# UVa 11235: Frequent values

*July 24, 2015*

Tags: uva segment-tree

---

UVa 11235: Frequent values simplified problem statement: You are given an array of integers and a number of queries. Each query is in the form of two positive integers i and j, and your program must print the number of occurances of the most frequently occuring value in the array between indicies i and j, inclusive. This problem really stretched my problem solving abilities. I knew from the start that the problem could be solved in time using a segment tree because the problem was listed as a segment tree practice problem in CP3. However, beyond that, I was stuck. This was clearly different from an RMQ or RSQ, so what data was I supposed to store?

Eventually, I figured out that because the array given in each test case is sorted in non-decreasing order, the frequency of a number within a given range can be calculated in constant time, assuming you already have the range in which the number appears in the array. This meant that it wasn't necessary to store the frequencies of the array elements in the segment tree itself, because I could calculate the frequencies whenever I wanted with no significant time cost. Thus, I kept an STL map of integers to the ranges in which they appeared in the array, and each node in the segment tree stored an array element. While writing this solution, I made a couple of mistakes I had to spent precious time debugging, including not noticing that the problem specified 1-based indexing, and counting the frequencies within the wrong ranges in my search method.

It was after debugging my solution on the problem input that I realized there was a flaw in my reasoning. My solution failed for inputs with a certain property, which I will illustrate with an example. Take the input array [2, 2, 2, 3, 3, 3, 3, 4, 4, 4] and the query (1, 10) (i.e. the entire array). By inspection, it is clear the answer to this query should be 4 (because the number 3 appears 4 times), but my program returned 3. This is because the interval of the array containing the 3s was split in half between the left and right side of the segment tree. In other words, if the root of the tree is at height 0, the left node at height 1 (responsible for the range [2, 2, 2, 3, 3]) correctly held the value 2, and the right side (responsible for the range [3, 3, 4, 4, 4]) correctly held the value 4, which meant that in the calculation of the value for the root node, 3 was not even considered. To work around this corner case, I added some logic that detected ranges of the same integer that were "split" around left and right child nodes, and took this additional number into consideration if found.

At this point I submitted and got TL. I realized that the STL map I was using to look up rangesdid lookups in O(logn) (where n is the size of the map), which was too slow. The lookup needed to be in constant time. Luckily, the numbers in the array were guaranteed to be between -10^6 and 10^6, which was a small enough range to store in an array. After switching the map to a vector, my solution passed.

```
#include <iostream>
#include <vector>
using namespace std;

#define n_unique 200005
class SegmentTree {
private:
    int n;
    // A is the input array, st is the segment tree
    vector<int> A, st;
    // this vector maps array entries to the range
    // of indexes they appear
    vector< pair<int, int> > freqs;

    // convenience methods to get the left and right child
    // of a node
    int left(int i) { return i<<1; }
    int right(int i) { return (i<<1)+1; }

    int build(int L, int R, int idx) {
        if(L == R) st[idx] = A[L];
        else {
            int left_val = build(L, (L+R)/2, left(idx));
```

```
            int right_val = build(((L+R)/2)+1, R, right(idx));
            int left_freq = getFreq(left_val, L, R);
            int right_freq = getFreq(right_val, L, R);
            // detect "split" ranges and take them into account
            if(A[(L+R)/2] == A[((L+R)/2)+1]) {
                int split_freq = getFreq(A[(L+R)/2], L, R);
                if(split_freq > left_freq)
                    st[idx] = (split_freq > right_freq) ? A[(L+R)/2] : right_val;
                else
                    st[idx] = (left_freq > right_freq) ? left_val : right_val;
            } else {
                st[idx] = (left_freq > right_freq) ? left_val : right_val;
            }
        }
        return st[idx];
    }

    int rfq(int i, int j, int L, int R, int idx) {
        if(R < i || j < L) return n_unique;
        if(i <= L && R <= j) return st[idx];

        int left_val = rfq(i, j, L, (L+R)/2, left(idx));
        int right_val = rfq(i, j, ((L+R)/2)+1, R, right(idx));
        int lo = i, hi = j, left_freq, right_freq;
        if(i < L) lo = L;
        if(R < j) hi = R;
        if(left_val != n_unique) left_freq = getFreq(left_val, lo, hi);
        if(right_val != n_unique) right_freq = getFreq(right_val, lo, hi);
        // detect "split" ranges and take them into account
        if(A[(L+R)/2] == A[((L+R)/2)+1]) {
            int split_freq = getFreq(A[(L+R)/2], lo, hi);
            if(left_val == n_unique) return (split_freq > right_freq) ? A[(L+R)/2] : right_val;
            if(right_val == n_unique) return (split_freq > left_freq) ? A[(L+R)/2] : left_val;
            if(split_freq > left_freq)
                return (split_freq > right_freq) ? A[(L+R)/2] : right_val;
            else
                return (left_freq > right_freq) ? left_val : right_val;
        } else {
            if(left_val == n_unique) return right_val;
            if(right_val == n_unique) return left_val;
            return (left_freq > right_freq) ? left_val : right_val;
        }
    }

public:
    SegmentTree(vector<int> &_A) {
        A = _A;
        freqs.assign(n_unique, make_pair(-1, -1));
        int last = n_unique;
        for(int i=0; i<(int)A.size(); i++) {
            if(A[i] != last) {
                if(i > 0) freqs[last+100000].second = i-1;
                freqs[A[i]+100000].first = i;
            }
            last = A[i];
        }
        freqs[last+100000].second = ((int)A.size())-1;
        n = A.size();
        st.assign(4*n, 0);
        build(0, n-1, 1);
    }

    int rfq(int i, int j) {
        return rfq(i, j, 0, n-1, 1);
    }

    int getFreq(int n, int i, int j) {
        int lo = freqs[n+100000].first, hi = freqs[n+100000].second;
```

```cpp
        if(lo <= i && j <= hi) return j-i+1;
        else if(i < lo && j <= hi) return j-lo+1;
        else if(lo <= i && hi < j) return hi-i+1;
        else return hi-lo+1;
    }
};
int n, q, i, j;
vector<int> a;
int main() {
    while(cin >> n, n) {
        cin >> q;
        a.resize(n);
        for(auto &x : a) cin >> x;
        SegmentTree st(a);
        for(int qq=0; qq<q; qq++) {
            cin >> i >> j; i--; j--;
            int number = st.rfq(i, j);
            int freq = st.getFreq(number, i, j);
            cout << freq << '\n';
        }
    }

    return 0;
}
```

11235.cpp hosted with ♡ by GitHub view raw

*For technical reasons, comments are temporarily unavailable on my posts. If you'd like to discuss this article with me, feel free to email me at david@davidudelson.com or hit me up on Twitter.*

Read More Posts Top