```
// Basic Segment Tree (Without Propagation)
vi tree;

void segment_build(int pos, int L, int R) {
   tree[pos] = 0;            //init is here, no need to manual init
   if(L==R) {
      tree[pos] = arr[L];
      return;
   }
   int mid = (L+R)/2;
   segment_build(pos*2, L, mid);
   segment_build(pos*2+1, mid+1, R);
   tree[pos] = tree[pos*2] * tree[pos*2+1]; //depends on usage
}

void segment_update(int pos, int L, int R, int i, int val) {
   if(L==R) {
      tree[pos] = val;
      return;
   }
   int mid = (L+R)/2;
   if(i <= mid) segment_update(pos*2, L, mid, i, val);
   else segment_update(pos*2+1, mid+1, R, i, val);
   tree[pos] = tree[pos*2] * tree[pos*2+1];       //depends on usage
}

int segment_query(int pos, int L, int R, int l, int r) {
   if(R < l || r < L)
         return 1;  //depends on usage (return 0)
   if(l <= L && R <= r)
         return tree[pos];    //depends on usage
   int mid = (L+R)/2;
   int x = segment_query(pos*2, L, mid, l, r);
   int y = segment_query(pos*2+1, mid+1, R, l, r);
   return x*y;               //depends on usage
}


// Segment Tree Lazy Propagation (Without Propagation Update)

vector<pair<ull, ull> > tree;

// (l, r) : tree segment, (x, y) : update segment
void update(ll pos, ll l, ll r, ll x, ll y, ll val) {
      if(y < l || x > r)
            return;
      if(x <= l && r <= y) {          // Tree segment in update segment
            tree[pos].fi += (r-l+1)*val;
            tree[pos].se += val;      // Propagate
            return;
      }

      ll mid = (l+r)/2LL;
      update(pos*2LL, l, mid, x, y, val);
      update(pos*2LL + 1, mid+1, r, x, y, val);
      tree[pos].fi = tree[pos*2].fi + tree[pos*2+1].fi + (r-l+1)*tree[pos].se;
}

// Pass propagate value through carry
```

```
ll query(ll pos, ll l, ll r, ll x, ll y, ll carry) {
	if(y < l || x > r)
		return 0;
	if(x <= l && r <= y)
		return tree[pos].fi + (carry * (r-l+1));

	ll mid = (l+r)/2LL;
	ll lft = query(pos*2LL, l, mid, x, y, carry + tree[pos].se);
	ll rht = query(pos*2LL + 1, mid+1, r, x, y, carry + tree[pos].se);
	return lft + rht;
}



// Segment Tree Laze (With Propagation Update)
struct node {
   int val, prop;
};

// Prop :
// 0 : No prop operation
// 1 : Prop operation should be done

node tree[409000];

void init(int L, int R, int pos) {
   if(L == R) {
      tree[pos].val = 0;
      tree[pos].prop = 0;
      return;
   }

   int mid = (L+R)>>1;
   init(L, mid, pos<<1);
   init(mid+1, R, pos<<1|1);

   tree[pos].val = 0;
   tree[pos].prop = 0;
}

int flipProp(int parentVal, int childVal) {
   if(parentVal == childVal)
      return 0;
   return parentVal;
}

void propagate(int L, int R, int pos) {
   if(tree[pos].prop == 0 || L == R)       // If no propagation tag
      return;                              // or leaf node, then no need to change

   int mid = (L+R)>>1;
   tree[pos<<1].val = (mid-L+1) - tree[pos<<1].val;        // Set left & right child value
   tree[pos<<1|1].val = (R-mid) – tree[pos<<1|1].val;

   tree[pos<<1].prop = flipProp(tree[pos].prop, tree[pos<<1].prop);    // Flip child prop according to problem
   tree[pos<<1|1].prop = flipProp(tree[pos].prop, tree[pos<<1|1].prop);
   tree[pos].prop = 0;         // Clear parent propagation tag
}
```

```
void update(int L, int R, int l, int r, int pos) {
   if(r < L || R < l)
      return;
   propagate(L, R, pos);
   if(l <= L && R <= r) {
      tree[pos].val = (R-L+1) - tree[pos].val;    // Value updated
      tree[pos].prop = 1;                         // Propagation tag set
      return;
   }

   int mid = (L+R)>>1;
   update(L, mid, l, r, pos<<1);
   update(mid+1, R, l, r, pos<<1|1);
   tree[pos].val = tree[pos<<1].val + tree[pos<<1|1].val;
}

int querySum(int L, int R, int l, int r, int pos) {
   if(r < L || R < l)
      return 0;
   propagate(L, R, pos);
   if(l <= L && R <= r)
      return tree[pos].val;

   int mid = (L+R)>>1;
   int lft = querySum(L, mid, l, r, pos<<1);
   int rht = querySum(mid+1, R, l, r, pos<<1|1);
   return lft+rht;
}


// Segment Tree Max Sum
// Node Structures:
// Update with tree[pos] = node(-INF) if out of range

struct node {
      ll sum, prefix, suffix, ans;

      node(ll val = 0) {
            sum = prefix = suffix = ans = val;
      }

      void merge(node left, node right) {
            sum = left.sum + right.sum;
            prefix = max(left.prefix, left.sum+right.prefix);
            suffix = max(right.suffix, right.sum+left.suffix);
            ans = max(left.ans, max(right.ans, left.suffix+right.prefix));
      }
};

// Segment Tree Line Sweep with Path Compression
// LightOJ 1120 - Rectangle Union

struct Node {
      ll yMin, yMax, x, val;

      Node(ll a, ll b, ll c, ll d) {
            this->yMin = a;
            this->yMax = b;
            this->x = c;
```

```
            this->val = d;
        }
};

bool cmp(Node a, Node b) {
        return a.x < b.x;
}

vl yAxis;
vector<Node>yLine;

// Segment tree functions
ll tree[4*100010], prop[4*100010];
ll calculate(int node, int l, int r) {
        if(prop[node] > 0)
                return yAxis[r]-yAxis[l];
        else
                return tree[node<<1] + tree[node<<1|1];
}

void update(ll node, ll l, ll r, ll yMin, ll yMax, ll val) {
        if(yMax < yAxis[l] || yAxis[r] < yMin)
                return;

        if(yMin <= yAxis[l] && yAxis[r] <= yMax) {
                prop[node] += val;
                tree[node] = calculate(node, l, r);
                return;
        }

        if(l+1 == r) return;        // The leaf node must be double node as this is a Cartesian Graph
        ll mid = (l+r)>>1;
        update(node<<1, l, mid, yMin, yMax, val);
        update(node<<1|1, mid, r, yMin, yMax, val);
        tree[node] = calculate(node, l, r);
}

int main() {
        int t, x1, y1, x2, y2, n;
        scanf("%d", &t);

        for(int Case = 1; Case <= t; ++Case) {
                scanf("%d", &n);
                yAxis.pb(0);
                for(int i = 0; i < n; ++i) {
                        // lower-left start point, upper-right end point
                        scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
                        yAxis.pb(y1);
                        yAxis.pb(y2);
                        yLine.pb(Node(y1, y2, x1, 1));
                        yLine.pb(Node(y1, y2, x2, -1));
                }

                // Taking only unique y values (will be used as segment tree nodes)
                sort(yAxis.begin(), yAxis.end());
                yAxis.erase(unique(yAxis.begin()+1, yAxis.end()), yAxis.end());

                // Sorting y axis lines according to x axis (left to right)
                sort(yLine.begin(), yLine.end(), cmp);
```

```
            memset(tree, 0, sizeof tree);
            memset(prop, 0, sizeof prop);

            update(1, 1, (int)yAxis.size()-1, yLine[0].yMin, yLine[0].yMax, yLine[0].val);
            ll area = 0;
            for(int i = 1; i < (int)yLine.size(); ++i) {
                area += tree[1] * (yLine[i].x - yLine[i-1].x);
                update(1, 1, (int)yAxis.size()-1, yLine[i].yMin, yLine[i].yMax, yLine[i].val);
            }

            printf("Case %d: %lld\n", Case, area);
            yAxis.clear();
            yLine.clear();
        }
        return 0;
}

// LightOJ – Lining Up Students
// Number delete operation
// Every Pos contains 1 by default, parent nodes are sum of child node

int SearchVal(int pos, int L, int R, int val) {                    // Searches for val'th value(returns the value)
    if(L == R) {                                                   // and sets the val'th value to zero at the same time
        tree[pos] = 0;                  // By modifying this line, the set-to-zero can be ignored (only query func)
        return L;
    }

    int mid = (L+R)>>1;
    if(val <= tree[pos<<1]) {
        int idx = SearchVal(pos<<1, L, mid, val);
        tree[pos] = tree[pos<<1] + tree[pos<<1|1];
        return idx;
    }
    else {
        int idx = SearchVal(pos<<1|1, mid+1, R, val-tree[pos<<1]);
        tree[pos] = tree[pos<<1] + tree[pos<<1|1];
        return idx;
    }
}
```