Other Range queries

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

These notes sketch the content of 7th lecture. These are rough and non-exaustive notes that I used while lecturing. Please use them just to have a list of topics of each lecture and use the reported references to study these arguments.

1 Colored range query problem

▶ **Problem 1** (Colored range query problem). *Given an array $A[1, n]$ of colors (integers in the range $[1, n]$), we would like to support the query* Distinct(i, j) *which returns all the distinct colors that occur in $A[i, \ldots, i]$.*

The naïve solution scans the range from i to j and reports each color in $A[i, j]$ exactly once. Obviously, the complexity of this strategy is $\Theta(j - i + 1)$ time in the worst case.

Our goal, instead, is to support any query in time proportional to the number of reported colors, i.e., $\Theta(k)$ time where k is the number of distinct colors in the range.

The idea is to use RMQ queries over an appropriate array $P[1, n]$. Given the array $A[1, n]$, we construct the array $P[1, n]$ as follows. The entry $P[i]$ equals to the rightmost occurrence of $A[i]$ in $A[1, i - 1]$, or 0 if such occurrence does not exist. We build a data structure to answer range minimum queries (RMQ) on top of P.

The figure below shows an array A and the corresponding array P.

	1	2	3	4	5	6	7	8
A	2	4	4	3	4	1	2	3

	1	2	3	4	5	6	7	8
P	0	0	2	0	3	0	1	4

The query Distinct$(2, 5)$ has to report $4, 3$. Our goal is to design an algorithm that reports only the first occurrence of each color in the range $A[i, j]$. Note that the first occurrence of a color is such that its corresponding value in P is smaller than i. Why? This means that our goal is equivalent to finding all the entries in $P[i, j]$ that are smaller than i. The reporting algorithm is as follows. In order to find all ℓ in $[i, j]$ with $P[\ell] < i$, we find m in $[i, j]$ such that $P[m]$ is the smallest value in $P[i, j]$ using RMQ(i, j). If $P[m] \geq i$, we stop. Otherwise, we output $A[m]$ and we solve the same problem on $[i, m - 1]$ and $[m + 1, j]$. In all the recursive calls we are looking for entries smaller than i (i.e., the left end of the initial range) and we stop the recursion in a subrange as soon as its smallest value is larger than or equal to i.

Let k be the number of distinct colors in the range. The algorithm performs at most $2k$ RMQ queries. Why? Thus, its time complexity is $\Theta(k)$ as required.

2 MO's algorithm

The description here is based on the blog post at `https://blog.anudeep2011.com/mos-algorithm/`.

In the previous problems we solved different kinds of range queries ($\mathsf{Sum}(i)$, $\mathsf{RMQ}(i,j)$, $\mathsf{Distinct}(i,j)$). In these cases there exist appropriate data structures to be built on top of the array A such that a query can be solved efficiently and online. However, there are more difficult kind of queries for which there is no such online efficient data structure.

For some of these queries, the best we can hope for is an algorithm which is able to solve efficiently only a batch of queries. Thus, the time complexity of a query is efficient only in an amortized sense.

The MO's algorithm is a technique which can be instantiated to solve (offline) several kind of queries. In several cases, it guarantees that, if the batch contains $\Omega(n)$ queries, then each query can be solved in $\Theta(\sqrt{n})$ amortized time.

Consider the following problem.

▶ **Problem 2.** *We are given an array $A[1,n]$ of colors (integers in $[1,n]$) and a set Q of m range queries* $\mathsf{3OrMore}$. *The query* $\mathsf{3OrMore}(l,r)$ *has to report all the colors that occurs at least three times in $A[l,r]$.*

A naïve algorithm solve each query by, potentially, scanning the whole array A. Thue, this takes $\Theta(mn)$ time. The following is the pseudocode of the algorithm.

```
Solve(A, Q)
1.  for each ⟨l, r⟩ ∈ Q
2.      answer = 0
3.      counter[1,n] = [0] * n
4.      for(k = l; k < r; ++k)
5.          counter[A[k]]++
6.          if counter[A[k]] == 3
7.              answer++
8.      print answer
```

Consider instead the following algorithm with the same worst-case time complexity.

```
Add(p)
1.      counter[A[p]]++
2.      if counter[A[k]] == 3
3.          answer++
```

```
Remove(p)
1.      counter[A[p]]–
2.      if counter[A[k]] == 2
3.          answer–
```

```
Solve(A, Q)
1.  current_l = 0
2.  current_r = 0
3.  answer = 0
4.  counter[1,n] = [0] * n
5.  for each ⟨l, r⟩ ∈ Q
6.      ## current_l moves to l, current_r moves to r
7.      while (current_l < l)
```

```
8.          current_l++
9.          Add(current_l)
10.     while (current_l > l)
11.         current_l–
12.         Remove(current_l)
13.     while (current_r < r)
14.         current_r++
15.         Add(current_r)
16.     while (current_r > r)
17.         current_r–
18.         Remove(current_r)
19.     print answer
```

It should be clear that the second algorithm is correct and that its worst-case complexity is $\Theta(nm)$ time.

MO's algorithm is just a reordering of the queries such that the complexity of the second algorithm reduces significantly. Let us divide the array A into $\sqrt{n}$ buckets of size $\sqrt{n}$, named $B_1, B_2, \ldots B_{\sqrt{n}}$. A query belongs to bucket B_k if and only if its left end l fall into kth bucket, i.e., $\lfloor l/\sqrt{n} \rfloor = k$. First, We group the queries accordingly to their buckets. Queries within the same bucket are solved in increasing order of their right ends. What is the time complexity to process a bucket of queries? As we process the queries in increasing order of their right ends, current_r in the above algorithm can only increase from 1 to n, thus, it moves only n steps. Instead, current_l may both increase and decrease. However, it cannot moves by more than $\sqrt{n}$ steps per query. Thus, if the bucket has b queries, the time to process its queries is at most $\Theta(b\sqrt{n} + n)$. Summing up over all buckets, the time complexity becomes $\Theta(m\sqrt{n} + n\sqrt{n})$. This is $\Theta(m\sqrt{n}$ amortized time per query, whenever $m = \Omega(n)$.

MO's algorithm is offline, i.e., we cannot use it when we are forced to stick to given order of queries or when there are update operations. Not just that, there is one important possible limitation. We should be able to write the functions Add and Remove. There will be many cases where add is trivial but remove is not. One such example is where we want maximum in a range. As we add elements, we can keep track of maximum. But when we remove elements it is not trivial. Clearly, this issue can be solved with a Max-Heap, but this adds a $\log n$ factor to the overall time complexity. Thus, in this case the amortized time per query is at least $\Theta(\sqrt{n} \log n)$, which is much worse that the adhoc (and online) solution we discussed last lecture. This should not be a surprise, adhoc solutions that exploit particular property of the problem at hand may be much better than a general technique, like MO's algorithm.