How exactly is the square root decomposition of queries (also sometimes referred to as Mo's Algorithm), used for offline processing of queries?

If possible, please explain with an example.

NOTE I :- I've gone over Mo's Algorithm - Codeforces post, but it lacks in coverin... More

If possible, please explain with an example.

NOTE I :- I've gone over Mo's Algorithm - Codeforces post, but it lacks in covering the topic with an example and complexity analysis properly.

NOTE II :- Here are some problems that I encountered :-

a) Problem - D - Codeforces

b) SPOJ.com - Problem ZQUERY

Related Question: How does the technique of sqrt N decomposition work and in what kind of problems is it useful?

1 ANSWER

John Kurlak, works at Facebook

Updated 16 Aug 2015

Mo's algorithm is an efficient way to compute the answers to a bunch of range queries.

A range query is a query that asks for the result of some computation over the values in a particular range. For example, if you have an array, one range query might be, "What is the sum of the values from index 5 to index 10?" Another range query might be, "What is the minimum value among the integers from index 2 to index 98?"

For Mo's algorithm to apply, the following conditions must be met:

> You must have data that falls in a range (typically, this means you have an array of values).
>
> You must know the queries ahead of time (this is why it's called an offline algorithm).
>
> You must be able to incrementally build the answer for a particular query by including/excluding one value in the range at a time. For example, to find the sum of values in a range, you can maintain a running sum of values you've seen so far, and you can update that value as you encounter more values in the range. To remove values from the left side of the range, you can subtract the running sum of those values.

The basic idea behind Mo's algorithm is this:

Reduce the overall number of calculations across all queries by reusing the result of overlapping queries.

For example, say we have a query over the range [0, 100] and a query over the range [50, 150]. Mo's algorithm says: "Both queries share calculations for the range [50, 100], so let's not do those calculations twice!"

This idea is very similar to the idea behind dynamic programming. The difference is that Mo's algorithm does not require optimal substructure (it has it's own set of requirements though, as I have mentioned above).

So how do we reuse calculations? Well, the answer is that we first calculate the result of a single query. We don't need to apply any tricks to make this calculation. We just calculate it directly.

Once we have the result to a single query, we can transform it into the result for the next query. For example, suppose we have the array:

4, 8, 2, 5, 6, 3, 3 (values)
0, 1, 2, 3, 4, 5, 6 (indices)

And suppose we have the sum of values from index 1 to index 3:
8+2+5 = 15.

Now, suppose our next query is the sum of values from index 2 to index 5:
2+5+6+3 = 16.

Instead of calculating the result of the second query directly, let's see how we can transform the result of the first query to get the answer for the second query.

Well, our start index must increase from 1 to 2. Therefore, we need to exclude the contribution from index 1. Since our problem involves the sum over a range, the way we **exclude** a single value is to**subtract** it from our existing answer. Since the answer to query 1 is 15, we have to subtract the value at index 1 from 15. Thus, we have: 15 - 8 = 7. Now we have the result for the range from index 2 to index 3. Let's verify that result: 2 + 5 = 7. Good.

Now, our end index must increase from 3 to 5. Therefore, we need to include the contributions from indices 4 and 5. Since our problem involves the sum over a range, the way we **include** a single value is to **add** it to our existing answer. Since the answer to the query from index 2 to index 3 is 7, we have to add the values at indices 4 and 5 to 7. Thus, we have: 7 + 6 + 3 = 16. And that's the correct value for the sum from index 2 to index 5.

Thus far, I have shown how to take the answer from a single query and transform it to the answer of another query. This process is only possible if condition (3) from above is met. If we cannot incrementally generate the answer to a single query, then there is no way to include/exclude a single value at a time, and we cannot slowly morph the answer from one query to the answer for another query.

The exact steps you take to include/exclude a value from consideration will vary as the problem varies. For summations over a range, you use addition to include a value and subtraction to exclude a value. For the minimum value over a range, you might use a heap with a hash map for inclusions/exclusions (use the hash map for fast removals). For the median of a range, you might use a min heap and a max heap combination, where one heap stores the top n/2 values and another heap stores the bottom n/2 values. Again, you would want a hash map here. Any algorithm that allows you to calculate the *running* _____ (whether it be *running sum*, *running median*, etc.) will be what you're looking for here. However, if you can't modify that algorithm to exclude values, then you might not be able to use it. For example, you could use a single variable to calculate the running minimum of an array, but you can't use that algorithm with Mo's algorithm because there is no way to exclude previously seen values from the result.

The next part of Mo's algorithm is where stuff gets clever. At a high level, the goal is to re-order the queries so that a minimum amount of work has to happen to get from one query to the next.

For example, say you have the queries: [1, 30], [30, 50], and [20, 40].

If you solve for [1, 30], then transform to get the solution for [30, 50], and then transform to get the solution for [20, 40], then you're doing more work than you have to. You'll have to move the start pointer from 1 to 30 to 20. That's close to 40

moves. You'll have to move the end pointer from 30 to 50 to 40. That's close to 30 moves. Overall, that's nearly 70 moves.

However, if you solve for [1, 30], then transform to get the solution for [20, 40], and then transform to get the solution for [30, 50], then you do less work. You'll move the start pointer from 1 to 20 to 30. That's close to 30 moves. You'll move the end pointer from 30 to 40 to 50. That's close to 20 moves. Overall, that's nearly 50 moves, which is better than 70 moves.

So how do we sort the queries? Well, we break our range into
n
$\sqrt{}$
size chunks, where
n
is the number of values in our range. For now, let's pretend
n
$\sqrt{}$
is an even integer. That will make it easier to explain :)

Suppose our queries are all in the range [0, 99]. Then we break our range into chunks of size 10 (since 10 * 10 = 100). That is: [0, 9], [10, 19], [20, 29], [30, 39], [40, 49], [50, 59], [60, 69], [70, 79], [80, 89], [90, 99].

Let's call each chunk a "bucket." The first bucket is [0, 9], the second bucket is [10, 19], and so on.

The first thing we do is take each of our queries and assign it to a bucket according to its starting index.

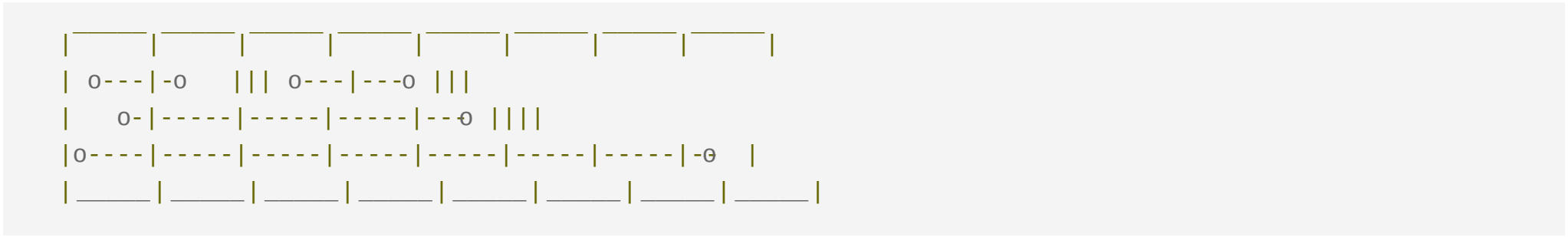
For example, if we had the query [12, 80], it would go in bucket 2 = [10, 19] since 12 is in the range [10, 19].

Next, we sort the queries in each bucket in ascending order of their end indices.

For example, suppose this is our range:

Now suppose this is our range broken up into
$\sqrt{n}$
sized chunks:

Now suppose this is our range with queries (shown as horizontal lines):



For this example, all of the queries except one lie in the first bucket. This is because all of those queries have a starting index in the first bucket.

All of the queries in the first bucket are sorted from top to bottom. (The end index of the top query comes before the end index of the second query, and the end index of the second query comes before the end index of the third query.)

The total ordering of the queries is thus:

> All of the queries from the first bucket in ascending order of their end indices.
>
> Followed by all of the queries from the second bucket in ascending order of their end indices.
>
> Followed by all of the queries from the third bucket in ascending order of their end indices.
>
> And so on...

It's actually really easy to sort the values in this way with a single comparator. Since that's an implementation detail, I

won't get into it. After reading my answer, the comparator on Mo's Algorithm - Codeforces will make more sense.

After sorting the queries, the essence of Mo's algorithm is to process the queries in sorted order.

You start by assuming you have the answer to the query [0, 0]. (In most cases, this answer is 0. For example, the sum of elements from index 0 to index 0 is 0.)

You then transform the result for the [0, 0] query to the result for the first query in your sorted list of queries. You do this by increasing the start index and the end index one value at a time until you get to the start index of the first query and the end index of the first query. You store the result for that query.

Then you transform to the second query in your sorted list of queries. You do this by altering the start index and end index of the first query until you get the start index and end index of the second query. You store the result for that query.

You continue in this way until all queries are processed.

Note: there are four possible cases that happen when transforming indices from one query to another:

> **Query 1's start index is less than query 2's start index:** continue increasing query 1's start index until it is equal to query 2's start index. In doing so, you *exclude* values from consideration.
>
> **Query 1's start index is greater than query 2's start index:** continue decreasing query 1's start index until it is equal to query 2's start index. In doing so, you *include* values for consideration.
>
> **Query 1's end index is less than query 2's end index:** continue increasing query 1's end index until it is equal to query 2's end index. In doing so, you *include* values for consideration.
>
> **Query 1's end index is greater than query 2's end index:** continue decreasing query 1's end index until it is

equal to query 2's end index. In doing so, you *exclude* values from consideration.

This transformation is simple with four while loops, one for each of the cases above. But again, that is implementation detail :)

Also note: the result of each of the above cases is that values are either included or excluded from consideration, one at a time. Again, this is why condition (3) from the list at the very top of this answer is important.

And that's Mo's algorithm!

The run-time is
$$O((n+q)\sqrt{k})$$
, where
n
is the number of values in your range,
q
is the number of queries, and
k
is the amount of time it takes to include/exclude a value from consideration. In the case of sums over a range, it takes constant time to add an integer and constant time to subtract an integer, so you're left with an
$$O((n+q)\sqrt{})$$
algorithm.

Let me give a very informal explanation as to why the run-time complexity is

$O((n+q)\sqrt{k})$
.

For any two queries in a particular bucket, the start index between them will change by no more than $\sqrt{n}$. If it could change more, then one of the queries is in the wrong bucket!

Further, the start index from the last query of one bucket to the start index of the first query of another bucket will change by no more than $2\sqrt{n}$.

Therefore, the start index between any two subsequent queries will change by no more than $O(\sqrt{n})$. Over q queries, that means we include/exclude a value from consideration $O(q\sqrt{n})$ times. That means it takes $O(q\sqrt{k})$ time to adjust all of the start indices.

Now, for each subsequent query in a particular bucket, the end index will continue increasing (or may stay the same). Across all queries in a single bucket, the end index can increase by a maximum of n. Since there are $\sqrt{n}$ buckets, that means we include/exclude a value from consideration $O(n\sqrt{})$ times. That means it takes $O(n\sqrt{k})$ time to adjust all of the end indices.

Overall, that means it takes $O((n+q)\sqrt{k})$ time to process all queries.

Pretty cool, right?

Please let me know if you have any questions at all, and I will try my best to help you understand :)

How do I solve SPOJ.com - Problem PATULJCI using MO's algorithm (query square root decomposition)?

7,447 Views

How do I solve this problem by using square root decomposition?

3,367 Views

How does the technique of sqrt N decomposition work and in what kind of problems is it useful?

10,787 Views

Can we modify the square root decomposition technique to the cube root decomposition? If not, why?

1,287 Views

What kind of problems can be solved most efficiently using square-root decomposition and not any other technique?

1,834 Views

From where top coders learn about square root decomposition algorithm?

659 Views

How do I decide which method/DS to use while querying on a range, square root decomposition, sparse table method, segment tree or binary indexed tree?

311 Views

How will you solve the K-Query using segment trees?

2,785 Views

How do we query in 2D BIT?

239 Views

What is the idea behind the D-query problem on SPOJ?

5,562 Views

What do "offline" queries in the context of an algorithmic problem mean?

2,852 Views

How do I solve range power sum query using treap?

707 Views

Is it necessary to use C++ for algorithmic coding?

1,876 Views

Which key causes the root to be on level 4 for the first time (refer question details)?

157 Views

How do I maintain a segment tree for range minimum query and updation?

963 Views

How do I solve the SPOJ problem of distance query?

711 Views

How do I answer the following queries in a tree?

1,887 Views

How do I implement a 2D binary indexed tree for range update and range query operations?

4,692 Views

Given a string of length n, I have to reverse it from index l to r for m queries. What should be the approach other than naive?

3,056 Views

How can we define our own query language?

1,493 Views

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy

BESbswy