Substring with Concatenation of All Words

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words** exactly once and without any intervening characters.

For example, given:
s: `"barfoothefoobarman"`
words: `["foo", "bar"]`

You should return the indices: `[0,9]`.
(order does not matter).

Solution 1

```cpp
// travel all the words combinations to maintain a window
// there are wl(word len) times travel
// each time, n/wl words, mostly 2 times travel for each word
// one left side of the window, the other right side of the window
// so, time complexity O(wl * 2 * N/wl) = O(2N)
vector<int> findSubstring(string S, vector<string> &L) {
    vector<int> ans;
    int n = S.size(), cnt = L.size();
    if (n <= 0 || cnt <= 0) return ans;

    // init word occurence
    unordered_map<string, int> dict;
    for (int i = 0; i < cnt; ++i) dict[L[i]]++;

    // travel all sub string combinations
    int wl = L[0].size();
    for (int i = 0; i < wl; ++i) {
        int left = i, count = 0;
        unordered_map<string, int> tdict;
        for (int j = i; j <= n - wl; j += wl) {
            string str = S.substr(j, wl);
            // a valid word, accumulate results
            if (dict.count(str)) {
                tdict[str]++;
                if (tdict[str] <= dict[str])
                    count++;
                else {
                    // a more word, advance the window left side possiablly
                    while (tdict[str] > dict[str]) {
                        string str1 = S.substr(left, wl);
                        tdict[str1]--;
                        if (tdict[str1] < dict[str1]) count--;
                        left += wl;
                    }
                }
                // come to a result
                if (count == cnt) {
                    ans.push_back(left);
                    // advance one word
                    tdict[S.substr(left, wl)]--;
                    count--;
                    left += wl;
                }
            }
            // not a valid word, reset all vars
            else {
                tdict.clear();
                count = 0;
                left = j + wl;
            }
        }
    }

    return ans;
}
```

written by shichaotan original link here

Solution 2

```
class Solution {
// The general idea:
// Construct a hash function f for L, f: vector<string> -> int,
// Then use the return value of f to check whether a substring is a concatenation

// of all words in L.
// f has two levels, the first level is a hash function f1 for every single word
in L.
// f1 : string -> double
// So with f1, L is converted into a vector of float numbers
// Then another hash function f2 is defined to convert a vector of doubles into a
single int.
// Finally f(L) := f2(f1(L))
// To obtain lower complexity, we require f1 and f2 can be computed through moving
window.
// The following corner case also needs to be considered:
// f2(f1(["ab", "cd"])) != f2(f1(["ac", "bd"]))
// There are many possible options for f2 and f1.
// The following code only shows one possibility (probably not the best),
// f2 is the function "hash" in the class,
// f1([a1, a2, ... , an]) := int( decimal_part(log(a1) + log(a2) + ... + log(an))
* 1000000000 )
public:
    // The complexity of this function is O(nW).
    double hash(double f, double code[], string &word) {
        double result = 0.;
        for (auto &c : word) result = result * f + code[c];
        return result;
    }
    vector<int> findSubstring(string S, vector<string> &L) {
        uniform_real_distribution<double> unif(0., 1.);
        default_random_engine seed;
        double code[128];
        for (auto &d : code) d = unif(seed);
        double f = unif(seed) / 5. + 0.8;
        double value = 0;

        // The complexity of the following for loop is O(L.size( ) * nW).
        for (auto &str : L) value += log(hash(f, code, str));

        int unit = 1e9;
        int key = (value-floor(value))*unit;
        int nS = S.size(), nL = L.size(), nW = L[0].size();
        double fn = pow(f, nW-1.);
        vector<int> result;
        if (nS < nW) return result;
        vector<double> values(nS-nW+1);
        string word(S.begin(), S.begin()+nW);
        values[0] = hash(f, code, word);

        // Use a moving window to hash every word with length nW in S to a float
number,
        // which is stored in vector values[]
        // The complexity of this step is O(nS).
```

```
        for (int i=1; i<=nS-nW; ++i) values[i] = (values[i-1] - code[S[i-1]]*fn)*
f + code[S[i+nW-1]];

        // This for loop will run nW times, each iteration has a complexity O(nS/
nW)
        // So the overall complexity is O(nW * (nS / nW)) = O(nS)
        for (int i=0; i<nW; ++i) {
            int j0=i, j1=i, k=0;
            double sum = 0.;

            // Use a moving window to hash every L.size() continuous words with l
ength nW in S.
            // This while loop will terminate within nS/nW iterations since the i
ncreasement of j1 is nW,
            // So the complexity of this while loop is O(nS / nW).
            while(j1<=nS-nW) {
                sum += log(values[j1]);
                ++k;
                j1 += nW;
                if (k==nL) {
                    int key1 = (sum-floor(sum)) * unit;
                    if (key1==key) result.push_back(j0);
                    sum -= log(values[j0]);
                    --k;
                    j0 += nW;
                }
            }
        }
        return result;
    }
};
```

Though theoretically it has a very small chance to fail.

written by jiajingfei original link here

Solution 3

The following python code is accepted by OJ. It is based on the following idea (assumption)

- We know that two multisets consist of same elements and size of the multisets are equal. if sum of hashes of all elements are the same for these multisets -> those multisets are identical

This is not true for same very very rare cases. Please describe such a case.

```
def findSubstring(self, S, L):
    n = len(L) #num words
    w = len(L[0])  #length of each word
    t = n*w    # total length

    hashsum = sum([hash(x) for x in L])
    h = [hash(S[i:i+w])*(S[i:i+w] in L) for i in xrange(len(S)-w+1)]
    return [i for i in xrange(len(S)-t+1) if sum(h[i:i+t:w])==hashsum]
```

written by hebele original link here