Sherlock and Anagrams

Isaac DeJager

March 15, 2023

Click here to view the HackerRank page

1 Problem Description

Two strings are anagrams of each other if the letters of one string can be rearranged to form the other string. Given a string, find the number of pairs of substrings of the string that are anagrams of each other.

Exmaple
$s = mom$
The list of all anagrammatic pairs is $[m, m]$, $[mo, mo]$ at positions $[[0], [2]]$, $[[0, 1], [1, 2]]$ respectively.

Function Description
Complete the function sherlockAndAnagrams in the editor below. sherlockAndAnagrams has the following parameter:

- string s: a string

Returns

- int: the number of unordered anagrammatic pairs of substrings in s

Input Format
The first line contains an integer q, the number of queries. Each of the next q lines contains a string s to analyze.

Constraints

- $1 \leq q \leq 10$
- $2 \leq |s| \leq 100$
- s contains only lowercase letters in the range ascii[a-z]

Sample Input 0

2
abba
abcd

Sample Output 0

4
0

Explanation 0

The list of all anagrammatic pairs is $[a, a], [ab, ba], [b, b]$ and $[abb, bba]$ at positions $[[0], [3]], [[0, 1], [2, 3]], [[1], [2]]$ and $[[0, 1, 2], [1, 2, 3]]$ respectively.

No anagrammatic pairs exist in the second query as no character repeats.

Sample Input 1

2
ifailuhkqq
kkkk

Sample Output 1

3
10

Explanation 1

For the first query, we have anagram pairs $[i, i], [q, q]$ and $[ifa, fai]$ at positions $[[0], [3]], [[8], [9]]$ and $[[0, 1, 2], [1, 2, 3]]$ respectively.

For the second query:

There are 6 anagrams of the form $[k, k]$ at positions $[[0], [1]], [[0], [2]], [[0], [3]], [[1], [2]], [[1], [3]]$ and $[[2], [3]]$.

There are 3 anagrams of the form $[kk, kk]$ at positions $[[0, 1], [1, 2]], [[0, 1], [2, 3]]$ and $[[1, 2], [2, 3]]$.

There is 1 anagram of the form $[kkk, kkk]$ at position $[[0, 1, 2], [1, 2, 3]]$.

Sample Input 2

1
cdcd

Sample Output 2

5

Explanation 2

There are two anagrammatic pairs of length 1: $[c, c]$ and $[d, d]$. There are three anagrammatic pairs of length 2: $[cd, dc], [cd, cd], [dc, cd]$ at positions $[[0, 1], [1, 2]], [[0, 1], [2, 3]], [[1, 2], [2, 3]]$ respectively.

2 Solution

```
int compareStrings(int aChars[26], int bChars[26]) {
    for (int i = 0; i < 26; i++) {
        if (aChars[i] != bChars[i]) {
            return 0;
        }
    }
    return 1;
}

int sherlockAndAnagrams(string s) {
    string a, b;
    int aChars[26] = { 0 };
    int bChars[26] = { 0 };
    int x = 0;

    for (int l = 1; l < s.length(); l++) {
        for (int i = 0; i < s.length() - l; i++) {
            a = s.substr(i, l);
            for (int p = 0; p < 26; p++) {
                aChars[p] = 0;
            }
            for (int p = 0; p < a.length(); p++) {
                aChars[int(a.at(p)) - 97]++;
            }
            for (int j = i + 1; j < s.length() - l + 1; j++) {
                b = s.substr(j, l);
                for (int p = 0; p < 26; p++) {
                    bChars[p] = 0;
                }
                for (int p = 0; p < b.length(); p++) {
                    bChars[int(b.at(p)) - 97]++;
                }

                x += compareStrings(aChars, bChars);
            }
        }
    }

    return x;
}
```

3 Analysis

The logic to this solution is very straight forward. Given a string s, find every possible substring of s with the exception of s itself. And, for every substring, compare it against every other substring of equal length and determine whether or not they are anagrams.

The code works by use 3 nested loops. Loop 1 uses variable l to represent the length of each substring. l counts from 1 up to $|s|-1$, meaning that it checks all substrings of length 1, then all substrings of length 2, etc until l reaches $|s|-1$. Loop 2 creates the substring a of length l, starting at index $i = 0$. The final loop compares a against every substring of length l that follows a.

As an example:
$s = abcde$
When $l = 1$, the loops just do the following substring comparisions:

$[a, b], [a, c], [a, d], [a, e]$
$[b, c], [b, d], [b, e]$
$[c, d], [c, e]$
$[d, e]$

For $l = 2$:

$[ab, bc], [ab, cd], [ab, de]$
$[bc, cd], [bc, de]$
$[cd, de]$

For $l = 3$

$[abc, bcd], [abc, cde]$
$[bcd, cde]$

And for $l = 4$

$[abcd, bcde]$.

So, every substring is compared to every other substring of equal length. If two substrings have different lengths, we know they cannot be anagrams of each other because the longer one is sure to have more a character than the shorter.

When comparing two strings, we need only create a hashmap for each string, hence the arrays $aChars$ and $bChars$. After collecting each string, we loop through them, using the indices of the array to represent the number of each character. Index $0 = a$, index $1 = b$, and so on. Then, we use a separate function to loop through the arrays. If at any index, the number of characters is different, we know the substrings are not anagrams. If the arrays match perfectly, they are anagrams and we increment the total number of anagrams, x, by 1.

The time complexity for this, unfortunately, is not pretty. To find the time complexity, we count the number of substring comparisons. Consider again the string $s = abcde$. For $l = 1$, there are:

$[a,b],[a,c],[a,d],[a,e] \rightarrow 4$ comparisons
$[b,c],[b,d],[b,e] \rightarrow 3$ comparisons
$[c,d],[c,e] \rightarrow 2$ comparisons
$[d,e] \rightarrow 1$ comparisons

$$\sum_{i=1}^{4} i = 4+3+2+1 = 10$$

So, there are 10 comparisons for $l = 1$.
For $l = 2$, there are $\sum_{i=1}^{3} i = 6$ comparisons (see the substrings above).
For $l = 3$, $\sum_{i=1}^{2} = 3$.
And for $l = 4$, $\sum_{i=1}^{1} = 1$.
So, the total number of comparisons for $s = abcde$ is $10+6+3+1 = 20$.

Now, to calculate the number of comparisons for a string s with any length n, we sum up all the comparisons between all substrings of s with the same length. So, we end up doing a sum of sums. The outer sum counts the length of the substrings, so it starts at $i = 1$ and goes to $i = n-1$. It doesn't need to count all the way to $i = n$ since there is only one substring of s with that length and that is s itself.

The inner sum adds up all the comparisons, and as seen in the example above, it is a simple sum of j. So, the full sum looks like:

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} j = \sum_{i=1}^{n-1} \frac{(n-i)(n-i+1)}{2} = \sum_{i=1}^{n-1} \frac{n^2-ni+n-ni+i^2-i}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{n-1}\Big(n^2+n+i^2-i(2n+1)\Big) = \frac{1}{2}\Big(\sum_{i=1}^{n-1}(n^2+n)+\sum_{i=1}^{n-1} i^2+\sum_{i=1}^{n-1} i(2n+1)\Big)$$

$$= \frac{1}{2}\Big((n^2+n)(n-1)+\frac{(n-1)(n-1+1)(2(n-1)+1)}{6}-(2n+1)\frac{(n-1)(n-1+1)}{2}\Big)$$

$$= \frac{n^3-n^2+n^2-n}{2}+\frac{(n^2-n)(2n-1)}{12}-\frac{(2n+1)(n^2-n)}{4}$$

$$= \frac{n^3-n}{2}+\frac{2n^3-n^2-2n^2+n}{12}-\frac{n^2-n+2n^3-2n^2}{4}$$

$$= \frac{n^3-n}{2}+\frac{2n^3-3n^2+n}{12}-\frac{2n^3-n^2-n}{4}$$

$$= \frac{6n^3-6n+2n^3-3n^2+n-6n^3+3n^2+3n}{12}$$

$$= \frac{2n^3 - 2n}{12} = \frac{n^3}{6} + \frac{n}{6}$$

So, this algorithm takes $O(n^3)$ time complexity. Fortunately, the space complexity only requires a 3 isolated variables and two integer arrays of size 26.