Try Palindromes

locked

by pruthvishalcodi1

Editorial by PruthvishE

Let there be two words x and y. Let us look at the cases that arise when they are concatenated (i.e., x+y) from the point of view of palindromes:

```
Case 1: length(x)<length(y):
In this case, let us assume that y' is that suffix of y which is of the sam
e length as x. If x+y is to be a palindrome then two conditions must be tru
e - x=y' and y-y' is a palindrome. For example, aba and ccaba will form a p
alindrome upon concatenation but aba and cdaba won't.

Case 2: length(x)>length(y):
In this case, let us assume that x' is that prefix of x which is of the sam
e length as y. If x+y is to be a palindrome then two conditions must be tru
e - y=x' and x-x' is a palindrome.

Case 3: length(x)=length(y):
In this case, if x+y is to be a palindrome then x=y.
```

These three cases lead us to formulate an algorithm. We need a data structure which can match prefixes of strings quickly. A trie is exactly for this purpose. We will take the words one by one, and put them in the trie. Before inserting word i, we will first find the number of words already in the trie that it forms a palindrome with upon being concatenated (word i being the second part in concatenation), and then insert it.

For finding the required numbers, we need to store two variables per node of the trie: uptill and below. The variable uptill stores the number of words that have the exact same letters as the ones from the root of the trie to this node. The variable below stores the number of those words w that have a prefix w' which contains the exact same letters as the ones from the root of the trie to this node and w−w' is a palindrome.

Keeping these operations in mind, we can define our insert(w) and getanswer(w) functions. The insert(w) function inserts the word w into the trie and getanswer(w) calculates the number of words already in the trie with which w would form a palindrome upon being concatenated.

We first define how getanswer(w) works. It first reverses w. Let's call the new string revw. Then it starts at the root of the trie and goes down as per the letters of revw. When it is at a node which is at depth i from the root, it has already processed the first i letters of revw, i.e., the prefix of length i. Let us call the processed prefix revw'. If revw−revw' is a palindrome, then the word w can form palindrome upon being contenated with the words ending at this particular node. Hence, if revw−revw' is a palindrome, we add the value stored in uptill variable of this node to the answer (this counts all possibilities under case 1). Once we reach the node which is the last letter of revw, we add the values stored in the

Statistics

Difficulty: Medium

Time Complexity: O(N) Required

Knowledge: Strings, Tries, Rolling Hash, Palindromes

Publish Date: Jul 05 2019

below and uptill variables of this node to our answer. This counts possibilities under case 2 and case 3.

The insert(w) works in a similar way. For this, we don’t have to reverse the word. We start at the root and go down the edges as per the letters in w. When we are at a node at depth i from the root, we have already processed the prefix w' of length i. At this node, we add 1 to the value stored in below variable if w−w' is a palidrome. When we reach the node which is last letter of w, we add 1 to the uptill variable of the node.

How can we efficiently check whether a prefix of a word is a palidrome or not? We can use ‘rolling hash’ to calculate for a given word w of length N an array mark in O(N) where mark[i]=1 if the prefix w[0..i] is a palidrome and 0 otherwise. Thus we can preprocess each string before inserting it or calculating the number of strings it forms palindromes with upon concatenation. Please see the author’s/editorialist’s solution for an example of rolling hash. You can read more about it from these links:

Set by PruthvishE

Problem Setter's code :

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cassert>
#include <algorithm>
#include <unordered_map>
#include <vector>
#include <string>
using namespace std;

const int MAX_LEN = 1000000;
const int MAXN = 1000000;

struct Trie
{
    vector< unordered_map<int, int> > next;
    vector<int> cnt;
    int nodes;

    void clear() {
        nodes = 0;
        int root = newNode();
    }

    int newNode() {
        if (nodes == next.size()) {
            next.push_back(unordered_map<int, int>());
            cnt.push_back(0);
        } else {
            next[nodes].clear();
            cnt[nodes] = 0;
        }
        return nodes ++;
    }

    void insert(const string &s) {
        int u = 0;
        for (int i = 0; i < s.size(); ++ i) {
            int c = s[i] - 'a';
            if (!next[u].count(c)) {
                next[u][c] = newNode();
            }
            u = next[u][c];
        }
        ++ cnt[u];
    }
}trie;

char temp[MAX_LEN + 1];
```

```
string words[MAXN];
int n;

unsigned long long MAGIC = 0xabcdef;
unsigned long long prefix[MAX_LEN + 1], suffix[MAX_LEN + 1], pw[MAX_LEN + 1
];

unsigned long long getHash(unsigned long long s[], int l, int r)
{
    return s[r + 1] - s[l] * pw[r - l + 1];
}

// AB is a Palindrome
// Suppose |A| < |B|
// Insert all A's in a trie
// Traverse the reverse of B on the trie, accumulate the answer when the pr
efix of B is a palindrome
long long solve()
{
    trie.clear();
    for (int i = 0; i < n; ++ i) {
        trie.insert(words[i]);
    }
    long long answer = 0;
    for (int i = 0; i < n; ++ i) {
        int len = words[i].size();
        prefix[0] = 0;
        suffix[0] = 0;
        for (int j = 0; j < len; ++ j) {
            prefix[j + 1] = prefix[j] * MAGIC + words[i][j];
            suffix[j + 1] = suffix[j] * MAGIC + words[i][len - 1 - j];
        }
        int u = 0;
        for (int j = len - 1; j >= 1; -- j) {
            int c = words[i][j] - 'a';
            if (!trie.next[u].count(c)) {
                break;
            }
            u = trie.next[u][c];

            if (getHash(prefix, 0, j - 1) ==  getHash(suffix, len - j, len
- 1)) {
                answer += trie.cnt[u];
            }
        }
    }
    return answer;
}

int main()
{
    pw[0] = 1;
    for (int i = 1; i <= MAX_LEN; ++ i) {
        pw[i] = pw[i - 1] * MAGIC;
    }
    int tests;
    for (assert(scanf("%d", &tests) == 1 && 1 <= tests && tests <= 5); test
s --; ) {
        assert(scanf("%d", &n) == 1 && 1 <= n && n <= MAXN);
        unordered_map<string, int> h;
        long long answer = 0;
        int length = 0;
        for (int i = 0; i < n; ++ i) {
            assert(scanf("%s", temp) == 1);
            int len = strlen(temp);
            assert(len <= MAX_LEN);
            for (int j = 0; j < len; ++ j) {
                assert('a' <= temp[j] && temp[j] <= 'z');
            }
            length += len;
            assert(length <= MAX_LEN);

            words[i] = temp;

            // deal with strings with the same length, i.e., |A| = |B|
            string rev = words[i];
            reverse(rev.begin(), rev.end());
            if (h.count(rev)) {
                answer += h[rev] * 2;
            }
            h[words[i]] += 1;
        }
        // solve |A| < |B|
        answer += solve();

        // solve |B| < |A|
        for (int i = 0; i < n; ++ i) {
            reverse(words[i].begin(), words[i].end());
        }
        answer += solve();

        printf("%lld\n", answer);
    }
    return 0;
}
```