Find Patterns Forming Clumps in a String

Problem Statement (1):

Rosalind describes the problem as such: “Given integers L and t, a string Pattern forms an (L, t)-clump inside a (larger) string Genome if there is an interval of Genome of length L in which Pattern appears at least t times”.

Input(s): Our inputs are **k** (the *integer* length of the clump string), **L** (the *integer* length of the segment of the genome we are looking at for clumps), **t** (the minimum number of times the clump must occur in the genome segment,*integer*) and **Genome** (the input *string* we segment and parse through for our clumps).

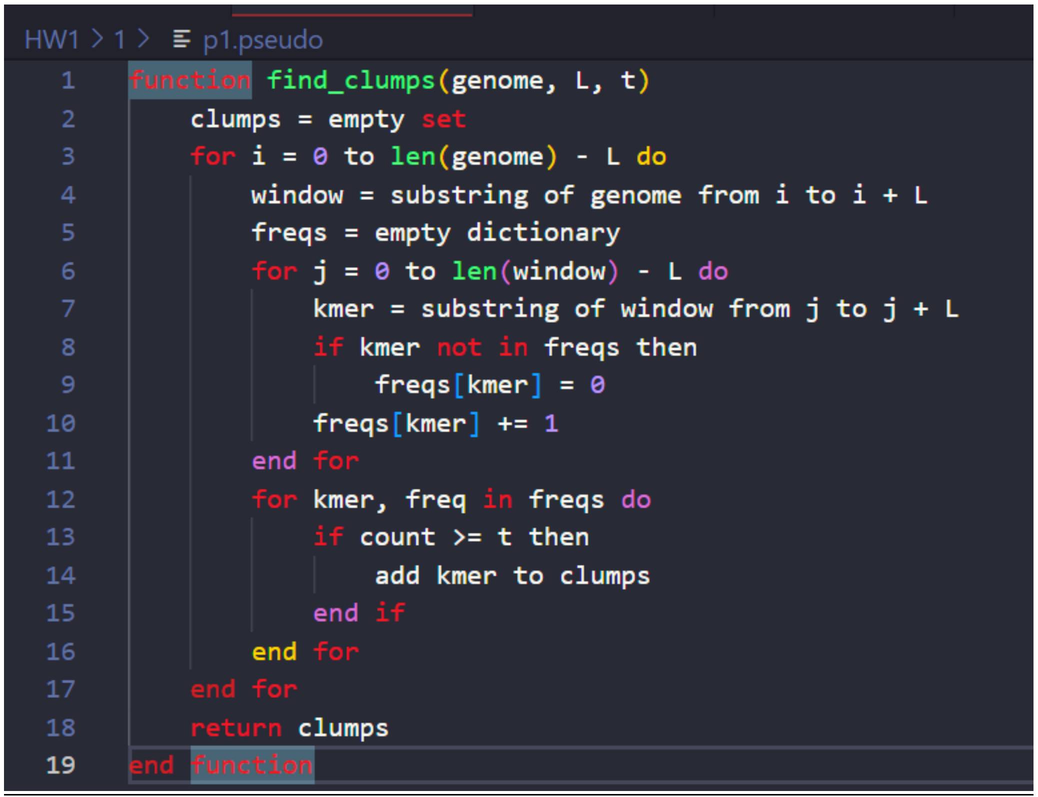
Output: We should return any clumps (strings) that are of length **L** that occur at least **t** times in the given **Genome.** The datasets assume that there are no cases where there are no clumps, but perhaps we could implement a message or error code if no clumps are found.

Function(s): Our main function, ***find_clumps()***, will drive our algorithm. Outside of the standard lib Python functions, we won’t be using any auxiliary/utility functions for this solution. Our data structure(s) will be a hashmap (dictionary in python) to track the frequencies of found clumps, and a set to track **unique** clumps against **Genome**.

Constraints:

- **L** cannot exceed the length of the **Genome** string
  k * **t** cannot exceed the length of the **Genome** string
- The datasets assume that there will be at least one (L,t) string, but there are no output specifications if no clumps are found
- **L** should be >= 1, otherwise there would not be clumps but single characters
- Some arbitrary constraints include input validation, edge cases, and memory

Algorithm description/Pseudocode (2.1):



HW1 > 1 > p1.pseudo

```
function find_clumps(genome, L, t)
    clumps = empty set
    for i = 0 to len(genome) - L do
        window = substring of genome from i to i + L
        freqs = empty dictionary
        for j = 0 to len(window) - L do
            kmer = substring of window from j to j + L
            if kmer not in freqs then
                freqs[kmer] = 0
            freqs[kmer] += 1
        end for
        for kmer, freq in freqs do
            if count >= t then
                add kmer to clumps
            end if
        end for
    end for
    return clumps
end function
```

NOTE: This is pseudo-code, I just have syntax highlighting on VS Code to help me write it 😉

Explanation (2.2)

We used the 'sliding window' approach for this problem.

Once in the substring of length L, we iterate over the string piece by piece by *sliding* our substring over the length, making sure we get all possible k-mers in the sequence.

We track the frequency of each k-mer using a dictionary (freqs), and we iterate over the dictionary checking for frequencies that are greater than or equal to ***t*,** adding those to our **clumps** set, building our end result.

Time analysis (3):

This problem has a rough time complexity of **(O(n-L(L*k))),** which is polynomial.

Our initial outer loop iterates over the string n – L + 1 times, where n is the length of the genome string and L is the segment of the genome we are parsing through, giving us **O(n-L)** for the general time complexity of the outer loop.

In our nested loop, we iterate over the window created in the outer loop, where we iterate k times, where k is the provided length of the k-mer(s) we are looking for. So, the nested loop's time complexity is **O(k)**

In the final outer loop we are iterating for each windows, which goes through all possible k-mers within the window, giving us **O(L – k)**

We multiply the individual time complexities of the loops and: O((n-L) * (L-k) * k)) = (O(n-L(L*k))).

Discussion (5):

We used the 'sliding window' approach for this problem.

We have a polynomial time complexity for this problem which is not ideal, and we can probably use some more advanced data structures to reduce the runtime at the cost of memory.

Since the size of the window is fixed, we cannot find clumps of varying lengths in a single run of our program.

After a bit of googling, we could use suffix trees/arrays to efficiently search for repeated patterns, but we would need to also implement a pattern finding algorithm as well for this approach.

For more complex patterns/sequences we could employ machine learning but that seems to be way beyond the scope of this course.