```cpp
/*

Max Subarray segment tree

*/


#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef struct node
{
    ll suf, pref;
    ll best, sum;
    node()
    {
        sum = 0;
        best = pref = suf = -1e9;
    }
    node(ll val)
    {
        suf = pref = best = sum = val;
    }
} Node;
int const N = 1e5 + 5;
int n;
ll a[N];
Node seg[4 * N];
Node operator+(Node lf, Node rt)
{
    Node ret;
    ret.best = max({lf.best, rt.best, lf.suf +
rt.pref});
    ret.pref = max(lf.pref, lf.sum + rt.pref);
    ret.suf = max(rt.suf, rt.sum + lf.suf);
    ret.sum = lf.sum + rt.sum;
    return ret;
}
```

```cpp
void build(int si = 0, int ss = 0, int se = n - 1)
{
    //cout << si << " " << ss << " " << se <<
"\n";
    if (se == ss)
    {
        seg[si] = node(a[ss]);
        return;
    }
    int md = (se + ss) / 2;
    build(si * 2 + 1, ss, md),
        build(si * 2 + 2, md + 1, se);

    seg[si] = seg[si * 2 + 1] + seg[si * 2 + 2];
    //cout << "Trail " << si << " " <<
seg[si].best << "\n";
}
Node query(int l, int r, int si = 0, int ss = 0,
int se = n - 1)
{
    if (l > se || r < ss)
        return node();
    if (ss >= l && se <= r)
        return seg[si];
    int md = (se + ss) / 2;
    return query(l, r, si * 2 + 1, ss, md) +
query(l, r, si * 2 + 2, md + 1, se);
}
int main()
{
    ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int q, l, r;
    cin >> q;
    build();
    while (q--)
    {
        cin >> l >> r;
        --l, --r;
        Node ans = query(l, r);
        cout << ans.best << "\n";
    }
}
```

```
/*

Lazy propagation with bit wise

*/

#include <bits/stdc++.h>
using namespace std;
#define ll long long
int const N = 2e5 + 5;
int n;
ll seg[4 * N], lazy[4 * N];
int a[N];
void build(int node = 1, int s = 0, int e = n - 1)
{
    if (s == e)
    {
        seg[node] = a[s];
        return;
    }
    int md = s + e >> 1;
    build(node << 1, s, md),
        build(node << 1 | 1, md + 1, e);
    seg[node] = min(seg[node << 1], seg[node << 1
| 1]);
}
inline void propagate(int node, int s, int e)
{
    // propagate min or max
    seg[node] += lazy[node];
    if (s != e)
    {
        lazy[node << 1] += lazy[node];
        lazy[node << 1 | 1] += lazy[node];
    }
    lazy[node] = 0;
}
```

```
ll query(int l, int r, int node = 1, int s = 0,
int e = n - 1)
{
    propagate(node, s, e);
    if (s > r || e < l)
        return 1e15;
    if (s >= l && e <= r)
    {
        return seg[node];
    }
    int md = s + e >> 1;
    return min(query(l, r, node << 1, s, md),
query(l, r, node << 1 | 1, md + 1, e));
}

void updateRange(int l, int r, ll val, int node =
1, int s = 0, int e = n - 1)
{
    propagate(node, s, e);
    if (s > r || e < l)
        return;
    if (s >= l && e <= r)
    {
        lazy[node] += val;
        propagate(node, s, e);
        return;
    }
    int md = s + e >> 1;
    updateRange(l, r, val, node << 1, s, md),
        updateRange(l, r, val, node << 1 | 1, md +
1, e);
    seg[node] = min(seg[node << 1], seg[node << 1
| 1]);
}
int main()
{

ios_base::sync_with_stdio(0),cin.tie(0),cout.tie(0
);
    cin >> n;
    for (int i = 0; i < n; i++)
```

```
        cin >> a[i];
    int q;
    cin >> q;
    build();
    while (q--)
    {
        int l, r, val;
        cin >> l >> r;
        if (cin.peek() != '\n')
        {
            cin >> val;
            if (l <= r)
                updateRange(l, r, val);
            else
            {
                updateRange(l, n - 1, val);
                updateRange(0, r, val);
            }
        }
        else
        {
            if (l <= r)
                cout << query(l, r) << "\n";
            else
            {
               // cout << seg[3] << " " << seg[1]
<< " \n";
                cout << min(query(l, n - 1) ,
query(0, r)) << "\n";
            }
        }
    }
}
```

```
/*

Merge sort tree

*/

#include <bits/stdc++.h>
using namespace std;
#define ll long long
int const N = 2e5 + 5;
int n;
vector<int> seg[4 * N];
int a[N];
void build(int node = 1, int s = 0, int e = n - 1)
{
    if (s == e)
    {
        seg[node].push_back(a[s]);
        return;
    }
    int md = s + e >> 1;
    build(node << 1, s, md),
        build(node << 1 | 1, md + 1, e);
    for (auto &it : seg[node << 1])
    {
        seg[node].push_back(it);
    }
    for (auto &it : seg[node << 1 | 1])
    {
        seg[node].push_back(it);
    }
    sort(seg[node].begin(), seg[node].end());
}
```

```
ll query(int l, int r, int k, int node = 1, int s
= 0, int e = n - 1)
{
    if (s > r || e < l)
        return 0;
    if (s >= l && e <= r)
    {
        return lower_bound(seg[node].begin(),
seg[node].end(), k) - seg[node].begin();
    }
    int md = s + e >> 1;
    return query(l, r, k, node << 1, s, md) +
query(l, r, k, node << 1 | 1, md + 1, e);
}
int main()
{
    ios_base::sync_with_stdio(0), cin.tie(0),
cout.tie(0);
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int q;
    cin >> q;
    build();
    while (q--)
    {
        int l, r, k;
        cin >> l >> r >> k;
        cout << query(l - 1, r - 1, k) << "\n";
    }
}
```

```
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

// A utility function to get the
// middle index from corner indexes
int getMid(int s, int e)
{
    return (s + (e - s) / 2);
}

// A recursive function to get the gcd of values
in given range
// of the array. The following are parameters for
this function

// st --> Pointer to segment tree
// si --> Index of current node in the segment
tree. Initially
// 0 is passed as root is always at index 0
// ss & se --> Starting and ending indexes of the
segment represented
// by current node, i.e., st[si]
// qs & qe --> Starting and ending indexes of
query range
```

```
int getGcdUtil(int* st, int ss, int se, int qs,
int qe, int si)
{
    // If segment of this node is a part of given
range
    // then return the gcd of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the
given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the
given range
    int mid = getMid(ss, se);
    return __gcd(getGcdUtil(st, ss, mid, qs, qe, 2
* si + 1),
                 getGcdUtil(st, mid + 1, se, qs,
qe, 2 * si + 2));
}

// A recursive function to update the nodes which
have the given
// index in their range. The following are
parameters
// st, si, ss and se are same as getSumUtil()
// i --> index of the element to be updated. This
index is
// in the input array.
// diff --> Value to be added to all nodes which
have i in range
void updateValueUtil(int* st, int ss, int se, int
i, int diff, int si)
{
    // Base Case: If the input index lies outside
the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this
node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss) {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2 *
si + 1);
        updateValueUtil(st, mid + 1, se, i, diff,
2 * si + 2);
    }
}

// The function to update a value in input array
and segment tree.
// It uses updateValueUtil() to update the value
in segment tree
void updateValue(int arr[], int* st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        cout << "Invalid Input";
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n - 1, i, diff, 0);
}
```

```
// Function to return the sum of elements in range
// from index qs (query start) to qe (query end)
// It mainly uses getSumUtil()
int getGcd(int* st, int n, int qs, int qe)
{

    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        cout << "Invalid Input";
        return -1;
    }

    return getGcdUtil(st, 0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment
Tree for array[ss..se].
// si is index of current node in segment tree st
int constructGcdUtil(int arr[], int ss, int se,
int* st, int si)
{
    // If there is one element in array, store it
in current node of
    // segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one element then
recur for left and
    // right subtrees and store the sum of values
in this node
    int mid = getMid(ss, se);
    st[si] = __gcd(constructGcdUtil(arr, ss, mid,
st, si * 2 + 1),
                constructGcdUtil(arr, mid + 1, se,
st, si * 2 + 2));
    return st[si];
}
```

```
// Function to construct segment tree from given
array. This function
// allocates memory for segment tree and calls
constructSTUtil() to
// fill the allocated memory
int* constructGcd(int arr[], int n)
{
    // Allocate memory for the segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* st = new int[max_size];

    // Fill the allocated memory st
    constructGcdUtil(arr, 0, n - 1, st, 0);

    // Return the constructed segment tree
    return st;
}
```

```
// Driver code
int main()
{
    int arr[] = { 1, 3, 6, 9, 9, 11 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree from given array
    int* st = constructGcd(arr, n);

    // Print GCD of values in array from index 1
to 3
    cout << getGcd(st, n, 1, 3) << endl;

    // Update: set arr[1] = 10 and update
corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find GCD after the value is updated
    cout << getGcd(st, n, 1, 3) << endl;

    return 0;
}
```