USACO FEB15 Problem 'censor' Analysis

by Mark Gordon

This problem asks us to repeatedly delete the first occurrence of any string T_i from a larger string S until no string T_i appears in S.

Unfortunately, repeatedly searching for the first T_i that matches S and deleting it is far too slow; it can be as slow as $O(S^3)$ which obviously won't do.

A more reasoned approach is to build the result string, R, one character at a time. Whenever the end of R matches one of T_i we should delete it from R. Since this deletion is at the end of R this is just a simple $O(1)$ resize operation which is efficient.

Now the main problem is how to determine if the end of R matches one of T_i efficiently. I will discuss two ways of accomplishing this.

Hashing (Rabin Miller)

Hashing is a powerful tool in string based problems. However, it is not entirely straightforward to use hashing in this situation. Since the search strings each have different lengths it does not appear that we can check if any of T_i match the end of R efficiently. However, at most $O(S\sqrt{})$ different search string lengths (446 is the maximum) can appear in the input (the worse case is strings of length 1, 2, 3, ...).

Therefore we can attempt to match all the search strings of a given length simultaneously for all of the $O(S\sqrt{})$ possible search string lengths. This makes the overall algorithm run in $O(S^{1.5})$ which is sufficient to pass all test data.

Here's my hashing-based solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <cstdio>
#include <unordered_map>

using namespace std;

#define HM 1000000007
#define HA 100000007
#define HB 101

/* Given the hash 'h' of string S, computes the hash of S + 'ch'. */
int hext(int h, int ch) {
  return (1ll * h * HA + ch + HB) % HM;
}

int main() {
  freopen("censor.in", "r", stdin);
  freopen("censor.out", "w", stdout);
  int N;
  string S;
  cin >> S >> N;
```

```
  unordered_map<int, unordered_map<int, vector<string>>> T;
  for (int i = 0; i < N; i++) {
    string ti;
    cin >> ti;

    /* Compute the hash of search string ti. */
    int hsh = 0;
    for (int j = 0; j < ti.size(); j++) {
      hsh = hext(hsh, ti[j] - 'a');
    }

    /* Store all strings by length then hash value. */
    T[ti.size()][hsh].push_back(ti);
  }

  /* Build the result string one character at a time. */
  string R;
  vector<int> H(1, 0);
  vector<int> HAPW(1, 1);
  for (int i = 0; i < S.size(); i++) {
    int ch = S[i] - 'a';

    /* Update R and associated hash calculations. */
    H.push_back(hext(H.back(), ch));
    HAPW.push_back((1ll * HAPW.back() * HA) % HM);
    R += S[i];

    /* Loop over all search string lengths. */
    for (const auto& it : T) {
      int len = it.first;
      if (len > R.size()) {
        continue;
      }

      /* Compute the hash of the suffix of R with length len. */
      int hsub = (1ll * H[R.size() - len] * HAPW[len]) % HM;
      int hsh = (HM + H.back() - hsub) % HM;

      bool found = false;
      auto jt = it.second.find(hsh);
      if (jt != it.second.end()) {
        /* Loop over all input strings with matching hash and length. */
        for (const string& t : jt->second) {
          /* If the strings actually match then truncate R and associated hash arrays. */
          if (t == R.substr(R.size() - len)) {
            R.resize(R.size() - len);
            H.resize(H.size() - len);
            HAPW.resize(HAPW.size() - len);
            found = true;
            break;
          }
        }
      }
      if (found) {
        break;
      }
    }
  }

  cout << R << endl;
  return 0;
}
```

Aho-Corasick

The second approach is based on the Aho-Corasick algorithm. This algorithm is a generalization of the well known Knuth-Morris-Pratt algorithm to multiple search strings. By efficiently constructing a finite state machine Aho-Corasick can track what the longest prefix of any search string that matches and can efficiently update this when a new character is witnessed.

Therefore we can use Aho-Corasick to detect when a suffix of *R* matches a search string and simply truncate the search string by the length of the match string. In addition, it's also necessary to reset our Aho-Corasick state to what it was when we reached the truncated value of *R*.

There is one common pitfall seen in many competitors' code that caused them to miss one test case. Normally when using Aho-Corasick it is ok to not explicitly compute all of the next states for each possible character and simply walk back up the trie until a match for that character can be found. This works because the time spent walking up the trie can be amortized against the time spent walking down the trie. However, since we reset the state in this problem we cannot necessarily afford this. Therefore we should either explicitly precompute all state transitions or cache transitions.

Here's my Aho Corasick based solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <cstdio>

using namespace std;

#define SIGMA 26

/* The trie structure used to represent the Aho-Corasick finite state machine. */
struct trie {
  trie() : ind(-1), fall(NULL) {
    memset(next, 0, sizeof(next));
  }

  /* Gives the index in T that matches this state if any, otherwise -1. */
  int ind;

  /* Gives the longest state that is (strictly) a suffix of this state. */
  trie* fall;

  /* Gives the next state for each possible character. */
  trie* next[SIGMA];

  trie* step(int ch) {
    if (next[ch])
      return next[ch];
    else if (fall)
      /* Walk up the trie to find the next step and cache the result. */
      return next[ch] = fall->step(ch);
    else
      return this;
  }
};

int main() {
  int N;
  string S;
```

```
  cin >> S >> N;

  vector<string> T(N);
  for (int i = 0; i < N; i++)
    cin >> T[i];

  /* Sort the search strings by length using a C++11 lambda expression. */
  sort(T.begin(), T.end(), [](const string& x, const string& y) -> bool {
                             return x.size() < y.size(); });

  /* Build the AC finite state machine. */
  trie* rt = new trie;
  vector<trie*> tnode(N, rt);

  int j = 0;
  for (int i = 0; j < T.size(); i++) {
    /* Extend the trie one character deeper along all search strings. */
    for (int k = j; k < T.size(); k++) {
      /* Create a new node if necessary to represent the next prefix of T[k]. */
      int ch = T[k][i] - 'a';
      trie* nd = tnode[k];
      if (!nd->next[ch])
        nd->next[ch] = new trie;
      nd->next[ch]->fall = nd->fall;
      nd = nd->next[ch];

      /* Find the largest trie node that is a suffix of T[k][0...i]. */
      while (nd->fall && !nd->fall->next[ch]) {
        nd->fall = nd->fall->fall;
      }
      if (nd->fall)
        nd->fall = nd->fall->next[ch];
      else
        nd->fall = rt;

      tnode[k] = nd;
    }

    /* Set the 'ind' parameter on any finished strings and adjust our start index. */
    while (j < T.size() && i + 1 == T[j].size()) {
      tnode[j]->ind = j;
      j++;
    }
  }

  /* Build the result string one character at a time. */
  string R;
  vector<trie*> st(1, rt);
  for (int i = 0; i < S.size(); i++) {
    /* Extend the trie state with the new character. */
    trie* nd = st.back()->step(S[i] - 'a');
    R.push_back(S[i]);
    st.push_back(nd);

    /* If this state matches a string truncate R and rewind our state. */
    if (nd->ind != -1) {
      R.resize(R.size() - T[nd->ind].size());
      st.resize(R.size() + 1);
    }
  }

  cout << R << endl;
}
```