String I

Introduction

String

A **String** is generally considered to be a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, characters, using some character encoding.

An instance of a string is called **String Literal.** In other words, String is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than number. It consists of a set of characters that can also contain spaces and numbers.

Char Array

A **character array** is a collection of the variables of “char” datatype; it can be a one-dimensional array or a two-dimensional array. It is also called a “null terminated string”. A Character array is a sequence of the characters that are stored in consecutive memory addresses. In a character array, a particular character can be accessed by its index. A “Null character” terminates the character array”.

Example

String

C	O	D	I	N	G

Char Array

C	O	D	I	N	G	\O

String Matching Algorithms:

1. Brute Force Approach

When it comes to string matching, the most basic approach is what is known as brute force, which simply means to check every single character from the text to match against the pattern. In general we have a text and a pattern (most commonly shorter than the text). What we need to do is to answer the question whether this pattern appears in the text.

Overview

The principles of brute force string matching are quite simple. We must check for a match between the first characters of the pattern with the first character of the text as in the picture below.

Example

Text :

x	x	x	x	y	x	x	z

Pattern:

x	x	z

If they don't match, we move forward to the second character of the text. Now we compare the first character of the pattern with the second character of the text. If they don't match again, we move forward until we get a match or until we reach the end of the text.

Text:

x	x	x	x	y	x	x	z

Pattern:

x	x	z

Text:

x	x	x	x	y	x	x	z

Pattern:

x	x	z

Text:

x	x	x	x	y	x	x	z

Pattern:

x	x	z

Text:

x	x	x	x	y	x	x	z

Pattern:

x	x	z

In case they match, we move forward to the second character of the pattern, comparing it with the “next” character of the text, as shown in the picture below. Just because we have found a match between the first character from the pattern and some character of the text, doesn’t mean that the pattern appears in the text. We must move forward to see whether the full pattern is contained in the text.

Text:

x	x	x	x	y	x	x	z

Pattern:

x	x	z

Time Complexity

Suppose there are n characters in the Text and m characters in the Pattern. So the Time Complexity of Brute Force is **O(n*m)**,

Space Complexity

Space Complexity for the Brute Force algorithm is **O(1).**

2. Rabin Karp Algorithm

The **Rabin-Karp string matching algorithm** calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. Generalizations of the same idea can be used to find more than one match of a single pattern, or to find matches for more than one pattern.

How Hash Function Works?

The **Rabin-Karp algorithm** works by using hashing to see if the pattern that is being searched for is in the text if yes, then at what indexes. This works by assigning each character a value. For example, in the algorithm implementation we can use the **ASCII values** of the characters.

Example:

Find the hash code value of pattern string using the hash code assigned to characters

Text:

a	a	b	a	a	b	b

Pattern:

a	b	b

The hash value for the pattern 'abb' will be 1+2+2 = 5.

Hash value of the first three characters '**aab'** in the text is **4**. Similarly, we will start comparing the hash value of 3 characters of the text with the hash value of the pattern. If the hash value of any three characters matches with the hash value of the pattern. We say that pattern is matched with the text.

In this method, there might be a possibility that the hash value can be the same for more than one pattern in the text.

To overcome this situation or to avoid the spurious hits, we will take the nearest prime number of 26 (refers to the number of letters in an English alphabet) i.e. 29. And use it in the hash function to make the string unique.

Example :

Test :

d	d	b	d	b	a

Pattern :

d	b	a

The hash value for the pattern would be

$= 4x29^2 + 2x29^1 + 1x29^0$

$= \mathbf{3423}$

This hash value will be matched with the hash value of three characters in the text. If this value of hash function matches with the hash value of text, we say that pattern is matched in the text.

Code:

```
#include <iostream>
#include <bits/stdc++.h>
#define ll long long
using namespace std;

long long M = 1e9+7;
long long mod(long long N){
    return (N%M + M)%M;
}

long long mul(long long a, long long b){
    return mod(mod(a)*mod(b));
}

long long add(long long a, long long b){
    return mod(mod(a)+mod(b));
}
```

```
long long sub(long long a, long long b){
    return mod(mod(a)-mod(b)+M);
}

void solve(){
    string text,pattern;
    cin>>text>>pattern;
    int tn =text.size();
    int pn = pattern.size();

    long long p = 31;

    // power mod M
    vector<long long>powMod(tn);
    powMod[0]=1;
    for(int i=1;i<tn;i++){
        powMod[i] = mul(powMod[i-1],p);
    }

    // calculating hash for pattern
    long long ph=0;
    for(int i=0;i<pn;i++) ph=add(ph,mul(pattern[i]-'a'+1,powMod[i]));

    // calculating prefix Hash for the text
    vector<long long>th(tn+1);
    for(int i=0;i<tn;i++){
        th[i+1]=add(th[i],mul(text[i]-'a'+1,powMod[i]));
    }

    // matching the text hash and pattern hash
    for(int i=0;i<tn-pn+1;i++){
        long long hashVal = sub(th[i+pn],th[i]);
        if(hashVal == mul(ph,powMod[i])){
            cout<<"Fount at index "<<i<<"\n";
        }
    }
}

int main(){
    solve();
    return 0;
}
```

Time Complexity:

The best case complexity of the Rabin Karp algorithm is **O(m+n).** And the worst case is even after using the modified hash function is **O(n*m)**.

Space Complexity:

Space complexity of the Rabin Karp algorithm is **O(m+n)**

3. KMP Algorithm (Knuth - Morris - Pratt)

The KMP algorithm is used to find a **"Pattern"** in a **"Text"**. This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called **"Prefix Table " to** skip character comparison while matching. Sometimes the prefix table is also known as the LPS **Table**. Here LPS stands for **"Longest proper Prefix which is also Suffix"**.

It is used because it improves the worst time complexity to O(n). It is much preferred for larger strings.

Steps for creating LPS Table (Prefix Table):

- Step 1 - Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- Step 2 - Define variables **i & j**. Set i = 0, j = 1 and LPS[0] = 0.
- Step 3 - Compare the characters at **Pattern[i]** and **Pattern[j].**
- Step 4 - If both are matched then set **LPS[j] = i+1** and increment both i & j values by one. Goto to Step 3.

- Step 5 - If both are not matched then check the value of variable 'i'. If it is '0' then set **LPS[j] = 0** and increment the 'j' value by one, if it is not '0' then set **i = LPS[i-1]**. Goto Step 3.
- Step 6- Repeat above steps until all the values of LPS[] are filled.

Creating KMP Algorithm LPS Table :

Consider the following pattern

Pattern:

A	B	C	D	A	B	D

Let us define LPS Table with size 7 which is equal to the pattern

LPS:

0	1	2	3	4	5	6

The final LPS table for the pattern will be

0	1	2	3	4	5	6
0	0	0	0	1	2	0

How KMP Algorithm works?

Step 1:

Text:

A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E

Pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Step 2:

Text:

A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E

Pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Step 3:

Text:

A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E

Pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Step 4:

Text:

A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E

Pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Step 5:

Text:

A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E

Pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Code:

```
#include <iostream>
#include <bits/stdc++.h>
#define ll long long
using namespace std;


void solve(){
    string text,pattern;
    cin>>text>>pattern;

    int tn = text.size();
    int pn = pattern.size();

    vector<int>lps(pn);
    int j=0;
    for(int i=1;i<pn;i++){
```

```
        while(j>0 && pattern[i]!=pattern[j]){
            j = lps[j-1];
        }
        if(pattern[i] == pattern[j]) j++;
        lps[i]=j;
    }

    vector<int>index;
    int i=0;
    j=0;
    while(i<tn){
        if(text[i] == pattern[j]){
            i++;
            j++;
        }
        else{
            if(j!=0){
                j=lps[j-1];
            }
            else{
                i++;
            }
        }
        if(j==pn) index.push_back(i-pn);
    }

    if(index.size()==0) cout<<-1;
    else{
        for(auto i : index) cout<<"Pattern found at index "<<i<<"\n";
    }

}

int main(){
    solve();
    return 0;
}
```

Time Complexity:

We can see that the total time complexity of the KMP algorithm is **O(m+n) which is a linear time complexity** and where **m** is the size of the pattern string and **n** is the size of the main string.

Space Complexity:

Space Complexity for the KMP algorithm is **O(n).**

4. Z-Algorithm

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of the text be n and the pattern be m, then the total time taken is **O(m + n)** with linear space complexity. Now we can see that both time and space complexity is the same as the KMP algorithm but this algorithm is Simpler to understand.

In this algorithm, we construct a **Z array.**

What is a Z-Array?

For a string str[0..n-1], the Z array is of the same length as string. An element Z[i] of the Z array stores the length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of the Z array is meaningless as the complete string is always a prefix of itself.

Example:

Text:

a	b	a	b	a	b	c

Pattern:

a	b	a	b	c

Z - Array:

0	0	2	0	0

The idea is to concatenate pattern and text, and creating a string **"P#T"**, where P is the pattern, # is a special character and should not be present in the string or the text, T is the text. Now it is required to build the Z-array for the concatenated string. In the Z array, if the Z value at any point is equal to pattern length, then pattern is present at that point.

M-text:

a	b	a	b	c	#	a	b	a	b	a	b	c

Z -Array:

0	0	2	0	0	0	4	0	5	0	2	0	0

Since, length of the pattern is 5, the value of 5 in Z-Array indicates presence of the pattern.

Code:

```
#include <iostream>
#include <bits/stdc++.h>
#define ll long long
using namespace std;

vector<int> calZarray(string s){
    int n = s.size();
    vector<int>z(n);

    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
```

```
            l = i, r = i + z[i] - 1;
    }
    return z;
}

void solve(){
    string text,pattern;
    cin>>text>>pattern;

    int tn = text.size();
    int pn = pattern.size();

    string newString=pattern;
    newString+=(char)('#');
    newString+=text;

    int newLen = newString.size();
    vector<int>index;

    vector<int>Zarray = calZarray(newString);

    for(int i=0;i<newLen;i++){
        if(Zarray[i] == pn) index.push_back(i-pn-1);
    }

    for(auto i : index) cout<<i<<" ";
}

int main(){
    solve();
    return 0;
}
```

Time Complexity:

Z algorithm is an algorithm for searching a given pattern in a string. It is an efficient algorithm as it has linear time complexity. It has a time complexity of

O(m+n), where m is the length of the string and n is the length of the pattern to be searched.

Space Complexity:

The auxiliary space complexity is **O(n)**, where m is the length of the text.

Good Substring

Problem:

You've got string *s*, consisting of small English letters. Some of the English letters are *good*, the rest are *bad*.A substring ***s*[*l*...*r*] (1 ≤ *l* ≤ *r* ≤ |*s*|) of string *s* = *s*1*s*2...*s*|*s*|** (where |*s*| is the length of string *s*) is string $sl sl+1 \ldots sr$.

The substring ***s*[*l*...*r*]** is *good*, if among the letters $sl, sl+1, \ldots, sr$ there are at most *k* bad ones (look at the sample's explanation to understand it more clear).

Your task is to find the number of distinct good substrings of the given string *s*. Two substrings ***s*[*x*...*y*] and *s*[*p*...*q*]** are considered distinct if their content is different, i.e. ***s*[*x*...*y*] ≠ *s*[*p*...*q*]**.

Approach:

We can use Rabin-Karp rolling hash to count substrings that differ by content. Just sort the hashes of all good substrings and find the number of unique hashes (equal hashes will be on adjacent positions after sort). But these hashes are unreliable in general, so it's always better to use precise algorithm.

Code:

```
#include <iostream>
#include <bits/stdc++.h>
#define ll long long
using namespace std;
```

```
const long long mul = 1e18 + 7;

void solve(){
    string str,badChar;
    int k;
    cin>>str>>badChar;
    cin>>k;
    int n = str.size();
    set<long long>s;
    long long cnt,total;
    for(int i=0;i<n;i++){
        cnt=0;
        total=0;
        for(int j=i;j<n;j++){
            total = (total*26LL + (str[j]-'0'))%mul;
            cnt +=badChar[str[j]-'a'] == '0';
            if(cnt>k) break;
            s.insert(total);
        }
    }
    cout<<s.size()<<"\n";
}

int main(){
    solve();
    return 0;
}
```

Prefixes and Suffixes

Problem:

You have a string $s = s1s2...s|s|$, where $|s|$ is the length of string s, and si its i-th character.

Let's introduce several definitions:

- A substring $s[i..j]$ $(1 \le i \le j \le |s|)$ of string s is string $sisi+1...sj$.
- The prefix of string s of length l $(1 \le l \le |s|)$ is string $s[1..l]$.
- The suffix of string s of length l $(1 \le l \le |s|)$ is string $s[|s| - l + 1..|s|]$.

Your task is, for any prefix of string s which matches a suffix of string s, print the number of times it occurs in string s as a substring.

Approach:

The problem could be solved using different algorithms with **z** and **prefix functions**. Let's describe the solution with the prefix function **p** of string **s**.

Calc prefix function and create a tree where vertices — integers from 0 to |s|, edges — from **p[i]** to **i** for every **i**. The root of the tree is **0**. For every vertex **v** calc the number of values **p[i] = v** — that is **cnt[v]**. Then for every v calc the sum of all values cnt[u] for every **u** into subtrees of **v**. The general answer to the problem is:

Find all lengths of the prefixes which matches the suffixes — these values are **|s|, p[|s|], p[p[|s|]], p[p[p[|s|]]]...** For every such length **L** the answer to the problem is **sum[L] + 1.**

Code:

```
#include <iostream>
#include <bits/stdc++.h>
#define ll long long
using namespace std;

vector<int> calZarray(string s){
    int n = s.size();
```

```
    vector<int>z(n);

    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}

void solve(){
    string text;
    cin>>text;

    int tn = text.size();

    vector<int>z = calZarray(text);
    vector<int>temp(tn+1);

    z[0]=tn;

    for(int i=0;i<tn;i++) temp[z[i]]++;
    for(int i=tn;i>0;i--) temp[i-1]+=temp[i];

    vector<pair<int,int>>ans;
    for(int i=tn-1;i>=0;i--){
        if(i+z[i]==tn) ans.push_back({z[i],temp[z[i]]});
    }
    coût<<ans.size()<<"\n";
    for(auto i : ans) cout<<i.first<<" "<<i.second<<"\n";
}

int main(){
    solve();
    return 0;
}
```