Another variation of segment tree is sorted segment tree.

**Problem**

Suppose we have an array **A** of **N** elements and we need to answer **Q** queries. In each query there will be 4 integers **L**, **R**, **X** and **Y** and we have to tell that how many numbers exist in the range of **L ... R** that lie between **X** and **Y** (both inclusive).

**Approach**

At each node store all the elements of that node's range in sorted order. For eg if we have an array **A** = **{3, 5, 4, 3, 2, 9, 7, 6}** then following will be it's sorted segment tree.   
As we can see that at each level there are total **N** elements and there are total **logN** levels therefore space complexity will be **O(NlogN)**. After we create this segment tree than we can query over this tree and visit all the nodes whose range completely lies inside the query range and than find out how many values in that node lie between **X** and **Y**. Visiting all nodes can be achieved by same logic as discussed in previous problems so it will take **O(logN)** time. Checking at each node that how many values lie between **X** and **Y** can be done with the help of binary search which also takes **O(logN)** time. Hence total time complexity of each query will be **O(log2N)**

**Structure**

At each node we need to store all elements in that node's range in sorted order. Since each node is having different range hence each node should be allocated memory equal to its node's range. Hence we will use **vector (C++ STL)** for each node.Hence following will be the structure :

vector <int> segTree[4\*N] // size of segment tree is 4 times size of given input

**Build Tree**

If we are at leaf node than there will be a single element and a range of single element is sorted hence we will push that element into current node's vector.   
If we are at non leaf node then first build left and right child and merge their elements using merge function of merge sort to get all elements of current node in sorted order.   
Following is code snippet to build segment tree :

// index is location of current node

// indexLeft is location of left child of current node

// indexRight is location of right child of current node

void merge(int index, int indexLeft, int indexRight)

{

int i, j;

for (i = j = 0; i < segTree[indexLeft].size() && j < segTree[indexRight].size(); ) {

if (segTree[indexLeft][i] <= segTree[indexRight][j]) {

segTree[index].push\_back(segTree[indexLeft][i]);

++i;

}

else {

segTree[index].push\_back(segTree[indexRight][j]);

++j;

}

}

while (i < segTree[indexLeft].size()) {

segTree[index].push\_back(segTree[indexLeft][i]);

++i;

}

while (j < segTree[indexRight].size()) {

segTree[index].push\_back(segTree[indexRight][j]);

++j;

}

}

// rangeLeft will be left most index of current range

// rangeRight will be right most index of current range

// index will be location of current node in array of segTree

void build\_tree (int rangeLeft, int rangeRight, int index)

{

if (rangeLeft == rangeRight) {

segTree[index].push\_back(A[rangeLeft]);

return;

}

int mid = (rangeLeft + rangeRight) / 2;

build\_tree(rangeLeft, mid, 2\*index);

build\_tree(mid+1, rangeRight, 2\*index+1);

merge(index, 2\*index, 2\*index+1);

}

**Query Tree**

Start from root node and follow steps given below.

* If current node's range is completely outside query's range then return 0.
* If current node's range is completely inside query's range then find total number of elements in between **X** and **Y** by using binary search.
* Else query on both children and return sum of their result.

Following is code snippet to query segment tree :

// queryLeft will be left most index of query range

// queryRight will be right most index of query range

// X and Y are query parameters as discussed above

// rangeLeft will be left most index of current range

// rangeRight will be right most index of current range

// index will be location of current node in array of segTree

int query\_tree (int queryLeft, int queryRight, int X, int Y, int rangeLeft, int rangeRight, int index)

{

if (queryRight < rangeLeft || rangeRight < queryLeft)

return 0;

if (queryLeft <= rangeLeft && rangeRight <= queryRight)

return upper\_bound(segTree[index].begin(), segTree[index].end(), Y) - lower\_bound(segTree[index].begin(), segTree[index].end(), X);

int mid = (rangeLeft + rangeRight) / 2;

return query\_tree(queryLeft, queryRight, X, Y, rangeLeft, mid, 2\*index) + query\_tree(queryLeft, queryRight, X, Y, mid+1, rangeRight, 2\*index+1);

}