6 Pattern matching with mismatches

Problem Introduction

A natural generalization of the pattern matching problem is the following: find all text locations where distance from pattern is sufficiently small. This problems has applications in text searching (where mismatches correspond to typos) and bioinformatics (where mismatches correspond to mutations).

Problem Description

Task. For an integer parameter k and two strings $t = t_0t_1 \cdots t_{m-1}$ and $p = p_0p_1 \cdots p_{n-1}$, we say that p occurs in t at position i with at most k mismatches if the strings p and $t[i : i + p) = t_it_{i+1} \cdots t_{i+n-1}$ differ in at most k positions.

Input Format. Every line of the input contains an integer k and two strings t and p consisting of lower case Latin letters.

Constraints. $0 \le k \le 5$, $1 \le |t| \le 200\,000$, $1 \le |p| \le \min\{|t|, 100\,000\}$. The total length of all t's does not exceed 200 000, the total length of all p's does not exceed 100 000.

Output Format. For each triple (k, t, p), find all positions $0 \le i_1 < i_2 < \cdots < i_l < |t|$ where p occurs in t with at most k mismatches. Output l and $i_1, i_2, \ldots, i_l$.

Time Limits. `C`: 2 sec, `C++`: 2 sec, `Java`: 5 sec, `Python`: 40 sec. `C#`: 3 sec, `Haskell`: 4 sec, `JavaScript`: 10 sec, `Ruby`: 10 sec, `Scala`: 10 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
0 ababab baaa
1 ababab baaa
1 xabcabc ccc
2 xabcabc ccc
3 aaa xxx
```

Output:

```
0
1 1
0
4 1 2 3 4
1 0
```

Explanation:
For the first triple, there are no exact matches. For the second triple, `baaa` has distance one from the pattern. For the third triple, there are no occurrences with at most one mismatch. For the fourth triple, any (length three) substring of p containing at least one `c` has distance at most two from t. For the fifth triple, t and p differ in three positions.

What to Do

Start by computing hash values of prefixes of t and p and their partial sums. This allows you to compare any two substrings of t and p in expected constant time. For each candidate position i, perform k steps of the form "find the next mismatch". Each such mismatch can be found using binary search. Overall, this gives an algorithm running in time $O(nk \log n)$.