Max Distance Monotonic

Here we show how the Codility Challenge codenamed (Natrium-2014) can be solved. You can still give it a try, but no certificate will be granted. The problem asks you to find a pair of indices (P, Q), such that $A[P] \leqslant A[Q]$ and the distance between P and Q is maximal, that is the value $Q - P$ is maximal.

Slow solution $O(N^2)$

The simplest solution is to check every pair (P, Q) and choose a pair, such that $A[P] \leqslant A[Q]$, with the maximal distance between P and Q.

1: Slow solution — $O(N^2)$.

```
def maxDistanceMonotonicSlow(A):
    N = len(A)
    result = 0
    for P in xrange(N - 1):
        for Q in xrange(P + 1, N):
            if A[P] <= A[Q]:
                result = max(result, Q - P)
    return result
```

The time complexity of the above algorithm is $O(N^2)$, which is far from optimal.

Fast solution $O(N \log N)$

There is an easy way to improve the running time by going through all the elements of the array in the non-decreasing order. It suffices to sort the array remembering the original positions of the elements, so that $A[0] \leqslant A[1] \leqslant \ldots \leqslant A[N-1]$.

For every element of the sorted array A (for every index Q), we will assume that this is the second element of the pair. Then, we have to find the first element of the pair at some index P, such that $A[P] \leqslant A[Q]$ and the distance between the original positions is maximal. Notice that the candidates for the first element of the pair are only elements with indices $1, 2, \ldots, Q - 1$. Only these elements are not greater than $A[Q]$, and of them the best is that with the smallest original position. The minimal original position can be updated with every new element in a constant time.

2: Fast solution — $O(N \log N)$.

```
def maxDistanceMonotonicFast(A):
    N = len(A)
    result = 0
    pairs = []
    for i in xrange(N):
        pairs.append((A[i], i))
    pairs.sort()
    minOriginalPos = N
    for (a, b) in pairs:
        minOriginalPos = min(minOriginalPos, b)
        result = max(result, b - minOriginalPos)
    return result
```

We have to sort all the elements, so the time complexity of the above algorithm is $O(N \log N)$.

Golden solution $O(N)$

There is an even better way of solving this task. As in the above solution, for every element (for every index Q), we will assume that this is the second element of the pair. Next, we have to find the first element of the pair at some index P, such that $A[P] \leqslant A[Q]$ and the distance between the positions is maximal.

Let's consider which elements can be the first elements of the pair. We can create a list of such candidates, starting with the first element of the array. The next value can only be smaller. If this were not true, then we could use the smaller value in the earlier position and the distance would be larger. Thus, the candidates form a decreasing sequence.

Having computed the list of all candidates for the element $A[P]$, we have to find, for every index Q, the candidate (index P) with the minimal position. If we started our search for the candidate from the beginning every time, we would achieve a quadratic time. A better approach would be to perform a binary search for index P in the pool of candidates. That would produce an $O(N \log N)$ solution, as a binary search for a single element in a sorted array works in $O(\log N)$ time.

The best approach is to iterate through all the candidates from the minimal to the maximal values. At the same time we iterate through the array, in reverse order (from the last to the first element), considering the second elements $A[Q]$. Notice that if we find the best candidate $A[P] \leqslant A[Q]$, then there is no need to check candidates with higher positions in the future, because the distance would be only smaller (as the value Q would be smaller). Similarly, if we find $A[Q]$ such that $A[P] \leqslant A[Q]$ for the candidate $A[P]$, $Q - P$ is the best possible result to which $A[P]$ contributes. All other possible elements can only give shorter distances.

3: Golden solution — $O(N)$.

```
def solution(A):
    N = len(A)
    # list of candidates for the first element of the pair
    candidates = []
    for P in xrange(N):
        if (len(candidates) == 0 or A[P] < first(candidates[-1])):
            candidates.append((A[P], P))
    # finding the best distance
    result = -1
    for Q in xrange(N - 1, -1, -1):
        while (len(candidates) > 0 and A[Q] >= first(candidates[-1])):
            result = max(result, Q - second(candidates[-1]))
            candidates.pop()
    return result
```

The time complexity is $O(N)$, because with each step of the while loop, the pool of candidates decreases by one.