Bioinformatics Homework 1.1

Finding Patterns Forming Clumps in a String

Problem Statement: You are given a string *Genome* and integers *k*, *L*, and *t*. Find all strings of length k *(k-mers) t*hat occur at least *t* times completely within a window of L character for all windows of *L* characters in the *Genome*. Return all of these strings.

Algorithm Description:

```
    FindClumps(Text, k, L, t)
        Patterns <- a set of strings of size 0
        n <- |Text|
        for every integer i between 0 and n – L
            Window <- Text(i, L)
// This holds the frequency of each k-mer in a map
            freqMap <- FrequencyTable(Window, k)
            for every key s in freqMap
                if freqMap[s] >= t
                    append s to Patterns
        remove duplicates from Patterns
        return patterns
```

I used a sliding window of length L and kept track of the k-mers that I saw in a set. As I move the sliding window forward, I add the k-mer at the front of the window into the set and remove the one at the back of the window from the set. If the k-mer that I just added contains more than t occurrences in the set, then I add it to the hash-set of solutions.

Time Analysis: This algorithm uses a for loop that slides an L-length window across the Genome. For each window, I do hash-map/set operations, each of which run in O(1). For those hash/set operations, I take a substring of *Genome*, which takes O(k) time. At the end, I transfer the set into a vector; because there cannot be more than n-k+1 unique strings, the transfer runs in (n-k+1) * k.

The whole algorithm runs in 4*k*(n-k+1) + k*(n-k+1) = *O(n*k)* where n is the *Genome* size. Because we are potentially storing (n-k+1) strings of length k into a set, the space complexity is also O(n*k)

Implementation:

Submission Number: #**1071642**

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i, a, b) for (ll i = a; i < (ll)b; ++i)
typedef long long ll;

vector <string> FindClumps(string Genome, int k, int L, int t)
{
    unordered_set<string> ans;
    unordered_map<string, ll> mp;

    // Sliding window for L
    rep (i, 0, Genome.size()-k+1)
    {
        // updating which substrings we are seeing based on our window
        if (i+k-1 >= L) mp[Genome.substr(i-L+k-1, k)]--;
        mp[Genome.substr(i, k)]++;

        // Inserting the front most k-mer found if possible
        if (mp[Genome.substr(i, k)] >= t)
            ans.insert(Genome.substr(i, k));
    }

    vector<string> res;
    for (auto x : ans) res.push_back(x);

    return res;
}
```

Discussion:

I think that the implementation of this algorithm is generally efficient. The main constraint on the problem is that we are looking at every single substring in the *Genome*. If the problem was changed to say return the positions of the strings that are a part of a clump for example, then the algorithm could run much faster. With some further changes such as string hashing, the hash-maps/sets would run much faster, and we could remove a factor of k from the entire algorithm making the runtime O(n).