Count Bounded Slices

It's time to show you how the Codility Challenge codenamed (Oxygenium) can be solved. You can still give it a try, but no certificate will be granted. The problem asks you to calculate the number of slices in which ($max - min \leqslant K$). Such slices are described as being *bounded*.

Slow solution $O(N^2)$

For every index i we can find the largest index $j \geqslant i$ such that (i, j) is a bounded slice. Any slice with a larger value of j will not be bounded.

Using so called brute-force approach, we can check increasing indices, remembering the minimum and the maximum values at all times.

1: Slow solution — $O(N^2)$.

```
def boundedSlicesSlow(K, A):
    N = len(A)
    result = 0
    for i in xrange(N):
        minimum = A[i]
        maximum = A[i]
        for j in xrange(i, N):
            maximum = max(maximum, A[j])
            minimum = min(minimum, A[j])
            if maximum - minimum <= K:
                result += 1
                if result == maxINT: # maxINT = 10^9
                    return result
            else:
                break
    return result
```

The time complexity of the above algorithm is $O(N^2)$.

Fast solution $O(N \log N)$

Notice that if the slice (i, j) is bounded then every slice $(i + 1, j)$, $(i + 2, j)$, ..., (j, j) is bounded too. There is no need to check index j from the beginning. Let's assume there is a *minMaxQuery* function, which returns the difference between maximum and minimum values in a given slice. The algorithm could appear as follows:

2: Fast solution — $O(N \log N)$.

```
def boundedSlicesFast(K, A):
    N = len(A)
    result = 0
    j = 0
    for i in xrange(N):
        while (j < N):
            if (minMaxQuery(i, j) <= K):
                j += 1
            else:
                break
        result += (j - i)
        if result >= maxINT:
            return maxINT
    return result
```

There is a question of how to find the minimum and the maximum values in the slices. We can use a popular Interval_tree data structure. Finding the minimum/maximum values works in $O(\log N)$ time complexity. Thus, 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. Questions about intervals take a particular form: the interval *crawls* over the array and there is enough information to create a structure similar to a queue, supporting operations such as inserting elements at one end, removing them from the other end and finding maximum/minimum value. Let's focus on finding the maximum element (finding the minimum is analogical).

Let's consider which elements can become maximal. Of course, if there is already an element in the queue, and some bigger number is then inserted, then the previous element will never be the maximum. Thus, we don't need to remember its value. The numbers that remain will form a non-increasing sequence, so finding the maximum element is a simple matter of just taking the oldest element from the queue (in chronological order).

3: Golden solution — $O(N)$.

```
def boundedSlicesGolden(K, A):
    N = len(A)

    maxQ = [0] * (N + 1)
    posmaxQ = [0] * (N + 1)
    minQ = [0] * (N + 1)
    posminQ = [0] * (N + 1)

    firstMax, lastMax = 0, -1
    firstMin, lastMin = 0, -1
    j, result = 0, 0

    for i in xrange(N):
        while (j < N):
            # added new maximum element
            while (lastMax >= firstMax and maxQ[lastMax] <= A[j]):
                lastMax -= 1
            lastMax += 1
            maxQ[lastMax] = A[j]
            posmaxQ[lastMax] = j

```

```
            # added new minimum element
            while (lastMin >= firstMin and minQ[lastMin] >= A[j]):
                lastMin -= 1
            lastMin += 1
            minQ[lastMin] = A[j]
            posminQ[lastMin] = j

            if (maxQ[firstMax] - minQ[firstMin] <= K):
                j += 1
            else:
                break
        result += (j - i)
        if result >= maxINT:
            return maxINT
        if posminQ[firstMin] == i:
            firstMin += 1
        if posmaxQ[firstMax] == i:
            firstMax += 1
    return result
```

The time complexity is $O(N)$, because each operation works in amortized constant time.