**KMP Algorithm:**

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

(i.e, length of text in which pattern is searched is > the length of pattern)

The Naive pattern searching algorithm doesn’t work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

txt[] = "AAAAAAAAAAAAAAAAAB"

pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"

pat[] = "ABABAC" (not a worst case, but a bad case for Naive)

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

To find out the sub patterns appearing more than once in the pattern, we need to do preprocessing.

* KMP algorithm does preprocesses pat[] and constructs an auxiliary lps[] of size m (same as size of pattern) which is used to skip characters while matching.  
    
  (though, LPS’s full form is Longest Prefix which is also a suffix, it is actually longest proper prefix which is also a suffix)
* For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

Examples of lps[] construction:

For the pattern “AAAA”,

lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”,

lps[] is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”,

lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “AAACAAAAAC”,

lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern “AAABAAA”,

lps[] is [0, 1, 2, 0, 1, 2, 3]

**Algorithm For Computing LPS:**

void computeLPSArray(char \*pat, int M, int \*lps)

{

**// length of the previous longest prefix suffix**

int len = 0;

**//len always represent that. The length of previous longest proper prefix which is also suffix**

lps[0] = 0; **// lps[0] is always 0**

**// the loop calculates lps[i] for i = 1 to M-1**

int i = 1;

**//I is the starting index from where lps calculation starts**

while (i < M)

{

if (pat[i] == pat[len])

{

len++;

lps[i] = len;

i++;

}

else // (pat[i] != pat[len])

{

// This is tricky. Consider the example.

// AAACAAAA and i = 7. The idea is similar

// to search step.

if (len != 0)

{

len = lps[len-1];

// Also, note that we do not increment

// i here

}

else // if (len == 0)

{

lps[i] = 0;

i++;

//incrementing I is important

}

}

}

Now, let's consider the pattern: ABABCABAB

Now, initially i=1 and len=0

So, pat[i]!=pat[len]

hence, lps[1]=0 i.e. for string[0..1] the length of the proper prefix which is also a

suffix is 1

for, lps[0]=0 always as proper prefix is compared to suffix

Now, consider i=2 i.e. string length 3

we know lps[1]=0, lps[0]=0 hence, length is never increased

Now, length finally becomes 1 Since, pat[len]=pat[2] Hence lps[2]=1

**(for str[0..2] i.e. from ABABCABAB, we are checking for ABA Hence, A and A)**

Now, for str[0..3] length=1 pat[1]==pat[3] **(since, we already know str[0] matched with str[2])**

Hence, lps[3]=2

Now, for str[0...4]

length=2 pat[2]!=pat[4]

Now, length=0.

hence, len=lps[len-1]

Now, why this, because, previously len number of characters are matched

So, if pat[len]=pat[i] matches we could directly say

However, consider the pattern ABABB

Now, lps[0]=0

lps[1]=0

lps[2]=1 len=1 i=3 (after that)

Now, pat[len]!=pat[i]

So, len=lps[len-1]

Now, len becomes lps[0] i.e. no characters matched

Now, what does that really mean? Think again.

ABABCABAB

Now, length=0,i=1 lps[0] is always 0, because, for a single character string, there is no proper prefix

Now, lps[1] is 0

lps[2] is 1

Now, we already know, previous character i.e. character present in position 2 is matched with position 0, hence, we will compare position 1 to position 3

lps[3] is 2 since, character matched

Now, length=2 and i=4 Now, they don’t match Now, we already know, previously AB matched with AB (prefix). Now, A is not matching . Hence, It will went back to first character before B (at position 1) i.e. at position 0 (because, there is no point in comparing current character c with B since, B is already matched c’s previous character B and they are not same

A B A B C

^ ^

| |

**Pattern Searching Part:**

void KMPSearch(char \*pat, char \*txt)

{

int M = strlen(pat);

int N = strlen(txt);

// create lps[] that will hold the longest prefix suffix

// values for pattern

int lps[M];

// Preprocess the pattern (calculate lps[] array)

computeLPSArray(pat, M, lps);

int i = 0; // index for txt[]

int j = 0; // index for pat[]

while (i < N)

{

if (pat[j] == txt[i])

{

j++;

i++;

}

if (j == M)

{

printf("Found pattern at index %d n", i-j);

j = lps[j-1];

}

// mismatch after j matches

else if (i < N && pat[j] != txt[i])

{

// Do not match lps[0..lps[j-1]] characters,

// they will match anyway

if (j != 0)

j = lps[j-1];

else

i = i+1;

}

}

}

**Z Algorithm:**

Z algorithm (Linear time pattern searching Algorithm)

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

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

**What is Z Array?**

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

**The first entry of Z array is meaning less as complete string is always prefix of itself.**

Example:

Index 0 1 2 3 4 5 6 7 8 9 10 11

Text a a b c a a b x a a a z

Z values X 1 0 0 3 1 0 0 2 2 1 0

**How is Z array helpful in Searching Pattern in Linear time?**

The idea is to concatenate pattern and text, and create a string “P$T” where P is pattern, $ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

**(Note that pattern is concatenated first)**

**How To Construct An Z Array:**

The idea is to maintain an interval [L, R] which is the interval with max R

such that [L,R] is prefix Substring (Substring which is also prefix).

Steps for maintaining this interval are as follows –

1) If i > R then there is no prefix Substring that starts before i and

ends after i, so we reset L and R and compute new [L,R] by comparing

str[0..] to str[i..] and get Z[i] (= R-L+1).

**(in the current interval L,R)**

**(Since, I does not lie in the current interval. We need to calculate another interval)**

2) If i <= R then let K = i-L, now Z[i] >= min(Z[K], R-i+1) because

str[i..] matches with str[K..] for at least R-i+1 characters (they are in

[L,R] interval which we know is a prefix Substring).

**(What does that mean? I<=R now, that means str[0…k-1] is already matched with str[L..i-1] (rightly assumed)**

**Now, we will check if if range of L,R can be extended or not.**

**How? If Z[k] where k=I-L contains a value which is greater than the remaining characters (R-I+1) then the range can be extended**

**Otherwise, range cannot be extended and Z[I]=Z[k])**

**Now, we can use either Z[i-L] value for Z[i] or freshly calculate Z[i]'s value depending on whether the range could be expanded or not**

**)**

Now two sub cases arise –

a) If Z[K] < R-i+1 then there is no prefix Substring starting at

str[i] (otherwise Z[K] would be larger) so Z[i] = Z[K] and

interval [L,R] remains same.

b) If Z[K] >= R-i+1 then it is possible to extend the [L,R] interval

thus we will set L as i and start matching from str[R] onwards and

get new R then we will update interval [L,R] and calculate Z[i] (=R-L+1)

The algorithm runs in linear time because we never compare character less than R and with matching we increase R by one so there are at most T comparisons. In mismatch case, mismatch happen only once for each i (because of which R stops), that’s another at most T comparison making overall linear complexity.

(even when, we can see a room for extending the range, we starts it from R)

// A C++ program that implements Z algorithm for pattern searching

#include<iostream>

using namespace std;

void getZarr(string str, int Z[]);

// prints all occurrences of pattern in text using Z algo

void search(string text, string pattern)

{

// Create concatenated string "P$T"

string concat = pattern + "$" + text;

int l = concat.length();

// Construct Z array

int Z[l];

getZarr(concat, Z);

// now looping through Z array for matching condition

for (int i = 0; i < l; ++i)

{

// if Z[i] (matched region) is equal to pattern

// length we got the pattern

if (Z[i] == pattern.length())

cout << "Pattern found at index "

<< i - pattern.length() -1 << endl;

}

//we can safely do that, since, though Z[0] would logically contain pattern.length() we make it 0

}

// Fills Z array for given string str[]

void getZarr(string str, int Z[])

{

int n = str.length();

int L, R, k;

// [L,R] make a window which matches with prefix of s

L = R = 0;

**//see, never calculated for z[0]**

for (int i = 1; i < n; ++i)

{

// if i>R nothing matches so we will calculate.

// Z[i] using naive way.

if (i > R)

{

L = R = i;

**//set both L and R to current point and trying to extending it to get an interval L..R]**

**// R-L = 0 in starting, so it will start**

**// checking from 0'th index. For example,**

**// for "ababab" and i = 1, the value of R**

**// remains 0 and Z[i] becomes 0. For string**

**// "aaaaaa" and i = 1, Z[i] and R become 5**

while (R<n && str[R-L] == str[R])

R++;

Z[i] = R-L;

R--;

}

else

{

// k = i-L so k corresponds to number which

// matches in [L,R] interval.

k = i-L;

// if Z[k] is less than remaining interval

// then Z[i] will be equal to Z[k].

// For example, str = "ababab", i = 3, R = 5

// and L = 2

if (Z[k] < R-i+1)

Z[i] = Z[k];

// For example str = "aaaaaa" and i = 2, R is 5,

// L is 0

else

{

// else start from R and check manually

L = i;

**//we starts from L=I and R (no less than R is never checked. Here, R is just extended)**

while (R<n && str[R-L] == str[R])

R++;

Z[i] = R-L;

R--;

}

}

}

}

// Driver program

int main()

{

string text = "GEEKS FOR GEEKS";

string pattern = "GEEK";

search(text, pattern);

return 0;

}

**How to Find Longest Prefix Which Is Also Suffix:**

**This sounds like computing LPS array (but is it?)**

**Since, LPS: Longest Proper Prefix Which is also suffix**

It is indeed. But, a very obvious and small trick.

// Efficient CPP program to find length of

// the longest prefix which is also suffix

#include<bits/stdc++.h>

using namespace std;

// Returns length of the longest prefix

// which is also suffix and the two do

// not overlap. This function mainly is

// copy computeLPSArray() of in below post

// https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

int longestPrefixSuffix(string s)

{

int n = s.length();

int lps[n];

lps[0] = 0; // lps[0] is always 0

// length of the previous

// longest prefix suffix

int len = 0;

// the loop calculates lps[i]

// for i = 1 to n-1

int i = 1;

while (i < n)

{

if (s[i] == s[len])

{

len++;

lps[i] = len;

i++;

}

else // (pat[i] != pat[len])

{

// This is tricky. Consider

// the example. AAACAAAA

// and i = 7. The idea is

// similar to search step.

if (len != 0)

{

len = lps[len-1];

// Also, note that we do

// not increment i here

}

else // if (len == 0)

{

lps[i] = 0;

i++;

}

}

}

int res = lps[n-1];

**// Since we are looking for**

**// non overlapping parts.**

**//How, here is the trick**

return (res > n/2)? n/2 : res;

}

// Driver program to test above function

int main()

{

string s = "abcab";

cout << longestPrefixSuffix(s);

return 0;

}

**Splitting a Numeric String**

Given a numeric string (length <= 32), split it into two or more integers( if possible), such that

1) Difference between current and previous number is 1.

2) No number contains leading zeroes

If it is possible to separate a given numeric string then print “Possible” followed by the first number of the increasing sequence, else print “Not Possible“.

Approach : The idea is to take a substring from index 0 to any index i (i starting from 1) of the numeric string and convert it to long data type. Add 1 to it and convert the increased number back to string. Check if the next occurring substring is equal to the increased one. If yes, then carry on the procedure else increase the value of i and repeat the steps.

**Count of number of given string in 2D character array**

I can do it.

**Find minimum shift for longest common prefix:**

You are given two string str1 and str2 of same length. In a single shift you can rotate one string (str2) by 1 element such that its 1st element becomes the last and second one becomes the first like “abcd” will change to “bcda” after one shift operation. You have to find the minimum shift operation required to get common prefix of maximum length from str1 and str2.

If we will add second string at the end of itself that is str2 = str2 + str2 then there is no need of finding prefix for each shift separately. Now, after adding str2 to itself we have to only find the longest prefix of str1 present in str2 and the starting position of that prefix in str2 will give us the actual number of shift required. For finding longest prefix we can use KMP pattern search algorithm.

**Frequency Of A Substring In A String:**

I can do it.

**Count of occurrences of a “1(0+)1” pattern in a string**

The idea to solve this problem is to first find a ‘1’ and keep moving forward in the string and check as mentioned below:

If any character other than ‘0’ and ‘1’ is obtained then it means pattern is not valid. So we go on in the search of next ‘1’ from this index and repeat these steps again.

If a ‘1’ is seen, then check for the presence of ‘0’ at previous position to check the validity of sequence.

int countPattern(string str)

{

int len = str.size();

bool oneSeen = 0;

int count = 0; // Initialize result

for (int i = 0; i < len ; i++)

{

// if 1 encountered for first time

// set oneSeen to 1

if (str[i] == '1' && oneSeen == 0)

oneSeen = 1;

// Check if there is any other character

// other than '0' or '1'. If so then set

// oneSeen to 0 to search again for new

// pattern

if (str[i] != '0' && str[i] != '1')

oneSeen = 0;

// Check if encountered '1' forms a valid

// pattern as specified

if (str[i] == '1' && oneSeen == 1)

if (str[i - 1] == '0')

count++;

}

return count;

}

Note that, here, 1(0)+1 needs to be partially present.

**Match Expression where a single special character in pattern can match one or more characters:**

**Given two string, in which one is pattern (Pattern) and other is searching expression. Searching expression contains ‘#’.**

The # works in following way:

A # matches with one or more characters.

A # matches all characters before a pattern match is found. For example if pat = “A#B”, and text is “ACCBB”, then # would match only with “CC” and pattern is considered as not found.

Examples:

Input : str = "ABABABA"

pat = "A#B#A"

Output : yes

Input : str = "ABCCB"

pat = "A#B"

Output : yes

Input : str = "ABCABCCE"

pat = "A#C#"

Output : yes

Input : str = "ABCABCCE"

pat = "A#C"

Output : no

I get the idea. I can do it.

**Maximum Length Prefix Of One Strings That Occurs As Subsequence In Another:**

A simple solutions is to consider all prefixes on by one and check if current prefix of s[] is a subsequence of t[] or not. Finally return length of the largest prefix.

An efficient solution is based on the fact that to find a prefix of length n, we must first find the prefix of length n – 1 and then look for s[n-1] in t. Similarly, to find a prefix of length n – 1, we must first find the prefix of length n – 2 and then look for s[n – 2] and so on.

Thus, we keep a counter which stores the current length of prefix found. We initialize it with 0 and begin with the first letter of s and keep iterating over t to find the occurrence of the first letter. As soon as we encounter the first letter of s we we update the counter and look for second letter. We keep updating the counter and looking for next letter, until either the string s is found or there are no more letters in t.

Below is the implementation of this approach:

// C++ program to find maximum length prefix

// of one string occur as subsequence in another

// string.

#include<bits/stdc++.h>

using namespace std;

// Return the maximum length prefix which is

// subsequence.

int maxPrefix(char s[], char t[])

{

int count = 0;

// Iterating string T.

for (int i = 0; i < strlen(t); i++)

{

// If end of string S.

if (count == strlen(s))

break;

// If character match, increment counter.

if (t[i] == s[count])

count++;

}

return count;

}

// Driven Program

int main()

{

char S[] = "digger";

char T[] = "biggerdiagram";

cout << maxPrefix(S, T) << endl;

return 0;

}

**Replace All Occurrences Of AB with C:**

// Efficient C++ program to replace all occurrences

// of "AB" with "C"

#include <bits/stdc++.h>

void translate(char\* str)

{

int len = strlen(str);

if (len < 2)

return;

int i = 0; // Index in modified string

int j = 0; // Index in original string

// Traverse string

while (j < len-1)

{

// Replace occurrence of "AB" with "C"

if (str[j] == 'A' && str[j+1] == 'B')

{

// Increment j by 2

j = j + 2;

str[i++] = 'C';

continue;

}

str[i++] = str[j++];

}

if (j == len-1)

str[i++] = str[j];

// add a null character to terminate string

str[i] = '';

}

// Driver code

int main()

{

char str[] = "helloABworldABGfG";

translate(str);

printf("The modified string is :\n");

printf("%s", str);

return 0;

}