[零零散散学算法之详解RMQ & LCA](http://blog.csdn.net/pi9nc/article/details/8142384)

**RMQ、LCA概述**

LCA:Lowest Common Ancestor，译为最近公共祖先。其解释就是说：在有根树中，找出树中任意两个节点最近的公共祖先，或者说找到任意两个节点离树根最远的公共祖先。

RMQ：Range Minimum Query，译为区间最小值查询。其解释就是说：对于含有N个元素的数列A，在数列中找到两个指定索引之间的最小值及最小值的位置。

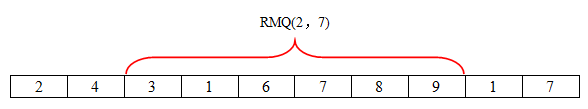
**RMQ Algorithm**

首先我们来看RQM算法，我将会根据预处理和查询的速度介绍几种解决该问题的方法。

设有数组A[N]，其表示如下：

http://img.my.csdn.net/uploads/201210/23/1351001922_7946.png

要求求得区间(2,7)的最小元素，如下图所示：



**解法一：直接遍历区间**

看到这个问题之后，我们最先想到的就是对区间的这些数进行一次遍历，就可以找到区间的最值，因此查询的时间为O(M)。但是，当数据量非常大并且查询很频繁时，直接遍历序列的效果就不是那么理想了。因为每查询一次就得对序列做一次遍历，对于大数据量这显然不能满足要求了。不过对于小数据量，这种算法倒是不错的选择！

查询：O(M)。

算法的代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

**int** MaxNum = 0;

**for**(i = 0; i < range; i++)

{

   /\*\*查找最大值\*\*/

**if**(array[i] > MaxNum)

   {

      MaxNum = array[i];

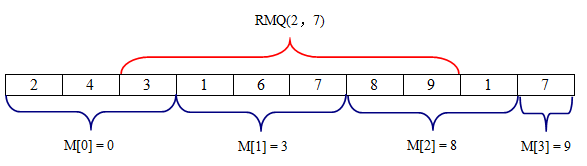
   }

}

**解法二：切割法**

解法一中查询的速度为O(M)，如果每次查询都这样的话，那真就成了龟速了。于是我们对解法一做了预处理，这就是该节要讲的：切割法。

首先，我们将序列分成sqrt(N)个部分，用数组M[sqrt(N) ]来表示每个部分中最小的值的下标，即这个最小数的位置。对于数组M，我们只需对原序列进行一次遍历就可以得到M。如下图所示：



接下来我们来求RMQ[2，7]。为了得到区间[2，7]的最小值，我们需比较A[2]，A[M[1]]，A[6]，以及A[7]，并得到他们中最小值的下标。

分析：其实，这种方法较第一种方法而言并没有实质的改进，甚至还不如方法一。至于为什么这样做，我的解释是*：我们是基于查询快慢的角度上来比较的，说白了，就是我们追求的是查询速度，所以说只要查的快了，先做一些预处理也是值得的（解法四正是基于这种思想）。*现在我们根据上面的例子来看看法二，当做完预处理之后，得到了数组M，此时我们要求区间的最值，那么我们只需将在区间内，包含数组M的值以及包含两个边界的值作比较就行，这样的话，查询的次数：O(M) <= 查询次数 < O(M) + K，其中K < sqrt(N)。

**解法三：排序**

解法二已经提到我们的目的是查得快，那么我们可对选择区间的这M个数据进行排序，然后就可以直接得到最小值。但是如果做排序的话，会有很大的缺陷。我们来看看。

分析：我们选择快速排序，O(M \* LogM)，但是快速排序会改变序列中数的相对位置，因此用快排的话，为了保证原数据的顺序不变，我们还得用O(M)的空间来维护原序列，因此这样的消耗是很大的。附注：复杂度为O(M \* M)的排序算法在这就不啰嗦了！你懂得！

查询：O(1)。

OK，我们来实现我们的想法，代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

快速排序

**int** partition(**int** \*array, **int** low, **int** high)

{

**int** key = array[high];

**int** i = low;

**int** j = high;

**while**(i < j)

    {

**while**(array[i] <= key && i < j)

        {

            i++;

        }

        array[j] = array[i];

**while**(array[j] >= key && i < j)

        {

            j--;

        }

        array[i] = array[j];

    }

    array[i] = key;

**return** i;

}

**void** quicksort(**int** \*array, **int** low, **int** high)

{

**int** index;

**int** i = low;

**int** j = high;

**if**(i < j)

    {

        index = partition(array, low, high);

        quicksort(array, low, index - 1);

        quicksort(array, index + 1, high);

    }

}

排完序之后就可以直接得到最值了！

**解法四：Sparse Table(ST) algorithm**

ST算法是一种比较高效的在线处理RMQ问题的算法，所谓在线算法，是指每输入一个查询就会马上处理这个查询。ST算法首先会对序列做预处理，完成之后就可以对查询做回答了。

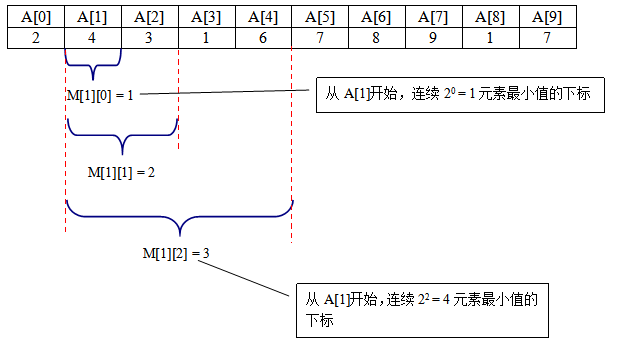
分析：

预处理：O(N \* LogN)。

查询：O(1)，这样的查询正是我们想要的。

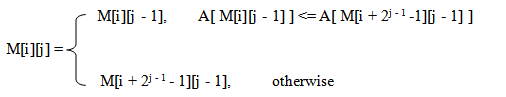
好了，我来详细讲述一下ST算法：

预处理：首先用维护一个数组M[N][LogN]，M[i][j]的值是从原序列A的i位置开始，连续2j 个元素的最小值的下标，如下所示：



那么，我们如何计算M[i][j]呢？

我们采用DP的思想将区间分成两部分，即M[i][j - 1]和M[i][2^(j - 1)]。现在我们只需比较这两个子区间就可以得到M[i][j]了。比较规则如下：



于是乎，就可按照此写出代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

**void** Proprocessing(**int** M[N][logN], **int** \*A, **int** N)

{

**int** i, j;

**for**(j = 1; (1 << j) < N; j++)

    {

**for**(i = 0; (i + (1 << j) - 1) < N; i++)

        {

**if**(A[ M[i][j - 1] ] < A[ M[i + (1 << (j - 1))][i - 1]])

            {

                M[i][j] = M[i][j - 1];

            }

**else**

            {

                M[i][j] = A[ M[i + (1 << (j - 1))][i - 1]];

            }

        }

    }

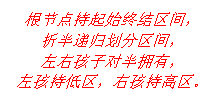
}

**解法五：线段树**

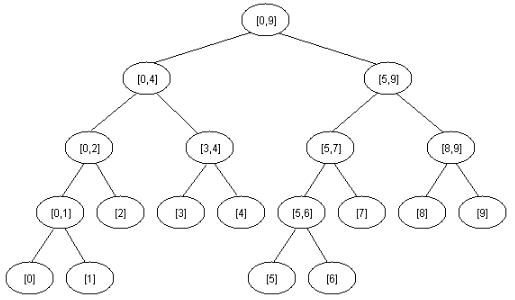
我们也可用线段树来解决RMQ问题，如需了解线段树，请到此一游：

线段树：<http://en.wikipedia.org/wiki/Segment_tree>

线段树的构造口诀：



ok，我们根据口诀，并用上面的例子构造了线段树，如下：



那么将线段树应用到RMQ问题中，首先，维护一个有2^([logN] + 1 + 1) 元素，名为M的数组，即M[2^([logN] + 1 + 1)]，我先来解释一些数组M的意义：M[i]表示已划分节点区间的最小值的位置（下标）。

知道了这些，那我们就通过代码来实现线段树的构造，并通过节点所代表的值来计算得到数组M。代码如下:

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

**void** init\_tree(**int** node, **int** low, **int** high, **int** \*array, **int** \*M)

{

/\*\*\*node:表示线段树中的某个节点

\*\*\*\*low :表示低索引

\*\*\*\*high:表示高索引

\*\*\*\*array:表示原数组

\*\*\*\*M：  表示维护下标的数组

\*\*\*/

**if**(low == high) //为叶子节点

        {

                M[node] = low;

        }

**else**

        {

                init\_tree(2 \* node, low, (low + high)/2, array, M);

                init\_tree(2 \* node + 1, (low + high)/2 + 1, high, array, M);

**if**(array[ M[2 \* node] ] <= array[ M[2 \* node + 1] ]) //拿到较小值的下标

                {

                        M[node] = M[2 \* node];

                }

**else**

                {

                        M[node] = M[2 \* node + 1];

                }

        }

}

通过代码，可得到构造线段树的复杂度为O(N)。

线段树构造成功，接下来就是查询了。我们知道，线段树查询所需的时间为O(LogN)。因为我们在前面已经了解了线段树的几种操作，所以查询在这就不赘述了，直接看代码吧！

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

**int** query(**int** node, **int** low, **int** high, **int** \*a, **int** \*b, **int** i, **int** j)

{

/\*\*\*node:表示线段树中的某个节点

\*\*\*\*low :表示低索引

\*\*\*\*high:表示高索引

\*\*\*\*array:表示原数组

\*\*\*\*M：  表示维护下标的数组

\*\*\*\*i, j:表示要查询的区间

\*\*\*/

**int** s, t;

**if**(i > high || j < low)

**return** -1;

**if**(low >= i && high <= j)

**return** b[node]; //返回最小值的下标

    s = query(2 \* node, low, (low + high)/2, a, b, i, j);

    t = query(2 \* node + 1, (low + high)/2 + 1, high, a, b, i, j);

**if**(s == -1)

**return** b[node] = t;

**if**(t == -1)

**return** b[node] = s;

**if**(a[s] <= a[t])

**return** b[node] = s;

**else**

**return** b[node] = t;

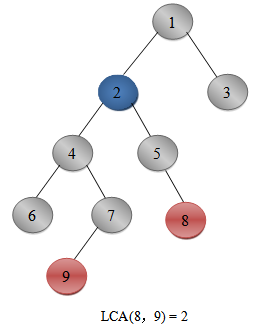
}

**第三节 LCA Algorithm**

LCA算法的概念我们已经知道了，那我们就来看看它的实现过程吧！

对于一棵树，在这我用二叉树，如下图所示。我们要找节点8和节点9的最近公共祖先，即节点2。

附注：有些朋友说这个问题可以当做两条链表是否相交的问题来解决，我们只需分别得到两个节点到根节点的路径，而这两条路径就是两条链表，问题就迎刃而解了。显然这是可行的。



战前准备：

数组T[i]：表示树中某个节点i的父节点；

数组L[i]：表示树中的某个节点i。

维护数组：P[N][LogN]：其中，P[i][j]表示树中i节点的第j个祖先。

实现的过程如下：

利用二分检索判断节点p和节点q是否在树的同一层：

如果在同一层，那么我们通过DP思想，不断地求LCA(p = P[p][j]，q = P[q][j])，一旦 p = q就停止，因为此时p和q的父节点是一样的，也就是说我们找到了最近公共祖先。

如果不在同一层，如果p > q，也就是说p相对与q，p在树的更深层。此时，我们仍然通过DP思想来找到q与p的祖先在同一层的节点，即q = p\_祖先。接下来就可按照在同一层的做法做了。

实现就是这么简单。

首先是预处理得到维护数组P[N][LogN]：

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

**void** preprocessing(**int** \*t, **int** n, **int** p[][max])

{

**int** i, j;

**for**(i = 0; i < n; i++)

        p[i][0] = t[i];

**for**(j = 1; (1 << j) <= n; j++)

    {

**for**(i = 0; i < n; i++)

        {

**if**(p[i][j - 1] != -1)

                p[i][j] = p[p[i][j - 1]][j - 1];

        }

    }

}

接下来就是查询了，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/fengchaokobe/article/details/8104784)

**int** query(**int** \*t, **int** \*l, **int** s, **int** t, **int** n, **int** p[][max])

{

**int** tmp, lg, i;

**if**(l[s] < l[t])

    {

        tmp = s;s = t;t = tmp;

    }

**for**(lg = 1; (1 << lg) <= l[s]; lg++);

**for**(i = lg; i >= 0; i--)

    {

**if**((l[s] - (1 << i)) >= l[t])

            s = p[s][i];

    }

**if**(s == t)

**return** s;

**for**(i = lg; i >= 0; i--)

    {

**if**(p[s][i] != -1 && p[s][i] != p[t][i])

        {

            s = p[s][i];

            t = p[t][i];

        }

    }

**return** t[s];

}

上面说的LCA的这种算法应该是最容易想到的，预处理过程O(NLogN)，查询O(LogN)。还有一种类似于RMQ分割法德算法，我先就不在这赘述了，以后有时间一定补上。

**结束语**

想想、写写、画画.......

后续：本文后半部分拖得周期较长，因此写的比较匆忙。如果本文的内容有任何不妥之处，请指正！

http://blog.csdn.net/pi9nc/article/details/8142384