TJSCT - String Algorithms

Tarushii Goel and Grace Huang

April 2021

1 Hashing

1.1 Problem

USACO 2017 US Open Contest, Gold
Problem 1. Bovine Genomics

Farmer John owns N cows with spots and N cows without spots. Having just completed a course in bovine genetics, he is convinced that the spots on his cows are caused by mutations in the bovine genome. At great expense, Farmer John sequences the genomes of his cows. Each genome is a string of length M built from the four characters A, C, G, and T. When he lines up the genomes of his cows, he gets a table like the following, shown here for N=3 and M=8:

Spotty Cow 1: A A T C C C A T
Spotty Cow 2: A C T T G C A A
Spotty Cow 3: G G T C G C A A

Plain Cow 1 : A C T C C C A G
Plain Cow 2 : A C T C G C A T
Plain Cow 3 : A C T T C C A T

Looking carefully at this table, he surmises that the sequence from position 2 through position 5 is sufficient to explain spottiness. That is, by looking at the characters in just these these positions (that is, positions 2...5), Farmer John can predict which of his cows are spotty and which are not. For example, if he sees the characters GTCG in these locations, he knows the cow must be spotty.

Please help FJ find the length of the shortest sequence of positions that can explain spottiness.

INPUT FORMAT:
The first line of input contains N [1,500] and M [3,500]. The next N lines each contain a string of M characters; these describe the genomes of the spotty

cows. The final N lines describe the genomes of the plain cows. No spotty cow has the same exact genome as a plain cow.

OUTPUT FORMAT:
Please print the length of the shortest sequence of positions that is sufficient to explain spottiness. A sequence of positions explains spottiness if the spottiness trait can be predicted with perfect accuracy among Farmer John's population of cows by looking at just those locations in the genome.

SAMPLE INPUT:
3 8
AATCCCAT
ACTTGCAA
GGTCGCAA
ACTCCCAG
ACTCGCAT
ACTTCCAT

SAMPLE OUTPUT:
4

1.2 Naive Solution

We can approach this problem in a brute-force manner by guess-and-check for the minimum substring length starting from 1 to M. We can start by getting all substrings of length 1, then for each substring, separate the spotted cows and non-spotted cows into two sets based on that substring and determine if they are disjoint. If yes, that means a 1-character substring is enough to distinguish between spotted and non-spotted cows. If there are common elements (non-disjoint), then we move on to substrings of length 2, so on so forth.

Eventually, we will run out of time with this naive approach due to the mass number of strings we have to compare and the time-consuming method of comparing each pair of strings character by character.

1.3 Algorithm

1.3.1 Polynomial Rolling Hash Function

To produce a solution that achieves full credit on this question, we can consider ways to speed up our naive solution. It is hard to reduce the number of strings we need to compare, but what if we try to shorten the time it takes to compare two strings? Thus, we introduce today's algorithm: String Hashing.

String hashing is basically turning a string into a hash value, which is a number, so that when comparing two strings, the complexity is O(1) for com-

paring the hash values instead of O(n) for n characters that we have to compare between the Strings.

The hash function most commonly used for this conversion is called a **polynomial rolling hash function**.

$$hash(s) = (s[0] + s[1] * p + s[2] * p^2 + ... + s[n-1] * p^{n-1}) mod M$$

$$= \sum_{i=0}^{n-1} (s[i] * p^i) mod M$$

Interpreting this formula:
s is our string
i is the current index, from 0 to n-1
p is a constant of our choice
M is another constant of our choice

1.3.2 Collision

The most important feature of the polynomial rolling hash function is this:

When two strings have the SAME content, they must generate the SAME hash value.

Notice that the contrapositive of this statement is automatically true:

When the hash values are DIFFERENT, they the strings are necessarily DIFFERENT.

The most important fact is that the converse of this statement isn't necessarily true, meaning that:

When the hash values are the SAME, the strings are NOT NECESSARILY THE SAME.

This statement explains the occurrence of collisions that we must deal with: two different strings generating the same hash value, or colliding. Our solution to this problem, or a common approach to minimize the occurrence of a collision so that this error doesn't affect the accuracy of our algorithm in practice, is to choose the values of p and M carefully.

Conventionally, we pick the value of p to be a prime number roughly equal to the number of characters in the input alphabet. For example, p = 31 is a common choice if the input contains only lowercase characters.

To pick the value of M, we have to keep in mind that:

$$prob(colliding) \approx \frac{1}{m}$$

Therefore, it is common practice to set M:

$$M \approx 10^9$$

Thus, after the conversion that is O(n), we will be able to compare strings in O(1) time instead of O(n), successfully improving our original naive algorithm so that it runs in time.

1.4 Solution Code

```
//By Tarushii Goel
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const int MXN = 510;
const int P = 31;
const int MOD = 1e9+7;
int N, M;
int cows[2*MXN][MXN];
int pows[MXN];

bool check(int a, int b) {
   set<int> strs;
   for (int i = 0; i < N; i++){
      strs.insert((cows[i][b] - cows[i][a-1]+MOD)%MOD);
   }
   bool works = true;
   for (int i = N; i < 2*N; i++){
      if (strs.count((cows[i][b] - cows[i][a-1]+MOD)%MOD)) {
         works = false; break;
      }
   }
   return works;
}

int main(){
   ifstream cin ("cownomics.in");
   ofstream cout ("cownomics.out");
   cin >> N >> M;
   pows[0] = 1;
   for (int i = 1; i < M; i++){
      pows[i] = ((ll)pows[i-1]*P)%MOD;
   }
```

```
   map<char, int> mp;
   mp['A'] = 1; mp['G'] = 2; mp['T'] = 3, mp['C'] = 4;
   for (int i = 0; i < 2*N; i++){
      string s; cin >> s;
      for (int j = 1; j <= M; j++){
         cows[i][j] = (cows[i][j-1] + ((ll)mp[s[j-1]]*pows[j-1]))%MOD;
      }
   }
   int a = 1; int b = 1; int best = M;
   while (b <=M){
      if (check(a, b)){
         best = min(best, b-a+1);
         a++; b = max(a, b);
      }
      else b++;
   }
   cout << best;
}
```

1.5 Side Note

As some of you might have noticed, this problem can also be solved efficiently by binary searching for the appropriate substring length. However, as binary searching is not our primary topic of discussion today, you may explore on your own, and we won't go into detail about it in this lecture.

1.6 Additional Practice

scroll to bottom for practice problems

`https://cp-algorithms.com/string/string-hashing.html`
`https://usaco.guide/gold/string-hashing?lang=cpp`

2 Knuth-Morris-Pratt

2.1 Problem: Censoring

Farmer John has purchased a subscription to Good Hooveskeeping magazine for his cows, so they have plenty of material to read while waiting around in the barn during milking sessions. Unfortunately, the latest issue contains a rather inappropriate article on how to cook the perfect steak, which FJ would rather his cows not see (clearly, the magazine is in need of better editorial oversight).

FJ has taken all of the text from the magazine to create the string S of length at most 10^6 characters. From this, he would like to remove occurrences of a substring T to censor the inappropriate content. To do this, Farmer John finds the *first* occurrence of T in S and deletes it. He then repeats the process again, deleting the first occurrence of T again, continuing until there are no

more occurrences of T in S. Note that the deletion of one occurrence might create a new occurrence of T that didn't exist before.

Please help FJ determine the final contents of S after censoring is complete.

INPUT FORMAT:

The first line will contain S. The second line will contain T. The length of T will be at most that of S, and all characters of S and T will be lower-case alphabet characters (in the range a..z).

OUTPUT FORMAT:

The string S after all deletions are complete. It is guaranteed that S will not become empty during the deletion process.

SAMPLE INPUT:

whatthemomooofun

moo

SAMPLE OUTPUT:

whatthefun

2.2 Naive Solution

One solution involves making one sweep through S. As you sweep through S, you can add the characters of S to a stack. Each time you add a character to the stack, you would check if the last characters equal T, and if so, remove them. While this works, you have to an $O(len(T))$ check $len(S)$ times, resulting inan overall time complexity of $O(len(T) * len(S))$, which is too slow.

Although this algorithm is too slow, it provides important motivation for KMP. We can observe that removing the letters is at most $O(len(S))$ and sweeping is at most $O(len(S))$, so if we were able to improve the time complexity of our check, that would be sufficient optimixation to make this passing.

2.3 Algorithm

2.3.1 Longest Prefix Suffix Problem

KMP is designed to solve the longest-prefix-suffix (LPS) problem. This is just the longest proper prefix which is also a suffix. For example, let's look at the string S = "aabbcaabaaac." Let's define $lps[i]$ = length of the LPS of the substring $S[0..i]$. Then our lps array would be $lps = [0, 1, 0, 0, 0, 1, 2, 3, 1, 2, 2, 0]$. Note that it is a **proper** prefix, which means it can't be the whole string (if it was the whole string that wouldn't be useful, anyways).

2.3.2 Finding the LPS array for a string

We can find this lps array in a two pointers style approach. We keep one pointer at the end of the suffix and one points at the end to the prefix. If the next letter in the suffix part and prefix part are the same, we can increment both pointers by 1. If they are not the same, we know we have to move the prefix pointer backwards until the prefix part in equal to the suffix part. The trick here is that

we can use our lps array to know how far to move the prefix pointer back. Let the index of our prefix pointer be i. Then, we would want $i = lps[i]$.

2.4 Full Solution

```
//By Tarushii Goel
#include <iostream>
#include <string>
#include <vector>
#include <utility>
using namespace std;

int lps[1000000];
int in[1000000]; //amount of T in S at index
vector<pair<char, int>> ans;
int main(){
   string S, T;
   cin >> S >> T;
   //preprocessing - find lps for T
   int i = 2; //current index
   int j = 0; //potential endpoint of prefix-suffix
   while (i < T.length()){
      if (T[j] == T[i-1]) {
         lps[i] = j + 1;
         i++;
         j++;
      }
      else if (j > 0) j = lps[j];
      else {
         lps[i] = 0;
         i++;
      }
   }
   //KMP
   i = 0; //potential endpoint of prefix of T
   j = 0; //index in S
   while (j < S.length()){
      if(T[i] == S[j]){ //found the longest prefix for j
         in[j] = i+1;
         ans.push_back(make_pair(S[j], j));
         if (in[j]==T.length()) { //found instance of T in S
            for (int a = 0; a < T.length(); a++){
               ans.pop_back();
            }
            i = in[ans.back().second];
         }
         else {
            i++;
         }
```

```
            j++;
        }
        else if (i > 0) i = lps[i]; //go down the chain to the next
            endpoint
        else { in[j] = 0; ans.push_back(make_pair(S[j], j)); j++;}//there
            is no match
        //cout << i << ' ' << j << endl;
    }
    string s = "";
    for (pair<char, int> c : ans){
        s+=c.first;
    }
    cout << s;
}
```

2.5 Additional Practice

- `https://cp-algorithms.com/string/prefix-function.html`: scroll to the bottom for problems