Mo's Algorithm

I. Introduction

Being able to query a range of data is one of the most basic and useful tools in programming. Many problems require looking at a subset of data, not the whole dataset. For example, we may want to know the sum of a range, the number of distinct values in a range, or how often a certain value appears between two indices.

The naïve approach to query a range is to loop through the entire range to answer the query. This means the naïve approach has a runtime of O(Q*n) where n is length of range and Q is number of queries. This solution is too slow for very large inputs.

There are several algorithms and implementations that reduce the time complexity of range querying. Prefix Sums can answer range queries in O(1) and Fenwick Trees in O(log n), but they are limited to work only for specific operations.

Segment Trees handle associative operations on a range in O(log n) time given a constant time merge function, but require each node to store possibly large data structures to make the merge step possible.

Mo's Algorithm is an approach to range querying which excels Prefix Sum, Fenwick Trees, and Segment Trees in solving range queries with complex/multiple operations, and additionally can handle non-associative queries. Mo's algorithm is a special application of sqrt-decomposition which divides the whole data into blocks of size sqrt(n). Mo's algorithm just requires answering queries offline, so it can select the order queries are solved to maintain a data structure of the elements currently being queries. The algorithm then can answer queries by simply adding and removing individual elements from this data structure.

II. Overall Intuitive Approach

Mo's Algorithm for an array of length n takes a list of queries Q and orders their execution to most benefit from overlapping values in the queries. The array is divided into blocks of size sqrt(n), and each query [L, R] is sorted into a block based on its left index L. Then, queries in each block are sorted by lowest right index R. This organizes the queries in a way so that both the left and right indices do not change much from query to query. As we go through the ordered queries for each block, we keep track of which items are in each query and update this data structure for each successive query. We define the current range [current_left, current_right]. As we solve each query we can increment or decrement current_left and only increment current_right. When current_left is incremented, the index that left the range (old current_left) is removed. Similarly, when current_left is decremented, we add the new index (old current_left-1) into the data structure. When current_right is incremented, the index that joins the range (new current_right) is added to the data structure. We choose a data structure that has fast add and remove, and can calculate our specific query fast once it includes all items in the range. This means we can quickly update the data structure between queries and can quickly get the answer for each query.

Within each block, each query might have its left index move between the edges of the block, and the right index can progressively make its way to the end of the entire range for each block. A block size of

sqrt(n) is chosen to balance the time spent doing each of these operations, which can be seen in the runtime of the algorithm, which ends up being O((N+Q)sqrt(N)) for Q queries over an array of size N.

III. Implementation

The implementation has two main parts:

1. Ordering the queries
2. Executing queries

First, for ordering queries, a query can be defined as:

```
class Query:
    left: int
    right: int
    idx: int
```

The index idx is necessary to place back the queries in the original order after sorting them.

The block size can be defined as:

```
sqrt_n = math.ceil(math.sqrt(len(array)))
```

The queries are then sorted:

```
query_blocks = [[] for _ in range(sqrt_n)]
for i in range(num_queries):
    left, right = [int(x) for x in input().split(" ")]
    query_blocks[left//sqrt_n].append(Query(left,right,i))

for block in query_blocks:
     block.sort(key=lambda x: x.right)
```

The queries are first added to buckets based on the block of the left index, and then sorted per bucket according to the lowest right index.

Having ordered the queries, we come to the second part of the implementation, executing the queries:

```
for query in sorted_queries:
    while current_left > query.left:
```

```
        current_left-=1
        add(current_left)
    while current_right < query.right:
        current_right+=1
        add(current_right)
    while current_left < query.left:
        remove(current_left)
        current_left+=1
    while current_right > query.right:
        remove(current_right)
        current_right-=1
```

These four while loops adjust the current range and handle adding and removing elements. The add() and remove() functions can be implemented to solve various range operations.

The total runtime of Mo's Algorithm is $O((N + Q) \cdot \sqrt{N})$. This is based on the dominant term which is the range pointer movement. Updating current_left is $O(Q \cdot \sqrt{N})$ (each of Q queries can move between the edges of the $\sqrt{N}$ sized block). Updating current_right is $O(N \cdot \sqrt{N})$ (as each of the $\sqrt{N}$ blocks can move the right index to the end of the array). So, the total work is $O((N + Q) \cdot \sqrt{N})$. (The O(QlogQ) sort is dwarfed by the $Q\sqrt{N}$ term (as we assume Q and N are roughly equal)and so not needed).

IV. Summary of Programming Challenge

The programming challenge we developed asks students to analyze network requests, where each request is represented by the ID of the computer accessing the server. The student must provide functionality to answer two types of range queries. First, given a range [L, R] and a specific ID, return the frequency of that ID over the range, or the amount of times that ID shows up in the range. Second, given a range [L, R], return the number of unique IDs that appear at least K times over the range, where K is a given constant for all queries.

Our twist is that this programming challenge does not merely ask for a basic range query. We ask students to calculate two different complex operations on the moving range. In order to solve this problem both quickly and without an excessive use of space, Mo's algorithm is necessary. Further, students will see the power of Mo's Algorithm in an implementation that does not simply perform one trivial range operation.

Learning objectives

- Recognize when Mo's Algorithm is the best approach for the problem.
- Understand and implement Mo's Algorithm, specifically why queries are rearranged to reduce total work.
- Implement add() and remove() that can handle complex operations, highlighting the strength of Mo's Algorithm.

- Understand how Mo's Algorithm can handle multiple pieces of information in the same query.

V. Key Ideas for Solving Programming Challenge

Students will need to process all queries offline and sort them by block of left index and then by right index. This is a standard Mo's Algorithm implementation. The more difficult part is working with the uniqueness queries. For the frequency queries, the student will map IDs to its frequency in current range. For the uniqueness queries, students will keep track of how many IDs are currently at or above frequency K. In order to make sure these queries can be solved quickly, students can simply keep a count of all elements with a frequency at least K in the range, and track when the frequency of any one ID changes from K-1 to K or from K to K-1 and increase or decrease the count for each case respectively.

This is an interesting programming challenge that will thoroughly teach students the inner workings of Mo's Algorithm. The Mo's implementation here is quite standard, but significant understanding will be necessary to design the range updates and update the data structure properly at each step.

VI. Conclusion

Mo's Algorithm is an offline range querying solution that is helpful when queries are too complex to answer with Prefix Sum, Fenwick Trees, or Segment Trees. By sorting queries, Mo's Algorithm can reuse a significant amount of information between queries, only adding or removing certain values. This is a key efficiency of Mo's Algorithm, saving time by not recomputing values across queries.

The programming challenge shows a plausible application of Mo's Algorithm to a situation where range queries are complex, specifically in a real example of network request analysis. It asks students to answer what the frequency of a value is over a range as well as the number of unique elements over a threshold. This is an opportunity to understand how Mo's Algorithm sorts queries and also uses efficient add and remove operations.

To conclude, this project shows that Mo's Algorithm is a useful tool to remember and implement when solving various problems which require the use of complex range queries.