Introduction
We often need some sort of data structure to make our algorithms faster. In this article we will discuss the **Binary Indexed Trees** structure. According to [Peter M. Fenwick](http://www.topcoder.com/tc?module=LinkTracking&link=http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol24/issue3/spe884.pdf&refer=binaryIndexedTrees), this structure was first used for data compression. Now it is often used for storing frequencies and manipulating cumulative frequency tables.

Let's define the following **problem**: We have n boxes. Possible queries are
1. add marble to box **i**
2. sum marbles from box **k** to box **l**

The naive solution has time complexity of O(1) for query 1 and O(n) for query 2. Suppose we make **m** queries. The worst case (when all queries are 2) has time complexity O(n \* m). Using some data structure (i.e. [RMQ](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor)) we can solve this problem with the worst case time complexity of O(m log n). Another approach is to use Binary Indexed Tree data structure, also with the worst time complexity O(m log n) -- but Binary Indexed Trees are much easier to code, and require less memory space, than RMQ.

Notation
  BIT - **B**inary **I**ndexed **T**ree
  MaxVal - maximum value which will have non-zero frequency
  f[i] - frequency of value with index **i**, **i** = 1 .. MaxVal
  c[i] - cumulative frequency for index **i** (f[1] + f[2] + ... + f[i])
  tree[i] - sum of frequencies stored in **BIT** with index **i** (latter will be described what index means); sometimes we will write *tree frequency* instead *sum of frequencies stored in BIT*
  num¯ - complement of integer **num** (integer where each binary digit is inverted: 0 -> 1; 1 -> 0 )
NOTE: Often we put f[0] = 0, c[0] = 0, tree[0] = 0, so sometimes I will just ignore index 0.

Basic idea
Each integer can be represented as sum of powers of two. In the same way, cumulative frequency can be represented as sum of sets of subfrequencies. In our case, each set contains some successive number of non-overlapping frequencies.

**idx** is some index of **BIT**. **r** is a position in **idx** of the last digit 1 (from left to right) in binary notation. **tree[idx]** is sum of frequencies from index (**idx** - 2^**r** + 1) to index **idx** (look at the Table 1.1 for clarification). We also write that **idx** is **responsible** for indexes from (**idx** - 2^**r** + 1) to **idx** (note that responsibility is the key in our algorithm and is the way of manipulating the tree).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **f** | 1 | 0 | 2 | 1 | 1 | 3 | 0 | 4 | 2 | 5 | 2 | 2 | 3 | 1 | 0 | 2 |
| **c** | 1 | 1 | 3 | 4 | 5 | 8 | 8 | 12 | 14 | 19 | 21 | 23 | 26 | 27 | 27 | 29 |
| **tree** | 1 | 1 | 2 | 4 | 1 | 4 | 0 | 12 | 2 | 7 | 2 | 11 | 3 | 4 | 0 | 29 |

*Table 1.1*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **tree** | 1 | 1..2 | 3 | 1..4 | 5 | 5..6 | 7 | 1..8 | 9 | 9..10 | 11 | 9..12 | 13 | 13..14 | 15 | 1..16 |

*Table 1.2 - table of responsibility*


*Image 1.3 - tree of responsibility for indexes (bar shows range of frequencies accumulated in top element)*


*Image 1.4 - tree with tree frequencies*

Suppose we are looking for cumulative frequency of index 13 (for the first 13 elements). In binary notation, 13 is equal to 1101. Accordingly, we will calculate **c[1101] = tree[1101] + tree[1100] + tree[1000]** (more about this later).

Isolating the last digit
**NOTE:** Instead of "the last non-zero digit," it will write only "the last digit."

There are times when we need to get just the last digit from a binary number, so we need an efficient way to do that. Let **num** be the integer whose last digit we want to isolate. In binary notation **num** can be represented as **a1b**, where **a** represents binary digits before the last digit and **b** represents zeroes after the last digit.

Integer **-num** is equal to **(a1b)¯ + 1 = a¯0b¯ + 1**. **b** consists of all zeroes, so **b¯** consists of all ones. Finally we have

**-num = (a1b)¯ + 1 = a¯0b¯ + 1 = a¯0(0...0)¯ + 1 = a¯0(1...1) + 1 = a¯1(0...0) = a¯1b**.

Now, we can easily isolate the last digit, using bitwise operator **AND** (in C++, Java it is **&**) with **num** and **-num**:

**a1b
&      a¯1b
--------------------
= (0...0)1(0...0)**

Read cumulative frequency
If we want to read cumulative frequency for some integer **idx**, we add to **sum tree[idx]**, substract last bit of **idx** from itself (also we can write - remove the last digit; change the last digit to zero) and repeat this while **idx** is greater than zero. We can use next function (written in C++)

int read(int idx){

 int sum = 0;

 while (idx > 0){

 sum += tree[idx];

 idx -= (idx & -idx);

 }

 return sum;

}

Example for **idx** = 13; **sum** = 0:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **iteration** | **idx** | **position of the last digit** | **idx & -idx** | **sum** |
|  |  |  |  |  |
| 1 | 13 = 1101 | 0 | 1 (2 ^0) | 3 |
| 2 | 12 = 1100 | 2 | 4 (2 ^2) | 14 |
| 3 | 8 = 1000 | 3 | 8 (2 ^3) | 26 |
| 4 | 0 = 0 | --- | --- | --- |


*Image 1.5 - arrows show path from index to zero which we use to get sum (image shows example for index 13)*

So, our result is 26. The number of iterations in this function is number if bits in **idx**, which is at most **log MaxVal**.

*Time complexity:* O(log MaxVal).
*Code length:* Up to ten lines.

Change frequency at some position and update tree
The concept is to update tree frequency at all indexes which are responsible for frequency whose value we are changing. In reading cumulative frequency at some index, we were removing the last bit and going on. In changing some frequency **val** in tree, we should increment value at the current index (the starting index is always the one whose frequency is changed) for **val**, add the last digit to index and go on while the index is less than or equal to **MaxVal**. Function in C++:

void update(int idx ,int val){

 while (idx <= MaxVal){

 tree[idx] += val;

 idx += (idx & -idx);

 }

}

Let's show example for **idx** = 5:

|  |  |  |  |
| --- | --- | --- | --- |
| **iteration** | **idx** | **position of the last digit** | **idx & -idx** |
|  |  |  |  |
| 1 | 5 = 101 | 0 | 1 (2 ^0) |
| 2 | 6 = 110 | 1 | 2 (2 ^1) |
| 3 | 8 = 1000 | 3 | 8 (2 ^3) |
| 4 | 16 = 10000 | 4 | 16 (2 ^4) |
| 5 | 32 = 100000 | --- | --- |


*Image 1.6 - Updating tree (in brackets are tree frequencies before updating); arrows show path while we update tree from index to* ***MaxVal*** *(image shows example for index 5)*

Using algorithm from above or following arrows shown in Image 1.6 we can update **BIT**.

*Time complexity:* O(log MaxVal).
*Code length:* Up to ten lines.

Read the actual frequency at a position
We've described how we can read cumulative frequency for an index. It is obvious that we can not read just **tree[idx]** to get the actual frequency for value at index **idx**. One approach is to have one aditional array, in which we will seperately store frequencies for values. Both reading and storing take O(1); memory space is linear. Sometimes it is more important to save memory, so we will show how you can get actual frequency for some value without using aditional structures.

Probably everyone can see that the actual frequency at a position **idx** can be calculated by calling function **read** twice -- **f[idx] = read(idx) - read(idx - 1)** -- just by taking the difference of two adjacent cumulative frequencies. This procedure always works in 2 \* O(log n) time. If we write a new function, we can get a bit faster algorithm, with smaller const.

If two paths from two indexes to root have the same part of path, then we can calculate the sum until the paths meet, substract stored sums and we get a sum of frequencies between that two indexes. It is pretty simple to calculate sum of frequencies between adjacent indexes, or read the actual frequency at a given index.

Mark given index with **x**, its predecessor with **y**. We can represent (binary notation) **y** as **a0b**, where **b** consists of all ones. Then, **x** will be **a1b¯** (note that **b¯** consists all zeros). Using our algorithm for getting **sum** of some index, let it be **x**, in first iteration we remove the last digit, so after the first iteration **x** will be **a0b¯**, mark a new value with **z**.

Repeat the same process with **y**. Using our function for reading **sum** we will remove the last digits from the number (one by one). After several steps, our **y** will become (just to remind, it was **a0b**) **a0b¯**, which is the same as **z**. Now, we can write our algorithm. Note that the only exception is when **x** is equal to 0. Function in C++:

int readSingle(int idx){

int sum = tree[idx]; *// sum will be decreased*

if (idx > 0){ *// special case*

 int z = idx - (idx & -idx); *// make z first*

 idx--; *// idx is no important any more, so instead y, you can use idx*

 while (idx != z){ *// at some iteration idx (y) will become z*

 sum -= tree[idx];

*// substruct tree frequency which is between y and "the same path"*

 idx -= (idx & -idx);

 }

}

return sum;

}

Here's an example for getting the actual frequency for index 12:

First, we will calculate **z = 12 - (12 & -12) = 8**, **sum = 11**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **iteration** | **y** | **position of the last digit** | **y & -y** | **sum** |
|  |  |  |  |  |
| 1 | 11 = 1011 | 0 | 1 (2 ^0) | 9 |
| 2 | 10 = 1010 | 1 | 2 (2 ^1) | 2 |
| 3 | 8 = 1000 | --- | --- | --- |


*Image 1.7 - read actual frequency at some index in BIT
(image shows example for index 12)*

Let's compare algorithm for reading actual frequency at some index when we twice use function **read** and the algorithm written above. Note that for each odd number, the algorithm will work in const time O(1), without any iteration. For almost every even number **idx**, it will work in c \* O(log idx), where c is strictly less than 1, compare to **read(idx) - read(idx - 1),** which will work in c1 \* O(log idx), where c1 is **always** greater than 1.

*Time complexity:* c \* O(log MaxVal), where c is less than 1.
*Code length:* Up to fifteen lines.

Scaling the entire tree by a constant factor
Sometimes we want to scale our tree by some factor. With the procedures described above it is very simple. If we want to scale by some factor **c**, then each index **idx** should be updated by **-(c - 1) \* readSingle(idx) / c** (because **f[idx] - (c - 1) \* f[idx] / c = f[idx] / c**). Simple function in C++:

void scale(int c){

 for (int i = 1 ; i <= MaxVal ; i++)

 update(-(c - 1) \* readSingle(i) / c , i);

}

This can also be done more quickly. Factor is linear operation. Each tree frequency is a linear composition of some frequencies. If we scale each frequency for some factor, we also scaled tree frequency for the same factor. Instead of rewriting the procedure above, which has time complexity O(MaxVal \* log MaxVal), we can achieve time complexity of O(MaxVal):

void scale(int c){

 for (int i = 1 ; i <= MaxVal ; i++)

 tree[i] = tree[i] / c;

}

*Time complexity:* O(MaxVal).
*Code length:* Just a few lines.

Find index with given cumulative frequency
The naive and most simple solution for finding an index with a given cumultive frequency is just simply iterating through all indexes, calculating cumulative frequency, and checking if it's equal to the given value. In case of negative frequencies it is the only solution. However, if we have only non-negative frequencies in our tree (that means cumulative frequencies for greater indexes are not smaller) we can figure out logarithmic algorithm, which is modification of [binary search](http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch). We go through all bits (starting with the highest one), make the index, compare the cumulative frequency of the current index and given value and, according to the outcome, take the lower or higher half of the interval (just like in binary search). Function in C++:

*// if in tree exists more than one index with a same*

*// cumulative frequency, this procedure will return*

*// some of them (we do not know which one)*

*// bitMask - initialy, it is the greatest bit of MaxVal*

*// bitMask store interval which should be searched*

int find(int cumFre){

 int idx = 0; *// this var is result of function*

 while ((bitMask != 0) && (idx < MaxVal)){ *// nobody likes overflow :)*

 int tIdx = idx + bitMask; *// we make midpoint of interval*

 if (cumFre == tree[tIdx]) *// if it is equal, we just return idx*

 return tIdx;

 else if (cumFre > tree[tIdx]){

 *// if tree frequency "can fit" into cumFre,*

 *// then include it*

 idx = tIdx; *// update index*

 cumFre -= tree[tIdx]; *// set frequency for next loop*

 }

 bitMask >>= 1; *// half current interval*

 }

 if (cumFre != 0) *// maybe given cumulative frequency doesn't exist*

 return -1;

 else

 return idx;

}

*// if in tree exists more than one index with a same*

*// cumulative frequency, this procedure will return*

*// the greatest one*

int findG(int cumFre){

 int idx = 0;

 while ((bitMask != 0) && (idx < MaxVal)){

 int tIdx = idx + bitMask;

 if (cumFre >= tree[tIdx]){

 *// if current cumulative frequency is equal to cumFre,*

 *// we are still looking for higher index (if exists)*

 idx = tIdx;

 cumFre -= tree[tIdx];

 }

 bitMask >>= 1;

 }

 if (cumFre != 0)

 return -1;

 else

 return idx;

}

Example for cumulative frequency 21 and function **find**:

|  |  |
| --- | --- |
| **First iteration** | tIdx is 16; tree[16] is greater than 21; half bitMask and continue |
| **Second iteration** | tIdx is 8; tree[8] is less than 21, so we should include first 8 indexes in result, remember idx because we surely know it is part of result; subtract tree[8] of cumFre (we do not want to look for the same cumulative frequency again - we are looking for another cumulative frequency in the rest/another part of tree); half bitMask and contiue |
| **Third iteration** | tIdx is 12; tree[12] is greater than 9 (there is no way to overlap interval 1-8, in this example, with some further intervals, because only interval 1-16 can overlap); half bitMask and continue |
| **Forth iteration** | tIdx is 10; tree[10] is less than 9, so we should update values; half bitMask and continue |
| **Fifth iteration** | tIdx is 11; tree[11] is equal to 2; return index (tIdx) |

*Time complexity:* O(log MaxVal).
*Code length:* Up to twenty lines.

2D BIT
BIT can be used as a multi-dimensional data structure. Suppose you have a plane with dots (with non-negative coordinates). You make three queries:

1. set dot at (x , y)
2. remove dot from (x , y)
3. count number of dots in rectangle (0 , 0), (x , y) - where (0 , 0) if down-left corner, (x , y) is up-right corner and sides are parallel to x-axis and y-axis.

If **m** is the number of queries, **max\_x** is maximum x coordinate, and **max\_y** is maximum y coordinate, then the problem should be solved in O(m \* log (max\_x) \* log (max\_y)). In this case, each element of the tree will contain array - (**tree[max\_x][max\_y]**). Updating indexes of x-coordinate is the same as before. For example, suppose we are setting/removing dot (**a** , **b**). We will call **update(a , b , 1)/update(a , b , -1)**, where **update** is:

void update(int x , int y , int val){

 while (x <= max\_x){

 updatey(x , y , val);

 *// this function should update array tree[x]*

 x += (x & -x);

 }

}

The function **updatey** is the "same" as function **update**:

void updatey(int x , int y , int val){

 while (y <= max\_y){

 tree[x][y] += val;

 y += (y & -y);

 }

}

It can be written in one function/procedure:

void update(int x , int y , int val){

 int y1;

 while (x <= max\_x){

 y1 = y;

 while (y1 <= max\_y){

 tree[x][y1] += val;

 y1 += (y1 & -y1);

 }

 x += (x & -x);

 }

}


*Image 1.8 - BIT is array of arrays, so this is two-dimensional BIT (size 16 x 8).
Blue fields are fields which we should update when we are updating index (5 , 3).*

The modification for other functions is very similar. Also, note that BIT can be used as an n-dimensional data structure.

Sample problem

* [SRM 310 - FloatingMedian](http://www.topcoder.com/stat?c=problem_statement&pm=6551&rd=9990)
* Problem 2:
**Statement:**
There is an array of **n** cards. Each card is putted face down on table. You have two queries:
  1. T i j (turn cards from index i to index j, include i-th and j-th card - card which was face down will be face up; card which was face up will be face down)
  2. Q i (answer 0 if i-th card is face down else answer 1)

**Solution:**
This has solution for each query (and 1 and 2) has time complexity O(log n). In array **f** (of length **n + 1**) we will store each query **T (i , j)** - we set **f[i]++** and **f[j + 1]--**. For each card **k** between **i** and **j** (include **i** and **j**) sum **f[1] + f[2] + ... + f[k]** will be increased for 1, for all others will be same as before (look at the image 2.0 for clarification), so our solution will be described sum (which is same as cumulative frequency) module 2.


*Image 2.0*

Use **BIT** to store (increase/decrease) frequency and read cumulative frequency.

Conclusion

* Binary Indexed Trees are very easy to code.
* Each query on Binary Indexed Tree takes constant or logarithmic time.
* Binary Indexeds Tree require linear memory space.
* You can use it as an n-dimensional data structure.

Introduction
The problem of finding the Lowest Common Ancestor (LCA) of a pair of nodes in a rooted tree has been studied more carefully in the second part of the 20th century and now is fairly basic in algorithmic graph theory. This problem is interesting not only for the tricky algorithms that can be used to solve it, but for its numerous applications in string processing and computational biology, for example, where LCA is used with suffix trees or other tree-like structures. [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=) were the first to study this problem more attentively and they showed that after linear preprocessing of the input tree LCA, queries can be answered in constant time. Their work has since been extended, and this tutorial will present many interesting approaches that can be used in other kinds of problems as well.

Let's consider a less abstract example of LCA: the tree of life. It's a well-known fact that the current habitants of Earth evolved from other species. This evolving structure can be represented as a tree, in which nodes represent species, and the sons of some node represent the directly evolved species. Now species with similar characteristics are divided into groups. By finding the LCA of some nodes in this tree we can actually find the common parent of two species, and we can determine that the similar characteristics they share are inherited from that parent.

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. We will see later that the LCA problem can be reduced to a restricted version of an RMQ problem, in which consecutive array elements differ by exactly 1.

However, RMQs are not only used with LCA. They have an important role in string preprocessing, where they are used with suffix arrays (a new data structure that supports string searches almost as fast as suffix trees, but uses less memory and less coding effort).

In this tutorial we will first talk about RMQ. We will present many approaches that solve the problem -- some slower but easier to code, and others faster. In the second part we will talk about the strong relation between LCA and RMQ. First we will review two easy approaches for LCA that don't use RMQ; then show that the RMQ and LCA problems are equivalent; and, at the end, we'll look at how the RMQ problem can be reduced to its restricted version, as well as show a fast algorithm for this particular case.

Notations
Suppose that an algorithm has preprocessing time **f(n)** and query time **g(n)**. The notation for the overall complexity for the algorithm is **<f(n), g(n)>**.

We will note the position of the element with the minimum value in some array **A** between indices **i** and **j** with **RMQA(i, j)**.

The furthest node from the root that is an ancestor of both **u** and **v** in some rooted  tree **T** is **LCAT(u, v)**.

Range Minimum Query(RMQ)
Given an array **A[0, N-1]** find the position of  the element with the minimum value between two given indices.



Trivial algorithms for RMQ
For every pair of indices **(i, j)** store the value of **RMQA(i, j)** in a table **M[0, N-1][0, N-1]**. Trivial computation will lead us to an **<O(N3), O(1)>** complexity. However, by using an easy dynamic programming approach we can reduce the complexity to **<O(N2), O(1)>**. The preprocessing function will look something like this:

 **void** process1(**int** M[MAXN][MAXN], **int** A[MAXN], **int** N)

 {

   **int** i, j;

     **for** (i =0; i < N; i++)

         M[i][i] = i;

     **for** (i = 0; i < N; i++)

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

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

               M[i][j] = M[i][j - 1];

             **else**

                 M[i][j] = j;

 }

This trivial algorithm is quite slow and uses **O(N2)** memory, so it won't work for large cases.

An <O(N), O(sqrt(N))> solution
An interesting idea is to split the vector in **sqrt(N)** pieces. We will keep in a vector **M[0, sqrt(N)-1]** the position for the minimum value for each section. **M** can be easily preprocessed in **O(N)**. Here is an example:



Now let's see how can we compute **RMQA(i, j)**. The idea is to get the overall minimum from the **sqrt(N)** sections that lie inside the interval, and from the end and the beginning of the first and the last sections that intersect the bounds of the interval. To get **RMQA(2,7)** in the above example we should compare **A[2]**, **A[M[1]]**, **A[6]** and **A[7]** and get the position of the minimum value. It's easy to see that this algorithm doesn't make more than **3 \* sqrt(N)** operations per query.

The main advantages of this approach are that is to quick to code (a plus for TopCoder-style competitions) and that you can adapt it to the dynamic version of the problem (where you can change the elements of the array between queries).

Sparse Table (ST) algorithm
A better approach is to preprocess **RMQ** for sub arrays of length **2k** using dynamic programming. We will keep an array **M[0, N-1][0, logN]** where **M[i][j]** is the index of the minimum value in the sub array starting at **i** having length **2j**. Here is an example:



For computing **M[i][j]** we must search for the minimum value in the first and second half of the interval. It's obvious that the small pieces have **2j - 1** length, so the recurrence is:



The preprocessing function will look something like this:

 **void** process2(**int** M[MAXN][LOGMAXN], **int** A[MAXN], **int** N)

 {

 **int** i, j;

 //initialize **M** for the intervals with length **1**

 **for** (i = 0; i < N; i++)

         M[i][0] = i;

 //compute values from smaller to bigger intervals

     **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))][j - 1]])

         M[i][j] = M[i][j - 1];

             **else**

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

 }

Once we have these values preprocessed, let's show how we can use them to calculate **RMQA(i, j)**. The idea is to select two blocks that entirely cover the interval **[i..j]** and  find the minimum between them. Let **k = [log(j - i + 1)]**. For computing **RMQA(i, j)** we can use the following formula:



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

Segment trees
For solving the RMQ problem we can also use segment trees. A segment tree is a heap-like data structure that can be used for making update/query operations upon array intervals in logarithmical time. We define the segment tree for the interval **[i, j]** in the following recursive manner:

* the first node will hold the information for the interval **[i, j]**
* if i<j the left and right son will hold the information for the intervals **[i, (i+j)/2]** and **[(i+j)/2+1, j]**

Notice that the height of a segment tree for an interval with **N** elements is **[logN] + 1**. Here is how a segment tree for the interval **[0, 9]** would look like:



The segment tree has the same structure as a heap, so if we have a node numbered **x** that is not a leaf the left son of **x** is **2\*x** and the right son **2\*x+1**.

For solving the RMQ problem using segment trees we should use an array **M[1, 2 \* 2[logN] + 1]** where **M[i]** holds the minimum value position in the interval assigned to node **i**. At the beginning all elements in **M** should be **-1**. The tree should be initialized with the following function (**b** and **e** are the bounds of the current interval):

 **void** initialize(**int**node, **int** b, **int** e, **int** M[MAXIND], **int** A[MAXN], **int** N)

 {

 **if** (b == e)

 M[node] = b;

    **else**

      {

 //compute the values in the left and right subtrees

     initialize(2 \* node, b, (b + e) / 2, M, A, N);

     initialize(2 \* node + 1, (b + e) / 2 + 1, e, M, A, N);

 //search for the minimum value in the first and

 //second half of the interval

     **if** (A[M[2 \* node]] <= A[M[2 \* node + 1]])

         M[node] = M[2 \* node];

      **else**

         M[node] = M[2 \* node + 1];

     }

 }

The function above reflects the way the tree is constructed. When calculating the minimum position for some interval we should look at the values of the sons. You should call the function with **node = 1**, **b = 0** and **e  = N-1**.

We can now start making queries. If we want to find the position of the minimum value in some interval **[i, j]** we should use the next easy function:

 **int** query(**int** node, **int** b, **int** e, **int** M[MAXIND], **int** A[MAXN], **int** i, **int** j)

 {

 **int** p1, p2;

 //if the current interval doesn't intersect

 //the query interval return **-1**

     **if** (i > e || j < b)

         **return** -1;

 //if the current interval is included in

 //the query interval return **M[node]**

     **if** (b >= i && e <= j)

         **return** M[node];

 //compute the minimum position in the

 //left and right part of the interval

     p1 = query(2 \* node, b, (b + e) / 2, M, A, i, j);

     p2 = query(2 \* node + 1, (b + e) / 2 + 1, e, M, A, i, j);

 //return the position where the overall

 //minimum is

     **if** (p1 == -1)

         **return** M[node] = p2;

    **if** (p2 == -1)

         **return** M[node] = p1;

     **if** (A[p1] <= A[p2])

         **return** M[node] = p1;

     **return** M[node] = p2;

 }

You should call this function with **node = 1**, **b = 0** and **e = N - 1**, because the interval assigned to the first node is **[0, N-1]**.

It's easy to see that any query is done in **O(log N)**. Notice that we stop when we reach completely in/out intervals, so our path in the tree should split only one time.

Using segment trees we get an **<O(N), O(logN)>** algorithm. Segment trees are very powerful, not only because they can be used for RMQ. They are a very flexible data structure, can solve even the dynamic version of RMQ problem, and have numerous applications in range searching problems.

Lowest Common Ancestor (LCA)
Given a rooted tree **T** and two nodes **u** and **v,** find the furthest node from the root that is an ancestor for both **u** and **v**. Here is an example (the root of the tree will be node 1 for all examples in this editorial):



An <O(N), O(sqrt(N))> solution
Dividing our input into equal-sized parts proves to be an interesting way to solve the RMQ problem. This method can be adapted for the LCA problem as well. The idea is to split the tree in **sqrt(H)** parts, were **H** is the height of the tree. Thus, the first section will contain the levels numbered from **0 to sqrt(H) - 1**, the second will contain the levels numbered from **sqrt(H) to 2 \* sqrt(H) - 1**, and so on. Here is how the tree in the example should be divided:



Now, for each node, we should know the ancestor that is situated on the last level of the upper next section. We will preprocess this values in an array **P[1, MAXN]**. Here is how **P** should look like for the tree in the example (for simplity, for every node **i** in the first section let **P[i] = 1**):



Notice that for the nodes situated on the levels that are the first ones in some sections, **P[i] = T[i]**. We can preprocess **P** using a depth first search (**T[i]** is the father of node **i** in the tree, **nr** is **[sqrt(H)]** and **L[i]** is the level of the node **i**):

 **void** dfs(**int** node, **int** T[MAXN], **int** N, **int** P[MAXN], **int** L[MAXN], **int** nr) {

 **int** k;

 //if node is situated in the first

 //section then **P[node] = 1**

 //if node is situated at the beginning

 //of some section then **P[node] = T[node]**

 //if none of those two cases occurs, then

 //**P[node] = P[T[node]]**

 **if** (L[node] < nr)

 P[node] = 1;

 **else**

  **if**(!(L[node] % nr))

 P[node] = T[node];

  **else**

 P[node] = P[T[node]];

 **for** each son k of node

 dfs(k, T, N, P, L, nr);

 }

Now, we can easily make queries. For finding **LCA(x, y)** we we will first find in what section it lays, and then trivially compute it. Here is the code:

 int LCA(**int** T[MAXN], **int** P[MAXN], **int** L[MAXN], **int** x, **int** y)

 {

 //as long as the node in the next section of

 //x and y is not one common ancestor

 //we get the node situated on the smaller

 //lever closer

     **while** (P[x] != P[y])

        **if** (L[x] > L[y])

 x = P[x];

**else**

      y = P[y];

 //now they are in the same section, so we trivially compute the LCA

 **while** (x != y)

         **if** (L[x] > L[y])

           x = T[x];

         **else**

           y = T[y];

    **return** x;

 }

This function makes at most **2 \* sqrt(H)** operations. Using this approach we get an **<O(N), O(sqrt(H))>** algorithm, where **H** is the height of the tree. In the worst case **H = N**, so the overall complexity is **<O(N), O(sqrt(N))>**. The main advantage of this algorithm is quick coding (an average Division 1 coder shouldn't need more than 15 minutes to code it).

Another easy solution in <O(N logN, O(logN)>
If we need a faster solution for this problem we could use dynamic programming. First, let's compute a table P[1,N][1,logN] where P[i][j] is the 2j-th ancestor of i. For computing this value we may use the following recursion:



The preprocessing function should look like this:

 **void** process3(**int** N, **int** T[MAXN], **int** P[MAXN][LOGMAXN])

 {

     **int** i, j;

 //we initialize every element in P with -1

     **for** (i = 0; i < N; i++)

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

      P[i][j] = -1;

 //the first ancestor of every node i is T[i]

     **for** (i = 0; i < N; i++)

         P[i][0] = T[i];

 //bottom up dynamic programing

 **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];

 }

This takes **O(N logN)** time and space. Now let's see how we can make queries. Let **L[i]** be the level of node **i** in the tree. We must observe that if **p** and **q** are on the same level in the tree we can compute **LCA(p, q)** using a meta-binary search. So, for every power **j** of **2 (**between **log(L[p])** and **0,** in descending order), if **P[p][j] != P[q][j]** then we know that **LCA(p, q)** is on a higher level and we will continue searching for **LCA(p = P[p][j], q = P[q][j])**. At the end, both **p** and **q** will have the same father, so return **T[p]**. Let's see what happens if **L[p] != L[q]**. Assume, without loss of generality, that **L[p] < L[q]**. We can use the same meta-binary search  for finding the ancestor of **p** situated on the same level with **q**, and then we can compute the **LCA**  as described below. Here is how the query function should look:

 **int** query(**int** N, **int** P[MAXN][LOGMAXN], **int** T[MAXN],

 **int** L[MAXN], **int** p, **int** q)

 {

     **int** tmp, log, i;

 //if p is situated on a higher level than q then we swap them

 **if** (L[p] < L[q])

 tmp = p, p = q, q = tmp;

 //we compute the value of [log(L[p)]

     **for** (log = 1; 1 << log <= L[p]; log++);

     log--;

 //we find the ancestor of node p situated on the same level

 //with q using the values in P

     **for** (i = log; i >= 0; i--)

      **if** (L[p] - (1 << i) >= L[q])

          **p** = P[p][i];

     **if** (p == q)

         return p;

 //we compute LCA(p, q) using the values in P

 **for** (i = log; i >= 0; i--)

      **if** (P[p][i] != -1 && P[p][i] != P[q][i])

         p = P[p][i], q = P[q][i];

     **return** T[p];

 }

Now, we can see that this function makes at most **2 \* log(H)** operations, where **H** is the height of the tree. In the worst case **H = N**, so the overall complexity of this algorithm is **<O(N logN), O(logN)>**. This solution is easy to code too, and it's faster than the previous one.

Reduction from LCA to RMQ
Now, let's show how we can use RMQ for computing LCA queries. Actually, we will reduce the LCA problem to RMQ in linear time, so every algorithm that solves the RMQ problem will solve the LCA problem too. Let's show how this reduction can be done using an example:




click to enlarge image

Notice that **LCAT(u, v)** is the closest node from the root encountered between the visits of **u** and **v** during a depth first search of **T**. So, we can consider all nodes between any two indices of **u** and **v** in the Euler Tour of the tree and then find the node situated on the smallest level between them. For this, we must build three arrays:

* **E[1, 2\*N-1]** - the nodes visited in an Euler Tour of **T**; **E[i]** is the label of **i-th** visited node in the tour
* **L[1, 2\*N-1]** - the levels of the nodes visited in the Euler Tour; **L[i]** is the level of node **E[i]**
* **H[1, N] - H[i]** is the index of the first occurrence of node **i** in **E** (any occurrence would be good, so it's not bad if we consider the first one)

Assume that **H[u] < H[v]** (otherwise you must swap **u** and **v**). It's easy to see that the nodes between the first occurrence of **u** and the first occurrence of **v** are **E[H[u]...H[v]]**. Now, we must find the node situated on the smallest level. For this, we can use **RMQ**. So, **LCAT(u, v) = E[RMQL(H[u], H[v])]** (remember that RMQ returns the index). Here is how **E**, **L** and **H** should look for the example:


click to enlarge image

Notice that consecutive elements in L differ by exactly 1.

From RMQ to LCA
We have shown that the LCA problem can be reduced to RMQ in linear time. Here we will show how we can reduce the RMQ problem to LCA. This means that we actually can reduce the general RMQ to the restricted version of the problem (where consecutive elements in the array differ by exactly 1). For this we should use cartesian trees.

A Cartesian Tree of an array **A[0, N - 1]** is a binary tree **C(A)** whose root is a minimum element of **A**, labeled with the position **i** of this minimum. The left child of the root is the Cartesian Tree of **A[0, i - 1]** if **i > 0**, otherwise there's no child. The right child is defined similary for **A[i + 1, N - 1]**. Note that the Cartesian Tree is not necessarily unique if **A** contains equal elements. In this tutorial the first appearance of the minimum value will be used, thus the Cartesian Tree will be unique.  It's easy to see now that **RMQA(i, j) = LCAC(i, j)**.

Here is an example:





Now we only have to compute **C(A)** in linear time. This can be done using a stack. At the beginning the stack is empty. We will then insert the elements of **A** in the stack. At the **i-th** step **A[i]** will be added next to the last element in the stack that has a smaller or equal value to **A[i]**, and all the greater elements will be removed. The element that was in the stack on the position of **A[i]** before the insertion was done will become the left son of **i**, and **A[i]** will become the right son of the smaller element behind him. At every step the first element in the stack is the root of the cartesian tree. It's easier to build the tree if the stack will hold the indexes of the elements, and not their value.

Here is how the stack will look at each step for the example above:

|  |  |  |
| --- | --- | --- |
| 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. |

Note that every element in **A** is only added once and removed at most once, so the complexity of this algorithm is **O(N)**. Here is how the tree-processing function will look:

 **void** computeTree(**int** A[MAXN], **int** N, **int** T[MAXN])

 {

     **int** st[MAXN], i, k, top = -1;

 //we start with an empty stack

 //at step **i** we insert **A[i]** in the stack

     **for** (i = 0; i < N; i++)

     {

 //compute the position of the first element that is

 //equal or smaller than **A[i]**

         k = top;

         **while** (k >= 0 && A[st[k]] > A[i])

             k--;

 //we modify the tree as explained above

        **if** (k != -1)

             T[i] = st[k];

        **if** (k < top)

             T[st[k + 1]] = i;

 //we insert **A[i]** in the stack and remove

 //any bigger elements

         st[++k] = i;

         top = k;

     }

 //the first element in the stack is the root of

 //the tree, so it has no father

     T[st[0]] = -1;

 }

An<O(N), O(1)> algorithm for the restricted RMQ
Now we know that the general RMQ problem can be reduced to the restricted version using LCA. Here, consecutive elements in the array differ by exactly 1. We can use this and give a fast **<O(N), O(1)>** algorithm. From now we will solve the RMQ problem for an array **A[0, N - 1]** where  **|A[i] - A[i + 1]| = 1**, **i = [1, N - 1]**. We transform **A** in a binary array with **N-1** elements, where **A[i] = A[i] - A[i + 1]**. It's obvious that elements in **A** can be just **+1** or **-1**. Notice that the old value of **A[i]** is now the sum of **A[1]**, **A[2]** .. **A[i]** plus the old **A[0]**. However, we won't need the old values from now on.

To solve this restricted version of the problem we need to partition **A** into blocks of size **l = [(log N) / 2]**. Let **A'[i]** be the minimum value for the **i-th** block in **A** and **B[i]** be the position of this minimum value in **A**. Both **A** and **B** are **N/l** long. Now, we preprocess **A'** using the ST algorithm described in Section1. This will take **O(N/l \* log(N/l)) = O(N)** time and space. After this preprocessing we can make queries that span over several blocks in **O(1)**. It remains now to show how the in-block queries can be made. Note that the length of a block is **l = [(log N) / 2]**, which is quite small. Also, note that **A** is a binary array. The total number of binary arrays of size **l** is **2l=sqrt(N)**. So, for each binary block of size **l** we need to lock up in a table **P** the value for RMQ between every pair of indices. This can be trivially computed in **O(sqrt(N)\*l2)=O(N**) time and space. To index table **P**, preprocess the type of each block in **A** and store it in array **T[1, N/l]**. The block type is a binary number obtained by replacing -1 with 0 and +1 with 1.

Now, to answer **RMQA(i, j)** we have two cases:

* **i** and **j** are in the same block, so we use the value computed in **P** and **T**
* **i** and **j** are in different blocks,  so we compute three values: the minimum from **i** to the end of **i's** block using **P** and **T**, the minimum of all blocks between **i's** and j**'s** block using precomputed queries  on **A'** and the minimum from the begining of **j's** block to **j**, again using **T** and **P**; finally return the position where the overall minimum is using the three values you just computed.

Conclusion
RMQ and LCA are strongly related problems that can be reduced one to another. Many algorithms can be used to solve them, and they can be adapted to other kind of problems as well.

Here are some training problems for segment trees, LCA and RMQ: