Yo Ms. Wolf,

This is the README for the random writer, as you probably have guessed. This is just a few notes to myself and to you to better help you understand my code. Thanks for taking the time to help me chill the f*ck out also lol. Anyway, I have dotted a few comments throughout my code, but if you want a better explanation it might be in here.

Approach: As we talked about before, I proved that the time and space complexity of the simple approach you suggested is actually the most efficient implementation of the random writer. So I am going with that, as efficiency matters more than coolness (sadly). However, I did make a slight improvement, that is I determine when we are guaranteed to have a seed repeat itself, and then I use a hashMap to store <seed, characters that follow>. This allows me to skip the search, and it decreases the time complexity for a small increase in space complexity. However, as the difference between length and source.length increases, this approach becomes more efficient.

- I read in my arguments, check them, then I pass them on to markov.cc.
- I determine if I should use hashMap.
- Create seed, print seed, make sure seed has chars after it.
- If there are no chars after seed, I generate new seed.
- Otherwise, I get a char randomly, print it, and repeat the process

File Structure:

- randomwriter.cc checks if the arguments are valid, then calls markov.cc
- markov.cc then conditionally calls hashMap, and it then simply executes the approach

HashMap:

- Destructor: Since C++ calls the destructor right after we create/initialize an object, if our destructor was non-empty, it would destroy our hashMap before we get to use it.
    - Thus we create the *destroy()* method, which we can call at our discretion to delete the hashMap.

- Constructor: Our hashMap is a pointer to an array of pointers, and we do this so that we can dynamically allocate memory for our hashMap so we're not wasting memory.

- Hash Function: For our hash function, I created a polynomial hash which uses primes as the variable of the polynomial and as the *modulus*. The reason we use a large prime for the variable of the polynomial is because if we didn't do that, there would be many common factors that the variable would share with the values of the chars in our string. This means there's a lot of room for collision. In a similar vein, if we hadn't used a large prime for our *modulus*, there would be a lot of values that map to 0 mod *modulus*. This is because the *modulus* would have a decent number of factors, meaning that there would be quite a few different combinations to get to the same value mod *modulus*.

- Setter and Getter: For my add function, we already talked about how I just insert the element into the circular dll. This is because each entry in the hashMap is a node, so I just link up new entries whose keys map to the same index. This prevents collisions. My get function is also pretty standard; I just find the index of the key, then iterate through the circular dll until I find a node with the same key. However, there is a small subtlety here: in theory, the characters mapped to a given key could be any string of length < source.length() - k + 2. So we need to return a string that does not comply with these restrictions in order to signal that there is no entry for the given key, so we return a string of length source.length() - k + 2. This way we can just check the length and know if the entry exists in the hashMap or not.

Markov:

- This class performs the meat of my operations.
- Pre-Process/Search: These methods are just my KMP search implementation from the pseudocode I wrote up and presented. It's fairly standard.
- mas: This method is the pre-processing for the two-way string search algorithm.
    - *Factorization* of a string is just a way to break a string into some number of substrings who can be concatenated to get the original string.
    - *local period* of a string is the shortest local period.
    - *Critical factorization* of a string *s* is a factorization of *s* such that the *local period* of the *factorization* is equal to the period of *s*.
    - What we do in this pre-processing is we find the *critical factorization* of *source*, and we also determine the *period* of this *critical factorization*.
    - This is very similar to what we do in Boyer-Moore, and you will also see this later when we are able to skip certain characters because of the *period*.
    - This is also fairly similar to what we do in KMP, as we are looking for repetitions of characters within our *seed*.
    - Anyway, we find this *critical factorization* by computing the maximal suffix for both left > right and left < right, then comparing the indices of our solutions and seeing whichever one is larger. This is because we want the left *factorization* to be minimal.
- tws: This method is the search itself.

    - This algorithm is actually fairly simple once you understand Boyer-Moore and KMP, as it is essentially an amalgamation of the two.
    - What we do is we start at the point of our *critical factorization*, and we essentially just go through the *source* until we find a matching character.
    - Once we do find this matching character, we ensure that the rest of the seed matches with the other characters connected to the matching character in *source*.
    - If they do, we've found a solution, and we move on.
    - If they don't, we move on.
    - Now comes the cool part.
    - Because we know the period of our *critical factorization*, we can skip a number of characters in *source* that's equal to a multiple of the period + 1, because we already know that those characters will match.
    - This is how it's similar to Boyer-Moore.
    - We essentially just continue doing this until we reach the end.
    - This has time complexity O(m + n) and space complexity O(1), so it is more space efficient than KMP.
    - We also just have to address this algorithm for 2 cases: the left *factorization* is larger than the right, and the right *factorization* is larger than the left.
    - Quick Note: This algorithm was super fun for me to implement on my own because there was so much pure math involved. It really challenged me and got me thinking harder about string searching. Thanks for that!
- Iteration: This method is just 1 iteration of random writing.
- Execute: This method just repeatedly calls iteration to perform the random writing.

Note on perfect hash functions: In order to create a perfect hash function, we would either have to get incredibly lucky, or we would have to spend polynomial time to determine a parameter that would allow us to map a set of n integers to a smaller range of size n reducing the strings to unique integers isn't hard: just take the bits of the string XOR them with 11111....11 and then just compute that binary number. Since each string has a unique bitwise representation and each binary number has a unique binary representation, XOR'ing with 1111...111 would reduce the size of the int and the reduced int would be unique for each string. However, if we were to find a polynomial-time method of constructing this perfect hash, then we would revolutionize modern cryptography overnight lol.