Today I want to discuss lazy update based data structures, via the following problem: support a sequence of numbers $A[1 \ldots n]$ subject to:

2. Max subinterval sum query: query for the max sum of an interval, or an interval contained in $[\hat{l}, \hat{r}]$ ($\hat{l} \leq l \leq r \leq \hat{r}$).

The simple case is the modifications all happen on length 1 intervals. For this, we can do the standard balanced binary tree construction. Here there are three cases:

1. The max is completely contained in $p.left$, or $p.right$.
2. The max crosses the middle, in which case it is $p.left.maxsuffix + p.right.maxprefix$.

Note that max prefix / suffix sum follow similar recurrences, but with the sum of the other portion.

## 2 Two Level Scheme

For the more general case, let’s first try to get something sublinear time. For that, consider partitioning into blocks of length $\sqrt{n}$.

Each update only partially intersect two intervals, and we can afford to recompute both completely. The other intervals are affected by global additions, so we have a global update flag on it. For these globally updated ones, there are three cases to consider:

1. an interval completely contained in a block.
2. an interval starting and ending in different blocks. Here we still should have the max suffix sum of the starting block, and the max prefix sum of the last block.

So up to a factor of $O(\sqrt{n})$, we need to maintain a data structure that, for a shift value $x$, outputs the max prefix/suffix/contained sum of a block. Here the issue is that a
length $i$ sequence gets updated by $x \cdot i$. So if we let $s[i]$ be the max sum of a subsequence with length $i$, we are looking for the maximum of

$$x \cdot i + s[i]$$

Those familiar with convex hulls will recognize this as an extreme point query, but we can do something a little more general here. Note that for two different values $(i_1, s[i_1])$ and $(i_2, s[i_2])$, the region that

$$x \cdot i_1 + s[i_1] > x \cdot i_2 + s[i_2]$$

is an interval as well. That is, if we consider the range of $x$ where some $(i, s[i])$ is optimal, it’s just interval. So for $\sqrt{n}$ values (one per prefix sum), we can build these ranges in $O(n \log n)$ time using some kind of divide-and-conquer.

Note that this representation complexity also holds for the contained problem: there are only at most $O(\sqrt{n})$ different lengths of contained intervals. So we can do some divide-and-conquer to construct this list for growing lists. For an interval $[l, r]$ (corresponding to a block), we can build the list of values crossing the middle, by looking up the different opt values (based on $x$) for $[l, \text{mid}]$ and $[\text{mid} + 1, r]$. As both lists have complexity $O(n)$, the overall list also has complexity $O(n)$. Merging these three lists also takes $O(n)$ time.

As we only touch two intervals, the complexity of these rebuilds is $O(\sqrt{n} \log n)$. Query complexity is also the same, since we need to check all $\sqrt{n}$ segments, and piece together their prefixes/suffixes.

3 Multi-Level Scheme

Now consider a multi-level scheme, built once again on top of a fully balanced static binary tree.

Whenever we do a comparison, of say $i_1 \cdot x + s[i_1]$ versus $i_2 \cdot x + s[i_2]$ we can find the smallest $x$ that needs to be added to this range until the comparison goes the other way.

Whenever that happens, we locate this, find the event, and ‘flip’ it. This in turn creates more events, which we also go update. That is, at each node, we track the minimum increase in $x$ needed for some max below to ‘flip’ in the opposite direction.

That is, after we finish updating a node, on the upward propagation, we also track

$$p.minx = \min \{p.l.minx, p.r.minx, \text{any of the } x \text{ needed for one of } p \text{’s events} \}$$

and propagate this value ‘up’ after we updated a subtree of $p$.

For simplicity, consider what happens if we just maintain the max prefix value ($maxpre$), as the maximum total value follows analogously.

We also consider the case of a three level bucketing scheme, for some

$$n = B^3$$

and deal with the case where the update point is always a multiple of $B$. That is, we have top and middle level buckets, of size $B^2$ and $B$ respectively. Each bucket stores:
1. a list of the max prefix sum of the $B$ buckets below,

2. how much global increment (to everything in the bucket) is needed until the overall max prefix sum of the bucket changes,

3. the min global increment of any of the buckets contained below it, propagated upward recursively.

Then when we increment a bucket globally by $\delta$, if $\delta$ is more than the threshold for a change, we update the max prefix sum, and inform the bucket above if it’s a middle level bucket.

When incrementing a top-level bucket, we also look for any middle bucket contained in it whose increment threshold is less, and go update those.

The cost of this three-layer structure is then constrained by how much it could change, which happen in one of the following ways:

1. A bucket changed due to a partial modify (to a prefix or suffix): this happens to $O(1)$ buckets per step, so we can afford to completely rebuild the min, at a cost of $O(B)$.

2. The max prefix of one of the middle level bucket changes, due adding to $\delta$ to everything in it. This can happen at most $O(B)$ times total per middle level bucket, for a total of $O(n)$ changes.

3. The maximizer of a top bucket changes, even though its list remains unchanged: this can happen at most $O(B)$ times per top-level bucket due to the list having size $B$ (one per middle level bucket it contains). So its a lower order term compared to the list changes.

4. The list of middle bucket maximizers of some top bucket changes: this is actually an insertion, as there is no need to delete the previous maximizer (which is always worse). So this works out to $O(n)$ total as well.

4 Amortized Analysis

More generally, if we do a binary tree like structure with $O(\log n)$ levels, note that only the structure of $O(\log n)$ ranges can change, each corresponding to the level $i$ bucket (of length $2^{d-i}$) that partially intersect the updated range. Each maximizer change can also propagate up $O(\log n)$ levels (their tree paths can be separate due to, say, an update on $[33, 64]$ triggering a new maximizer in $[37, 40]$.) So the cost of a $d$ level structure is $n^{1/d}d^2$, which comes out to $O(\log^2 n)$.

When an analysis of a $d$ level structure works, there is often a way to write down an invariant based proof that’s more formal that the casework based analysis above.
Here it’s possible to define a potential function for each comparison of the form of

\[ p.\text{maxpre} = \max\{p.\text{left.maxpre}, p.\text{left.sum} + p.\text{right.maxpre}\} \]

to be the number of values, contained in left and right, with longer interval lengths than the current maximizer.

For simplicity, let’s analyze the version of this for just MaxPrefix. Here we can write the global potential function as:

\[ \Phi = O(\log n) \cdot \sum_{\text{tree nodes } p} \text{number of prefixes that could supercede it after a global increase} \]

(3)

Note that this pays for all the changes, and associated propagations, ahead of time. So we need to make sure that range increments don’t increase this cost by too much. Now consider how a tree node’s interval interact with an update interval:

1. If the incremented range is a proper suffix (this includes incrementing the entire interval), then the longer ones won’t do worse. So here the potential can only drop.

2. If the incremented range is a proper prefix, then the number of potentially superceding points can only go up by 1: note that the longer intervals still gain at least as much as the shorter intervals: just not more than the prefix length. So the only one item that could be added is one at the end of the prefix, that was previously ‘shadowed’ by a longer interval outside of the prefix. For example, in the sequence

\[ 1, -10, -5 \]

(4)

the length two prefix can never be a max sum one, but upon adding 9 to the first two elements we get

\[ 10, -1, -5, \]

(5)

where length two prefix might be useful when 2 is added to everyone globally. This can be proven to be the only case where the potential goes up, so as \(O(\log n)\) regions have prefix increments, the total update cost also amortizes to \(O(\log^2 n)\).