The Longest Smooth Subarray Problem

Input: A Read-Only arbitrary array $A[1..n]$ of n real numbers, and a real number $D > 0$.
Output: The longest contiguous subarray $A[i..j]$, $1 \le i \le j \le n$, such that no pair of elements of that subarray has a difference more than D.
(Note: array A is read-only, so you cannot rearrange its elements.)

Design and analyze an efficient *Incremental* algorithm to solve this problem using the *Loop Invariant Method*.

Solution:
Note: This problem and its solution also appear in Lecture Slide 4.
Below we will develop an $O(n)$ time algorithm to solve the problem. The algorithm is incremental and the development is similar to the solution we discussed in class for the maximum sum subarray problem.

A subarray is Smooth iff the difference between its maximum and minimum elements is at most D. We want to find the longest Smooth subarray of $A[1..n]$. We concentrate on developing the right Loop Invariant. We will do this by starting from the obvious incremental LI and gradually strengthening it until we have it right. So, as a start, suppose we have scanned the prefix $A[1..t]$ of the given array and we want to have maintained its Longest Smooth Subarry, denoted $LSS(A[1..t])$. So our incomplete LI so far is:

LI(i): $A[i..j] = LSS(A[1..t])$, $1 \le i \le j \le t \le n$.

How do we maintain LI(i) when we go from iteration $t-1$ to t? That is, suppose we have $LSS(A[1..t-1])$. How do we update that to $LSS(A[1..t])$? We observe that $LSS(A[1..t])$ either uses the "last" element $A[t]$ or it doesn't. If it doesn't use $A[t]$, i.e., $j \le t-1$, then $LSS(A[1..t]) = LSS(A[1..t-1])$. What if $LSS(A[1..t])$ includes $A[t]$? In that case, $LSS(A[1..t])$ is a suffix of $A[1..t]$, say it's the suffix $A[k..t]$. As such, it must be the Longest Smooth Suffix of $A[1..t]$. Let $LSX(A[1..t])$ denote the Longest Smooth Suffix of $A[1..t]$. So, $LSS(A[1..t])$ is either $LSS(A[1..t-1])$ or $LSX(A[1..t])$, whichever is longer. We see we need that added information in the Loop Invariant also:

LI(ii): $A[k..t] = LSX(A[1..t])$, $\quad 1 \le k \le t$.

We now have to ask the question, how do we maintain LI(ii) when we go from iteration $t-1$ to t? Suppose $A[k'..t-1] = LSX(A[1..t-1])$. Now consider adding $A[t]$ to it to form $A[k'..t]$. If the latter is Smooth, then it is $LSX(A[1..t])$. If it's not Smooth, let $A[min_1]$ and $A[max_1]$ be, respectively, the rightmost minimum and rightmost maximum elements of $A[k'..t-1]$. Since $A[k'..t-1]$ is Smooth and $A[k'..t]$ is not, we conclude that either $A[t] > A[min_1] + D$ or $A[t] < A[max_1] - D$ (but not both, because $A[k'..t-1]$ is Smooth and hence $A[max_1] - A[min_1] \le D$). Let us consider the first case, i.e., $A[t] > A[min_1] + D$. Then $A[k..t] = LSX(A[1..t])$ cannot use any index below $min_1 + 1$. (Why?) To update the LSX, we may increase k' to $min_1 + 1$. But we may have the same problem with the new k'. Now suppose $A[min_2]$ is the rightmost minimum element of the new $A[k'..t-1]$. The process has to be repeated. A similar argument can be made if $A[t] < A[max_1] - D$. In

that case we would repeatedly need to find the index of the rightmost maximum element of $A[k'..t-1]$ and cut it off at that point. This suggests to strengthen the Loop Invariant further by requiring these additional min/max index information at hand.

To this end, let us define the two lists $MinL$ and $MaxL$ as follows. At the end of iteration t, MinL will hold the indices $(min_1, min_2, \cdots, min_p)$ with the following properties: $k-1 = min_0 < min_1 < min_2 < \cdots < min_{p-1} < min_p = t$, and for $m = 1..p$, $A[min_m]$ is the rightmost minimum element of $A[min_{m-1}+1\ ..\ t]$. Similarly, we define MaxL to be the list $(max_1, max_2, \cdots, max_q)$ with the following properties: $k-1 = max_0 < max_1 < max_2 < \cdots < max_{q-1} < max_q = t$, and for $m = 1..q$, $A[max_m]$ is the rightmost maximum element of $A[max_{m-1}+1\ ..\ t]$. So, we add these to the Loop Invariant:

LI(iii): Lists MinL and MaxL are as defined above.

Consider these two lists $MinL$ and $MaxL$. For each of these two lists we let its *rear* be on the smaller index side (i.e., min_1 and max_1 respectively), and its *front* be on the larger index side (i.e., index t). We need the following $O(1)$ time list operations: $Rear(L)$ and $Front(L)$ return the rear and front items of list L (without changing contents of L), and the insert/delete list operations at either end which we denote by $PushFront$, $PopFront$ and $PopRear$ (no $PushRear$ needed).

The algorithm appears below. The complete **Loop Invariant** for the main for-loop is:

LI(i): $A[i..j] = LSS(A[1..t])$, $1 \le i \le j \le t \le n$.
LI(ii): $A[k..t] = LSX(A[1..t])$, $1 \le k \le t$.
LI(iii): Lists MinL and MaxL are as defined above.

Algorithm $LongestSmoothSubarray(A[1..n], D)$

***PreCond*:** Array $A[1..n]$ of n real elements, and real $D > 0$.

***PostCond*:** Returns Longest Smooth Subarray of $A[1..n]$.

begin
▷ Establish LI for $t = 1$:
1. $i \leftarrow j \leftarrow k \leftarrow 1$
2. $MinL \leftarrow MaxL \leftarrow \{1\}$
3. **for** $t \leftarrow 2..n$ **do** ▷ Establish LI for t:
▷ Update LSX and establish LI(ii) and LI(iii):
4a. **while** $MinL \neq \emptyset$ and $A[t] > A[Rear(MinL)] + D$ **do** $k \leftarrow 1 + PopRear(MinL)$
4b. **while** $MaxL \neq \emptyset$ and $Rear(MaxL) < k$ **do** $PopRear(MaxL)$
5a. **while** $MaxL \neq \emptyset$ and $A[t] < A[Rear(MaxL)] - D$ **do** $k \leftarrow 1 + PopRear(MaxL)$
5b. **while** $MinL \neq \emptyset$ and $Rear(MinL) < k$ **do** $PopRear(MinL)$
6a. **while** $MinL \neq \emptyset$ and $A[t] \le A[Front(MinL)]$ **do** $PopFront(MinL)$
6b. **while** $MaxL \neq \emptyset$ and $A[t] \ge A[Front(MaxL)]$ **do** $PopFront(MaxL)$
6c. $PushFront(t, MinL)$; $PushFront(t, MaxL)$
▷ Update LSS and establish LI(i):
7. **if** $t - k > j - i$ **then** $(i, j) \leftarrow (k, t)$
8. **end-for**

9. **return** $A[i..j]$
end algorithm

Correctness: Lines 1-2 establish the LI for $t = 1$. For $t = 2..n$, LI is established by iteration t of the for-loop of lines 3-8. The purpose of lines 4a-6c is to establish LI(ii) and LI(iii). Line 7 establishes LI(i) by taking the longer of $A[i..j] = LSS(A[1..t-1])$ and $A[k..t] = LSX(A[1..t])$ to become $LSS(A[1..t])$. Line 9 establishes the post-condition.

We need to further explain the purpose of the while-loops of lines 4a-6b. As we said earlier, when we insert $A[t]$ at the end of the Longest Smooth Suffix $A[k..t-1]$, the result $A[k..t]$ may not be Smooth, either because $A[t] > \min(A[k..t-1]) + D = A[Rear(MinL)] + D$, or $A[t] < \max(A[k..t-1]) - D = A[Rear(MaxL)] - D$. We note that at most one of these two cases (or none) can occur. If any of these two cases occurs, we need to cut off some rear portion of the two lists and update k to a proper larger value. This is done in the first case by lines 4a-b, and in the second case by lines 5a-b. We then need to push t at the front of both lists while maintaining LI(iii). To accomplish that, before the push we should pop some front elements that are not rightmost minima/maxima indices anymore. Lines 6a-c accomplish that.

Time Analysis: At first it seems that since we have some while-loops nested within the for-loop, the algorithm takes quadratic time. However, let's look more closely. At line 2 we see that MinL and MaxL have size 1. Then, line 6c is the only place where we push an item into these two lists. So, a total of n items are ever pushed onto each of the two lists. Each iteration of the while-loops of lines 4a-6b performs a pop at some end of one of these two lists. Each such pop must correspond to an item that has been previously pushed into that list. Since we push at most n items into each of the lists, we can have at most n pops on each list also. So, the total number of iterations of the while-loops (over all iterations of the for-loop) is $O(n)$. Therefore, the entire algorithm runs in $O(n)$ time.