Mo's Algorithm

Executive Summary

Cole Frankenhoff, Josh Huddleston, Keshav Tarafdar, and Sebastian Condyles
CS 4501: Advanced Algorithms and Implementations

1 Introduction

In today's world, data science is one of the most important functions in which companies invest for their operations. With highly detailed and reliable data, companies can evaluate past customer behavior and predict future customer behavior. They can find trends and optimize their practices to make the business as efficient as possible. Part of a data scientist's role when examining their data is finding patterns that are not easily visible. There are often many different ways and many different places in the data to discover the patterns for which they are looking. Their supervisors may also want reports that include the answers to many questions regarding the data.

Efficiently testing different contiguous segments and time periods of the data is a significant part of what data scientists need to do every day, and this is where range queries come into play. Range queries are an integral part of real-world applications of computer science. They allow a user to pick an aggregate function, some examples being sum, mean, and mode, and the begin and end bounds with which to evaluate the data.

A few data structures and algorithms exist to compute range queries efficiently, but each of these has specific use cases, and they only work for a subset of all possible use cases, which is why multiple have been developed. Fenwick trees, for example, take an array of elements and compute prefix sums or other function outputs on contiguous portions of that array to perform queries in logarithmic time. Segment trees cover every function of a fenwick tree, and they operate using a divide and conquer-style tree with pre-computed range function outputs to use for calculating queries in logarithmic time. Fenwick trees are extremely efficient; they have better constant factors and space complexity than segment trees, but they only work for invertible functions.

In simple terms, invertibility is the property of a function being bidirectional, i.e. a user can calculate the output with the inputs and is also able to find an input again when given the output and a proper subset of the input(s). In other words, the operation is reversible. Sum is an invertible function because if a + b = c, one knows that a = c - b. Functions such as minimum and maximum are considered uninvertible because, when removing an element from a range (and that element is the minimum), and a user only has the current minimum, there is no way to find the new minimum with certainty without recomputing it in linear time.

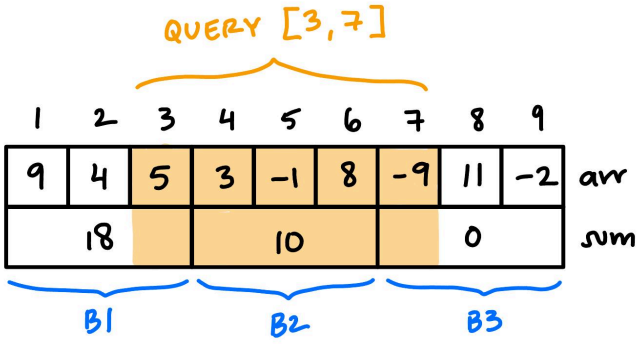
There are some functions that neither fenwick trees nor segment trees can handle due to single dimensional information, meaning they only store the single answer to a query, and nothing else. The mode function, for example, requires a separate data structure for a range query that stores the number of instances of each value. This type of

functionality is not natively supported by fenwick and segment trees. There are some preprocessing algorithms that help with mode, but are no better than linear time if only one query is performed. In cases where a user has a significant amount of queries (for any function, not just mode), manually calculating each query one-by-one can be slow. These issues necessitate a new algorithm that can handle a wide array of functions, supporting helper data structures and also solving multiple queries at once with a better runtime than calculating each query starting from scratch.

The target runtime of the new algorithm is $O(Q \log(Q) + (N + Q)\sqrt{(N)})$, with Q being the number of queries and N being the number of elements in the array on which to perform range queries. This runtime will make more sense in the next section.

2 Overall Approach

For the new algorithm, called Mo's Algorithm, it is useful to begin with an understanding of square root decomposition. It is similar to segment trees in that it starts with the entire array and breaks it into blocks of approximately the same size. However, instead of a divide-and-conquer style approach, dividing by 2 down to single elements, square root decomposition divides the array one time in $\sqrt{N}$ "blocks," or sections of the array. The "answer" of the block, the result of the function call, such as sum, on that block, is preprocessed and stored. The square root factor maximizes the product of the number of blocks and the number of elements per block. When processing a query, it is often necessary to combine the answers of whole blocks and partial blocks. A maximum of two partial blocks will be processed in a query - one on either side. In the example below, the middle block's pre-stored answer of 10 has to be combined with the single element values of 5 and -9 on either side.



With a square root factor, the number of operations in a query is minimized because there are two factors at play - O(number of blocks) and O(elements per block). To maximize efficiency, we have to minimize the sum of these two runtimes. This is done by finding where the terms are equal. With a fixed sum, the product is maximized when the terms are equal, which means the square root of N, in this case.

The square root decomposition algorithm allows for supplementary data structures for the purpose of supporting more complex operations. For example, for the mode function, the computer stores the state of a hashmap (key = element, value = number of instances of the element) of each block ahead of time. These results are combined with single values and other blocks during a regular query. For single queries, square root decomposition implements functions that fenwick trees and segment trees cannot in an algorithmically efficient manner.

Mo's Algorithm, the subject of this paper, builds on square root decomposition in one specific application: when there are multiple queries, and they are all known ahead of time. Building off of square root decomposition's "blocks", Mo's algorithm takes as input the array of elements as well as a list of queries. Based on the left and right bounds of each query, Mo's algorithm sorts the queries by the "block" that the left query is in, and then by the right query bound. The first query in the list is computed in O(N) time, but after that, the bounds are adjusted dynamically for the following queries element-by-element, and as the algorithm computes each query, it has a "living answer" (value of the function for the current range) that is changed dynamically. As elements are added and removed dynamically from either end of the range, the living answer is being altered along with it. Thus, the queries are sorted to minimize the differences in the bounds of successive queries and thus minimize operations in general. Mo's Algorithm is described in greater detail in the following section.

3 Implementation

Mo's algorithm has two parts:

- Mo's Procedure:
    - Sort queries.
    - Compute query results in sorted order by modifying the bounds of the query range.
    - Return the query results in their original order.
- Function Data Structure:
    - Represent the selected function.
    - Provide the *add*, *remove*, and *answer* methods.

Mo's procedure is always the same, but the data structure depends on the function being queried. In order to demonstrate an advantage of Mo's algorithm, we chose to implement the *mode* function, which doesn't work well with Fenwick or Segment Trees.

Mo's Procedure

This is python code for (most of) a comparator that implements the Mo query sorting scheme, which is to sort by the block of the query's left index, then to break ties with the right index.

```
    def sortWithBlockSize(q1, q2, blockSize):
        q1lb = q1[0] // blockSize
        q2lb = q2[0] // blockSize
        return q1lb - q2lb if q1lb != q2lb else q1[1] - q2[1]
```

This is python code for a class that implements the Mo's procedure, using the sorting method above. It assumes that the data object passed into the constructor has the methods init, add(idx), remove(idx), and answer(). The implementations for these for mode are in the next section. There are color matched descriptions for important lines below.

```
    class Mo:
        def __init__(self, blockSize, data):
            self.blockSize = blockSize
            self.data = data

        def query(self, queries):

            queriesWithIdx = [(queries[i][0], queries[i][1], i) for
i in range(len(queries))]

            sortedQueries = sorted(queriesWithIdx, key =
functools.cmp_to_key(lambda r1, r2 : sortWithBlockSize(r1, r2,
self.blockSize)))

            results = list(range(len(queries)))

            self.data.init()

            l = 0
            r = -1
            for q in sortedQueries:
                while q[0] < l:
                    l -= 1
                    self.data.add(l)
                while r < q[1]:
                    r += 1
                    self.data.add(r)
                while l < q[0]:
                    self.data.remove(l)
                    l += 1
                while q[1] < r:
                    self.data.remove(r)
                    r -= 1
                results[q[2]] = self.data.answer()
```

```
            return results
```

This maps each query into a tuple that also contains the original position index of the query, which we will later use to write the answer to the query to the correct position.

This line sorts the queries using the sorting method above and with the block size set in the constructor.

These while loops make the appropriate adjustments to the bounds of the current range (l, r) represented by the data structure to match the range of the current query.

Mode Data Structure

We use a map to track the frequency of each number in the current range. We also track which numbers have each frequency using an array of frequency "bucket" sets. When the current range changes, the frequency of the number removed/added is updated, and that number is moved left/right to the correct bucket. The rightmost non empty bucket contains the mode, and because we update buckets only by moving numbers left/right one at a time, updating the mode is simple (and constant time). Here is the python code, with color matched descriptions for important lines.

```
     class ModeData:
        def __init__(self, array):
            self.array = array

        def init(self):
            self.frequencies = {} # maps numbers to current
frequencies
            self.buckets = [None for i in range(len(self.array))]
            self.modeFreq = 0 # stores the frequency of the current
mode

        def addToBucket(self, freq, item):
            if freq - 1 < 0:
                return
            if not self.buckets[freq - 1]:
                self.buckets[freq - 1] = set()
            self.buckets[freq - 1].add(item)

        def removeFromBucket(self, freq, item):
            if freq - 1 < 0:
                return
            self.buckets[freq - 1].remove(item)


        def add(self, idx):
            val = self.array[idx]
```

```
            if not val in self.frequencies:
                self.frequencies[val] = 0
            self.removeFromBucket(self.frequencies[val], val)
            self.frequencies[val] += 1
            self.addToBucket(self.frequencies[val], val)
            if self.modeFreq < self.frequencies[val]:
                self.modeFreq = self.frequencies[val]

        def remove(self, idx):
            val = self.array[idx]
            self.removeFromBucket(self.frequencies[val], val)
            if self.frequencies[val] == self.modeFreq and not
self.buckets[self.frequencies[val] - 1]:
                self.modeFreq -= 1
            self.frequencies[val] -= 1
            self.addToBucket(self.frequencies[val], val)

        def answer(self):
            for val in self.buckets[self.modeFreq - 1]:
                return (val, self.modeFreq)
```

This line initializes an array of frequency buckets, index 0 represents frequency 1, so numbers with "zero" frequency are not stored in any buckets.

These are convenience methods for handling frequencies being offset by 1 from bucket indices. The add method handles creating sets and both methods ignore "zero" frequency calls.

These lines update the mode when the added/removed element affects it.

4 Summary of Programming Challenge

Our programming challenge asks students to write a program that, given an unordered log of M chat messages of the form (timestamp, username), can answer Q offline queries of the form (l, r), asking for the number of distinct users who have sent a message within that time range.

Although at a glance this might seem approachable via a sliding window or prefix-hash technique, there are two main characteristics that make it perfect for Mo's Algorithm: the queries arrive all at once (meaning they are *offline*), and the input length of queries can be as large as 10^4 (with up to $5*10^5$ total messages). Additionally, the brute force solution (treating each query independently) would result in a worst-case runtime of $\theta(MQ)$, where M is the total number of messages sent and Q is the total number of queries.

There are two main "twists" that students must adapt Mo's Algorithm to in order to solve this problem. The first is that although Mo's Algorithm works on contiguous indices, the queries given in this case arrive as arbitrary clock times/timestamps. This means that all messages must first be sorted, and then each query should be binary searched in order to translate the "time interval" to an "index interval" that we can slide over. Additionally,

students need to account for queries whose ranges contain *no* messages. The second is that in order to maintain the number of distinct users inside the current time window, students must design an invertible state such that both adding and removing a message must update the answer correctly in O(1). Although this can be easily handled via a frequency map from usernames to counts, students must also be careful to handle edge cases, such as when a user's count drops to zero.

By the time students implement a successful solution, they will have engaged with the full "lifecycle" of Mo's Algorithm: query preprocessing, intelligent reordering based on block decomposition, and efficient pointer sweeping to maintain distinct user count. Finally, accounting for tricky edge cases will also improve their understanding of the niche where Mo's Algorithm excels.

5 Programming Challenge Key Ideas

The problem is about tracking user activity across time. We're given a bunch of messages (timestamp + username) and a bunch of queries (start and end times), and for each query, we need to quickly figure out how many *unique* users sent a message during that time window. There are a potentially massive amount of messages and queries, so we need to handle them efficiently. Mo's Algorithm is a perfect fit for this problem, because it deals with a large number of offline queries over a range. The merge operation for a segment tree would be impractically complex.

Here's a brief overview of the algorithm:

- Load all given messages into a list.
- Sort the messages by timestamp and assign them array indices (position 0, 1, 2, etc.). Now timestamps are basically just indices in this array.
- For each query (start time, end time), binary search to find the range of indices that match — so every query becomes a simple [left, right] over the array.
- Set up Mo's Algorithm:
    - Sort all the queries by block (block size ≈ sqrt(total messages)) and then by right endpoint.
    - Maintain a sliding window [current_left, current_right] over the messages.
    - As the window moves:
        - When you add a message into the window, track the user — if it's their first appearance, bump up the unique user count.
        - When you remove a message from the window, track that too — if it's their last appearance, drop the unique user count.
- For each query, after adjusting the window to match its range, just output the current unique user count.

The total time complexity of this solution works out to about **$O((M + Q) \times \sqrt{M})$.**

6 Conclusion

There are some functions, such as mode and count distinct elements, where merge operations are simply too expensive to make a Fenwick Tree implementation practical. This is where Mo's Algorithm comes in, providing potentially huge optimizations for range query operations when all queries are offline, or provided altogether in advance. Mo's Algorithm uses the approach of preprocessing and sorting queries by block indices, all while maintaining a dynamic answer state. This allows it to achieve a time complexity of $O(Q \log(Q) + (N + Q)\sqrt{(N)})$, making it especially effective when there are a large number of queries.

The practical applications of this algorithm extend directly to data science. Analysts regularly evaluate multiple range queries across different time periods to identify patterns and trends, and Mo's Algorithm provides an efficient solution for these scenarios. As demonstrated in our programming challenge, it also handles complex tasks like counting distinct users within time windows effectively. It has also been seen in some competitive programming competitions/problems such as Spoj's DQUERY over the years.

The key insight of Mo's algorithm is to minimize work between consecutive queries. Our implementation proves this by sorting queries by block and optimizing transitions between ranges. For the challenging mode function, we developed a constant-time bucket-based data structure that dynamically tracks frequencies as elements enter and exit the active range. In our programming challenge, we successfully adapted the algorithm to count distinct users within time windows, translating timestamps to array indices and maintaining an invertible user frequency state. The results confirm Mo's Algorithm achieves $O(Q \log(Q) + (N + Q)\sqrt{(N)})$ runtime—significantly outperforming naive approaches for large query sets while handling functions that traditional range query structures struggle with.