Anudeep's blog

MO’s Algorithm (Query square root decomposition)

28 December 2014 anudeep2011 Algorithms 69 Comments

Once again I found a topic that is useful, interesting but has very less resources online. Before writing this I did a small survey and surprised that almost none of the Indian programmers knew this algorithm. Its important to learn this, all the red programmers on codeforces use this effectively in div 1 C and D problems. There were not any problems on this an year and half back, but from then there is a spike up! We can expect more problems on this in future contests.

1) State a problem
2) Explain a simple solution which takes O(N^2)
3) Slight modification to above algorithm. It still runs in O(N^2)
4) Explain an algorithm to solve above problem and state its correctness
5) Proof for complexity of above algorithm – O(Sqrt(N) * N)
6) Explain where and when we can use above algorithm
7) Problems for practice and sample code

State a problem

Given an array of size N. All elements of array <= N. You need to answer M queries. Each query is of the form L, R. You need to answer the count of values in range [L, R] which are

repeated at least 3 times.
Example: Let the array be {1, 2, 3, 1, 1, 2, 1, 2, 3, 1} (zero indexed)
Query: L = 0, R = 4. Answer = 1. Values in the range [L, R] = {1, 2, 3, 1, 1} only 1 is repeated at least 3 times.
Query: L = 1, R = 8. Answer = 2. Values in the range [L, R] = {2, 3, 1, 1, 2, 1, 2, 3} 1 is repeated 3 times and 2 is repeated 3 times. Number of elements repeated at least 3 times = Answer = 2.

Explain a simple solution which takes O(N^2)

For each query, loop from L to R, count the frequency of elements and report the answer. Considering M = N, following runs in O(N^2) in worst case.

```
for each query:
  answer = 0
  count[] = 0
  for i in {l..r}:
    count[array[i]]++
    if count[array[i]] == 3:
      answer++
```

Slight modification to above algorithm. It still runs in O(N^2)

```
add(position):
  count[array[position]]++
  if count[array[position]] == 3:
    answer++

remove(position):
  count[array[position]]--
  if count[array[position]] == 2:
    answer--

currentL = 0
```

```
currentR = 0
answer = 0
count[] = 0
for each query:
  // currentL should go to L, currentR should go to R
  while currentL &amp;lt; L:
    remove(currentL)
    currentL++
  while currentL &amp;gt; L:
    add(currentL)
    currentL--
  while currentR &amp;lt; R:
    add(currentR)
    currentR++
  while currentR &amp;gt; R:
    remove(currentR)
    currentR--
  output answer
```

Initially we always looped from L to R, but now we are changing the positions from previous query to adjust to current query.
If previous query was L=3, R=10, then we will have currentL=3 and currentR=10 by the end of that query. Now if the next query is L=5, R=7, then we move the currentL to 5 and currentR to 7.
add function means we are adding the element at position to our current set. And updating answer accordingly.
remove function means we are deleting the element at position from our current set. And updating answer accordingly.
Edit: Have a look a Shubajit Saha’s comment. And also this.

Explain an algorithm to solve above problem and state its correctness

MO’s algorithm is just an order in which we process the queries. We were given M queries, we will re-order the queries in a particular order and then process them. Clearly, this is an offline algorithm. Each query has L and R, we will call them opening and closing. Let us divide the given input array into Sqrt(N) blocks. Each block will be N / Sqrt(N) = Sqrt(N) size. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.

A query belongs to P’th block if the opening of that query fall in P’th block. In this algorithm we will process the queries of 1st block. Then we process the queries of 2nd block and so on.. finally Sqrt(N)’th block. We already have an ordering, queries are ordered in the ascending order of its block. There can be many queries that belong to the same block.

From now, I will ignore about all the blocks and only focus on how we query and answer block 1. We will similarly do for all blocks. All of these queries have their opening in block 1, but their closing can be in any block including block 1. Now let us reorder these queries in ascending order of their R value. We do this for all the blocks.

How does the final order look like?
All the queries are first ordered in ascending order of their block number (block number is the block in which its opening falls). Ties are ordered in ascending order of their R value.
For example consider following queries and assume we have 3 blocks each of size 3.
{0, 3} {1, 7} {2, 8} {7, 8} {4, 8} {4, 4} {1, 2}
Let us re-order them based on their block number.
{0, 3} {1, 7} {2, 8} {1, 2} {4, 8} {4, 4} {7, 8}
Now let us re-order ties based on their R value.
{1, 2} {0, 3} {1, 7} {2, 8} {4, 4} {4, 8} {7, 8}

Now we use the same code stated in previous section and solve the problem. Above algorithm is correct as we did not do any changes but just reordered the queries.

Proof for complexity of above algorithm – O(Sqrt(N) * N)

We are done with MO’s algorithm, it is just an ordering. Awesome part is its runtime analysis. It turns out that the O(N^2) code we wrote works in O(Sqrt(N) * N) time if we follow the order i specified above. Thats awesome right, with just reordering the queries we reduced the complexity from O(N^2) to O(Sqrt(N) * N), and that too with out any further modification to code. Hurray! we will get AC with O(Sqrt(N) * N).

Have a look at our code above, the complexity over all queries is determined by the 4 while loops. First 2 while loops can be stated as “Amount moved by left pointer in total”, second 2 while loops can be stated as “Amount moved by right pointer”. Sum of these two will be the over all complexity.

Most important. Let us talk about the right pointer first. For each block, the queries are sorted in increasing order, so clearly the right pointer (currentR) moves in increasing order. During the start of next block the pointer possibly at extreme end will move to least R in next block. That means for a given block, the amount moved by right pointer is O(N). We have O(Sqrt(N)) blocks, so the total is O(N * Sqrt(N)). Great!

Let us see how the left pointer moves. For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and current L fall in the same block, the moment is O(Sqrt(N)) (Size of the block). In each block the amount left pointer movies is O(Q * Sqrt(N)) where Q is number of queries falling in that block. Total complexity is O(M * Sqrt(N)) for all blocks.

There you go, total complexity is O((N + M) * Sqrt(N)) = O(N * Sqrt(N))

Explain where and when we can use above algorithm

As mentioned, this algorithm is offline, that means we cannot use it when we are forced to stick to given order of queries. That also means we cannot use this 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. Anyways in that case we can use a set to add elements, remove elements and report minimum. In that case the add and delete operations are O(log N) (Resulting in O(N * Sqrt(N) * log N) algorithm).

There are many cases where we can use this algorithm. In few cases we can also use other Data Structures like segment trees, but for few problems using MO’s algorithm is a must. Lets discuss few problems in the next section.

Problems for practice and sample code

DQUERY – SPOJ: Number of distinct elements in a range = number of elements with frequency >= 1. So it is the same problem we discussed above.

Click here for sample code

Note: That code will give TLE on submission, it will give AC if fast I/O is added. Removed fast I/O to keep code clean.

Powerful array – CF Div1 D: This is an example where MO’s algorithm is a must. I cannot think of any other solution. CF Div1 D means it is a hard problem. See how easy it is using MO’s algorithm in this case. You only need to modify add(), remove() functions in above code.

GERALD07 – Codechef

GERALD3 – Codechef

Tree and Queries – CF Div1 D

Powerful Array – CF Div1 D

Jeff and Removing Periods – CF Div1 D

Sherlock and Inversions – Codechef

I am sure there are more problems, if you know any of them, do comment, i will add them. While this algorithm has a special name “MO”, it is just smart square root decomposition.

Signing off! Wish you a happy new year

Share this post! Learn and let learn

Share And Smile:

Facebook Reddit Twitter Email More

When 2 guys talk, its not always about girls/sports ;)

Machine learning everywhere, why not in Competitive programming?

Subscribe to our Newsletter

Email*

SUBSCRIBE!

Categories

Algorithms (2)

Data Structures (1)

Machine Learning (1)

Segment trees (2)

SPOJ (1)

Uncategorized (2)

Recent Posts

20 by 25
September 26, 2016

Machine learning everywhere, why not in Competitive programming?
April 18, 2016

MO’s Algorithm (Query square root decomposition)
December 28, 2014

When 2 guys talk, its not always about girls/sports ;)
July 18, 2014

Persistent segment trees – Explained with spoj problems
July 13, 2014

Heavy Light Decomposition
April 11, 2014

Recent Comments

Rohit on Persistent segment trees – Explained with spoj problems

Jaswant Singh on Persistent segment trees – Explained with spoj problems

Vineeth Kanaparthi on 20 by 25

nabil1997 on MO’s Algorithm (Query square root decomposition)