```cpp
//
//   Consider a string of length N consisting of 0 and 1.
//   The score for the string is calculated as follows:
//     - For each i (1 ⩽ i ⩽ M), a is added to the score
//       if the string contains 1 at least once
//       between the l-th and r-th characters (inclusive).
//   Find the maximum possible score of a string.
//
//      Time Complexity: O(N log N)
//

#include <bits/stdc++.h>
#define ll long long

using namespace std;

inline void uadd(ll& a, ll b) { a += b; }
const ll NEG_INF = LLONG_MIN / 10;

struct LazySegmentTree {
    int N = -1;

    struct Node {
        ll value = NEG_INF; // default
        ll lazy = 0;
        Node operator+(const Node &other) {
            return {
                max(this->value, other.value)
            };
        }
    };

    vector<Node> seg;
    LazySegmentTree(int N) {
        this->N = N;
        int sz = 1 << 21;
        seg.resize(sz);
    }

    void push(int ind, int l, int r) {
        if (seg[ind].lazy == 0) return;
        if (seg[ind].value == NEG_INF) seg[ind].value = seg[ind].lazy;
        else uadd(seg[ind].value, seg[ind].lazy);
        if (r - l != 1) {
            uadd(seg[2*ind+1].lazy, seg[ind].lazy);
            uadd(seg[2*ind+2].lazy, seg[ind].lazy);
        }
        seg[ind].lazy = 0;
    }

    void updateRange(int b, int e, ll x, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        push(ind, l, r);
        if (e⩽l || b⩾r) { return; }
        if (b⩽l && r⩽e) {
            uadd(seg[ind].lazy, (ll)x);
            push(ind, l, r);
            return;
        }

        int m = (l + r) / 2;
        updateRange(b, e, x, 2 * ind + 1, l, m);
        updateRange(b, e, x, 2 * ind + 2, m, r);
        seg[ind] = seg[2*ind+1] + seg[2*ind+2];
    }

    // result in [b,e) of node that covers [l,r)
    Node askLR(int b, int e, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        push(ind, l, r);
        if (l⩾e || r⩽b) return {  }; // empty
        if (l⩾b && r⩽e) return seg[ind];
        int m = (l + r) / 2;
        return askLR(b, e, ind * 2 + 1, l, m) + askLR(b, e, ind * 2 + 2, m, r);
    }
};

struct query {
    ll l, r, v;
};
```

```cpp

int main() {
    int n, m;
    cin >> n >> m;

    vector<query> q(m);
    vector<vector<int>> events(n + 5);
    for (int i = 0; i < m; i++) {
        int l, r, v;
        cin >> l >> r >> v;
        q[i] = { l, r, v };
        events[l].push_back(i);
        events[r].push_back(~i);
    }

    // dp[i] - the best score of the prefix to index i if s[i] == 1
    // v[q] - value of the q-th query
    // dp[i] = max(dp[j] + sum(v[q])) for all j < i ,
    //         for all q if l[q] ⩽ i ⩽ r[q] && !(l[q] ⩽ j ⩽ r[q]))

    // when you meet a left edge  → tree.add( [ 0, l[q] - 1 ],  v[q] )
    // when you meet a right edge → tree.add( [ 0, l[q] - 1 ], -v[q] )

    // explanation:
    //  when you meed a left edge, this interval gets activated, so
    //  for every dp which is up to this point, we want to add this interval's value.
    //  when this interval dies out, we remove it from the dp tree, because other dps
    //  to the right will have this edge included if it was worth it.

    LazySegmentTree tree(n + 10);
    for (int i = 1; i ⩽ n; i++) {
        for (auto& event: events[i]) {
            if (event ⩾ 0) {
                tree.updateRange(0, q[event].l, q[event].v);
            }
        }

        ll new_dp = tree.askLR(0, i).value;
        tree.updateRange(i, i+1, new_dp);

        for (auto& event: events[i]) {
            if (event < 0) {
                tree.updateRange(0, q[~event].l, -q[~event].v);
            }
        }
    }

    cout << tree.askLR(0, n+1).value << endl;
    return 0;
}
```