GSS2 Solving Problems on SPOJ Sphere Online Judge (SPOJ)

How can the SPOJ problem GSS2 be solved?

Answer Follow · 41 Request 1

1 Answer

Brian Bi, 371 solved problems on SPOJ
Answered Jun 6, 2014

First, let's make clear what the problem is actually asking.

You are given a sequence of up to 100000 integers in the range [-100000, 100000]. You are also given up to 100000 queries. Each query names a nonempty slice of the sequence. For each query, return the maximum sum of a (possibly empty) slice of the given slice, where duplicate elements are ignored in the sum.

[Note: by "slice" we mean a contiguous subsequence. Slices of strings are substrings.]

[Note 2: Each query has a nonnegative answer since the sum of the empty slice is zero.]

The observation that kick-starts a solution to GSS2 is the following: **a slice of a sequence is a prefix of a suffix.** Now let's say we incrementally build up the sequence, one element at a time, starting from the first and working our way to the last. Then,

- Every time we append an element to our sequence, one new suffix is generated (consisting of the element we just appended).
- In addition, every previously existing suffix has the new element appended to it.
- Therefore, the sum of each suffix increases by the value of the new element. But if the suffix begins at or before the previous occurrence (if any) of the new element, then the new element contributes nothing at all to the total sum of the suffix ignoring duplicates. Therefore, if we identify the previous occurrence of the new element, only suffixes starting to the right are "updated".
- **Every slice was at one point an *entire* suffix** (*i.e.*, suppose a slice spans indices i through j, inclusive; then, immediately after element j was appended, this slice was the *entire* suffix starting from position i.) For this reason, the maximum sum of a slice starting at position x and ending at or before position y is the maximum sum that a suffix starting at position x has *ever* had so far (*i.e.*, assuming no elements after position y have been added.)

This suggests a query-sorting approach.

- Sort the queries by their upper bounds (*i.e.*, if each query is denoted [x, y], sort by y-values.)

You and Rajat De upvoted this

- After adding an element to the sequence, answer all queries whose upper bound equals the index of the element just added. In the sample case [4 -2 -2 3 -1 -4 2 2 -6], this means for example that the query [1, 5] should be answered immediately after the value -1 is added. The answer to the query [x, y] is the *greatest value that any suffix starting at position x or later has* ***ever*** *had, to date*.

In order to efficiently implement this approach we need a data structure that stores an array A and supports two kinds of operations:

1. Given x, y, z, add z to each element in the slice A[x..y]. (This handles the second bullet point above.)
2. ven y, return the maximum value that has eve A[x..y]. (This handles the third bullet point above.)

So all that remains is to design and implement such a data structure. This turns out to be the crux of the problem. Because we need to perform both queries and updates on **ranges** of the array A rather than individual elements, the necessary data structure is a segment tree with lazy propagation. I'll explain my first attempt to create a segtree that supports the two operations above, and then I'll explain how it failed and how I fixed it.

In the first attempt, I stored three fields in each node:

- **cur**, the current maximum of all elements stored in the slice the node is responsible for.
- **best**, the greatest maximum to date of all elements stored in the slice the node is responsible for.
- **lazy**, the sum of all range updates affecting this node that have not yet been propagated onto the node's children. (A lazy value at a leaf is simply ignored.)

Hopefully it's obvious how to update the cur and lazy values. (If not, review the lazy propagation technique for segment trees.) The best value is updated every time the cur value for a node changes.

This works for the sample data and a few small test cases I tried, but upon submission gave Wrong Answer. I was only able to discover the bug once I compared my solution automatically against an accepted solution I found on the internet, using randomly generated test data.

The problem occurs when multiple consecutive lazy updates occur to a node without intervening propagation to children. An example test case is [1 2 -1 -1]. Our segment tree has a height of 2. The root node is responsible for the entire array, the left child of the root node for the first half of the array, and so on. When we add the element 2, the root node's left child receives a lazy update of +2. When we add the next element, -1, the root node's left child receives a lazy update of -1. (-1 is also added to the left child of the right child of the root.) Now let's consider the suffix starting from the second element. After adding the

You and Rajat De upvoted this

discharge the root's left child, which pushes the current lazy value of +1 onto the children. As far as A[2] (1-based indexing) is aware, its value never was 2; first it was 0, and then it became 1 after its parent was discharged. And so we get the wrong answer for the query [2, 3]: we get 1 when the answer is 2.

To fix this, we need to add *another* field to each node in the segtree: a **bestlazy** value, which remembers the greatest value that the lazy field has held *since the last discharge*. Now, how do we perform a discharge taking into account the bestlazy value? The way I think about it is as follows: consider some leaf node of the tree, responsible for a single element of A. Now consider the queued updates we encounter as we walk up the tree. Updates queued at a lower node necessarily occurred earlier than updates queued at a higher node. (This is because when we update lower nodes, we have to discharge their parents first.) This implies that, in a sense, as we walk up the tree, we encounter queued updates in the order in which they originally occurred. The best value for the element at some intermediate stage during the application of those updates should be the best value that could be obtained by cutting off the queued updates at the ancestors at any point:

Upvote · 128 Downvote

- adding the bestlazy value from the parent to the current value
- adding the entire lazy value from the parent, and the bestlazy value from the grandparent to the current value
- adding the entire lazy values from the parent and grandparent, and the bestlazy value from the great-grandparent to the current value ...

To implement this as we discharge from top to bottom, we replace the bestlazy value at a child by the maximum of its current value, and the entire lazy value for the child plus the bestlazy value for the current node. It is not too hard to see that this accomplishes the above, as it maintains the invariant that *when a node's ancestors have no queued updates, the bestlazy value is the greatest prefix sum of the updates queued at this node*. (*i.e.*, because it takes into account prefixes that include the entire series of updates queued at the child, plus a prefix of those queued at the parent---and the maximum prefix of the updates queued at the parent is precisely the parent's bestlazy value.) So our full discharge procedure is as follows:

```
void discharge(int node) {
    for (int child = 2*node; child <= 2*node+1; child++) {
        best_lazy[child] = max(best_lazy[child],
                               lazy[child] + best_lazy[node]);
        lazy[child] += lazy[node];
        best[child] = max(best[child],
                          cur[child] + best_lazy[node]);
        cur[child] += lazy[node];
    }
    lazy[node] = best_lazy[node] = 0ll;
}
```

Using this I finally got AC.

8.4k Views · View Upvoters

You and Rajat De upvoted this

1 Answer Collapsed (Why?)

Top Stories from Your Feed

You and Rajat De upvoted this