January Contest Review ($\sqrt{N}$ decomposition)

Justin Zhang and Daniel Wisdom

9 February 2018

1 Lifeguards

1.1 Problem

Given N ($1 \leq N \leq 10^5$) intervals with endpoints in the range 0...10^9, and an integer K ($1 \leq K \leq N$), find the maximal area covered by the union of the segments after any K are removed.

Example:

```
3 2

1 8
7 15
2 14
```

Output: 12

1.2 Solution

Dynamic programming!

Let's first determine segments we can always remove to make the problem simpler.

Notice that if an segment is completely contained within another segment, we can remove it. This follows since there's no situation where we should keep the smaller one and not the large one. If we remove the larger one, we had to have already removed the smaller one. If we keep the larger one, the smaller one is useless.

Once we do this, notice that we can sort all segments so that they're monotonically increasing: if a segment starts before another segment, it also ends sooner.

Now we can do dynamic programming. Namely, we can do DP over a function $f(n, k)$ that gives us the maximum overlap area by all segments $1...n$ after we've optimally removed at least k of the prior $n - 1$ segments (we *must* include n).

We have three cases to consider now to find $f(n, k)$.

1. The segment n is the first one we're including
2. We remove all segments before n that overlap with n
3. We keep only the leftmost segment (q) that overlaps with n and remove all segments $q+1...n-1$

Let's consider why these are the only cases we have to consider for $f(n, k)$. The first case clearly must be considered, if applicable ($n - 1 \leq k$).

Regarding the third: if we keep q, note that all segments $q+1...n-1$ are fully contained by q and n. Thus, for the same reason we can remove segments that are completely contained within other segments, we can ignore $q+1...n-1$.

Now regarding the second: this case is the same as the second, but simply doesn't contain q. Since q is the last segment whose inclusion depends on n, we don't have to consider any segments before q. In other words, segments $1...q-1$ do not depend on whether we include n or not, so we can use previous results considered up to this point as-is.

Now how can we actually implement these three cases?

The first one is trivial: $dp[n][k] =$ the length of segment n. We can only do this case if $n-1 \leq k$, since this case removes all segments before n.

Regarding the second and third, we need to keep another pointer to segment q. Now, every time we increase n, we keep increasing q until we get to the first segment that overlaps with n (we know we don't have to decrease q to get the first segment that overlaps because the segments are monotonically increasing). This is the method of *two pointers*.

Now, in case two, the answer is $dp[n][k] = dp[q-1][k-(n-q)] +$ `area covered by n` and case three is $dp[n][k] = dp[q][k-(n-q-1)] +$ `area covered by the union of n and q`.

We set the final value of $dp[n][k]$ to the maximum of these three cases.

2 $\sqrt{N}$ Decomposition

When solving a problem it is sometimes helpful to split the data up into buckets, and then process the buckets separately. This helps when processing buckets of $\sqrt{N}$ elements is somehow easier than doing all N in one step.

For example, let's say we want to do two things to an array: change a range of the array by some amount and find the maximum element in a range. If we divide the array into $\sqrt{N}$ buckets, each with $\sqrt{N}$ elements, we can keep a lazy update counter for each bucket. This way, if we need to increase an entire bucket by k, we can just increase the lazy counter by k in $O(1)$ time. We can also store the maximum element in every bucket. When we update a whole bucket, we increment this maximum. To update part of a bucket, we can just loop over the whole bucket and update the individual values in the range. We can also recalculate the new max. This will take $O(\sqrt{N})$ time.

Putting this together we can update the structure for any range in $O(\sqrt{N})$ time. For the buckets which contain the two endpoints of the range, we recalculate the max in $O(\sqrt{N})$ time. For the buckets contained in the interval, we can just update the lazy counter. This way, we process 2 endpoints in $O(\sqrt{N})$ time and $\sqrt{N}$ whole buckets in $O(1)$ time each.

To query the maximum of a range we use a similar strategy. For the two buckets on the end of the range we loop over their elements in the range and find the max. For the $\sqrt{N}$ buckets in the middle, we just take the stored maximum. This also takes $O(\sqrt{N})$ time.

3 Cow at Large

Given a tree of $2 \leq N \leq 70,000$ nodes we need to find the number of farmers needed to catch Bessie if she starts at each barn. For every node we can find its distance to the nearest leaf. This means the farmers can reach a node before Bessie if and only if its distance to a leaf is less than or equal to that node's distance to Bessie.

If the farmers and get to a node first, then the farmers can also reach all of it's children first. This means that the nodes the farmers can reach first form a set of subtrees of the whole tree. For each subtree one farmer can reach the root of the subtree before Bessie and cut her off. This means that the number of farmers needed is the number of subtrees where the farmers get there first.

For any subtree of N nodes there are exactly $N-1 edges$. If we sum up the degree of every node in that subtree we will get $2N-1$ because every edge is counted twice, and the root of the subtree has one edge to its parent. Note that this is false if the subtree is the entire tree, but that case will never appear in this problem. If we take $\sum 2 - deg(v)$ this will give us exactly one. Likewise if we take the sum over all nodes the farmers can reach first it will be the number of subtrees, which is the number of farmers needed.

Now we can move Bessie's starting location around the tree in a DFS. As we o this we need to maintain an efficient way of summing $2-deg(v)$ over all nodes where $distLeaf(v) \leq distBessie(v)$. When we move the starting location along an edge we increment the distance to Bessie in the entire subtree of the node we are leaving. We also decrement the distance of Bessie in the subtree of the node we move to.

If we do an inorder traversal of the tree from the beginning node of the DFS, then every subtree, is a range in that inorder traversal. So as we move Bessie's starting node, we need to update the depth of a range of the inorder traversal. WE also need to find the sum of $2 - deg(v)$ for all nodes where $leafDist - besieDist \leq 0$. To do this we can divide the inorder traversal into $\sqrt{N}$ buckets. Each bucket will store the depth of every node in it, a lazy counter for depth updates, and a BIT. If a node has a depth difference of d, it contributes $2 - deg(v)$ to index d of the BIT. This means that BITQuery(0) is the sum of $2 - deg(v)$ for all nodes the farmers can reach first. To use the lazy counter, only query the BIT up to the lazy counter. This means that when we increment or decrement the difference of an entire block we just update the lazy counter and shift future queries to the BIT.

So what we do is run a DFS from a starting node u. Whenever we move the Bessie's starting location up or down an edge we update the depths of both subtrees. We will update $\sqrt{N}$ lazy counters in the middle of our range. On each end we can just go through the range and individually update the depths and rebuild the BIT for that block in $O(\sqrt{N}\log N)$. To find the number of farmers needed if Bessie starts at the current node we just sum up $2 - dev(v)$ for all nodes with difference ¡ 0. To do this we just query every block's BIT, taking into account the lazy counter.

4 Problems

1. (SPOJ D-QUERY) Given a sequence of n numbers a1, a2, ..., an and q d-queries. A d-query is a pair (i, j) $(1 \leq i \leq j \leq n)$. For each d-query (i, j), you have to return the number of distinct elements in the subsequence ai, ai+1, ..., aj. $(1 \leq n \leq 30000, 1 \leq a_i \leq 10^6, 1 \leq q \leq 2 * 10^5)$.

2. (Codechef Sherlock and Inversions) Given an array A of size N and a list of Q number of queries, where each query has two numbers L and R, find for each query the number of inversions in the subarray from Lth to Rth position (both inclusive) of the array A (1-based index). For an array A two elements A[i] and A[j] form an inversion if A[i] > A[j] and i < j. $(N \leq 20000, A[i] \leq 10^9, Q \leq 20000)$