQ和LCA是密切相关的问题，因为他们互相之间都可以规约。有许多算法可以用来解决它们，并且他们适应于一类问题。  
  
下面是一些用来练习线段树，LCA和RMQ的题目:  
  
SRM 310 -> [Floating Median](http://www.topcoder.com/stat?c=problem_statement&pm=6551&rd=9990)  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://acm.pku.edu.cn/JudgeOnline/problem?id=1986&refer=>  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://acm.pku.edu.cn/JudgeOnline/problem?id=2374&refer=>  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=2045&refer=>  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://acm.pku.edu.cn/JudgeOnline/problem?id=2763&refer=>  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://www.spoj.pl/problems/QTREE2/&refer=>  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://acm.uva.es/p/v109/10938.html&refer=>  
<http://www.topcoder.com/tc?module=LinkTracking&link=http://acm.sgu.ru/problem.php?contest=0%26problem=155&refer=>  
  
  
**References**  
- "[Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE](http://www.topcoder.com/tc?module=LinkTracking&link=http://www.bio.ifi.lmu.de/~fischer/fischer06theoretical.pdf&refer=)" [PDF] by Johannes Fischer and Volker Heunn  
- "[The LCA Problem Revisited](http://www.topcoder.com/tc?module=LinkTracking&link=http://www.math.tau.ac.il/~haimk/seminar04/LCA-seminar-modified.ppt&refer=)" [PPT] by Michael A.Bender and Martin Farach-Colton - a very good presentation, ideal for quick learning of some LCA and RMQ aproaches  
- "[Faster algorithms for finding lowest common ancestors in directed acyclic graphs](http://www.topcoder.com/tc?module=LinkTracking&link=http://www.cis.njit.edu/~czumaj/PUBLICATIONS/Czumaj-Kowaluk-Lingas-TCS-Summer-2005-r1-Aug-21-2006.pdf&refer=)" [PDF] by Artur Czumaj, Miroslav Kowaluk and Andrzej Lingas