# KMP

The KMP stands for Knuth–Morris–Pratt algorithm, which is a string search algorithm. The original purpose for KMP is to use roughly O(N) time complexity to search a substring in a very long string.

There are two steps in KMP implementation, the first part is to build the kmp\_table on the substring to be looked for. The second part is to use the kmp\_table in the search.

There is several ways to implement KMP algorithm. Here I choose the one in Geeks for geeks. This is to calculate the longest prefix which is also the suffix of the substring at any position in the target string.

<https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

In this paper, the KMP table is built in this way, we start the position of index 1, which with pointer to index 0 which is the head of the string, we compare the characters, if they match, we advance the pointer, and record in the current index (starting as 1), if not match we step back to the pointer p - 1, if p < 0, we set the current index as 0.

The following are some examples of KMP table

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]

When we use the kmp table to compare the long string, if we advance to the next position in both source and target string, if not match, we reset the position in the target string to the kmp table in the previous position, if target position is 0 still not match, we move position in source string to next position.

## 28. Implement strStr()

Easy

Implement [strStr()](http://www.cplusplus.com/reference/cstring/strstr/).

Return the index of the first occurrence of needle in haystack, or **-1** if needle is not part of haystack.

**Example 1:**

**Input:** haystack = "hello", needle = "ll"

**Output:** 2

**Example 2:**

**Input:** haystack = "aaaaa", needle = "bba"

**Output:** -1

**Clarification:**

What should we return when needle is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when needle is an empty string. This is consistent to C's [strstr()](http://www.cplusplus.com/reference/cstring/strstr/) and Java's [indexOf()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#indexOf(java.lang.String)).

### **Analysis**

**Although this problem does not require KMP algorithm, but you can do it in KMP. Search the substring is the typical scenario for KMP.**

/// <summary>

/// Leet code #28. Implement strStr()

///

/// Easy

///

/// Implement strStr().

///

/// Return the index of the first occurrence of needle in haystack, or -1

/// if needle is not part of haystack.

///

/// Example 1:

///

/// Input: haystack = "hello", needle = "ll"

/// Output: 2

///

/// Example 2:

///

/// Input: haystack = "aaaaa", needle = "bba"

/// Output: -1

/// Clarification:

///

/// What should we return when needle is an empty string? This is a great

/// question to ask during an interview.

///

/// For the purpose of this problem, we will return 0 when needle is an empty

/// string. This is consistent to C's strstr() and Java's indexOf().

/// </summary>

int LeetCodeString::strStr(string haystack, string needle)

{

if (haystack.size() < needle.size()) return -1;

// Step 1: build kmp table

vector<int> kmp\_table(needle.size());

int i = 1;

int j = 0;

while (i < (int)needle.size())

{

if (needle[i] == needle[j])

{

j++;

kmp\_table[i] = j;

i++;

}

else if (j == 0)

{

kmp\_table[i] = 0;

i++;

}

else

{

j = kmp\_table[j - 1];

}

}

// Step 2: search substring

i = 0;

j = 0;

while (i < (int)haystack.size())

{

if (j == needle.size())

{

break;

}

else if (haystack[i] == needle[j])

{

i++;

j++;

}

else if (j == 0)

{

i++;

}

else

{

j = kmp\_table[j - 1];

}

}

if (j == needle.size())

{

return i - needle.size();

}

else

{

return -1;

}

}

## Check Repeated Substring

Another scenario to use KMP is to determine whether the string is has a repeated pattern, i.e. the string is built by a substring concatenating itself multiple times.

There is an example:

0 1 2 3 4 5

S : a b a b a b

Pi: 0 0 1 2 3 4

The last value of prefix table = 4.. Now If it is a repeated string than , It’s minimal length would be 2. (6(string length) – 4)

So you have to check if len % (len - last\_in\_pi) == 0. If yes, then len - last\_in\_pi is the length of the shortest repeated string (the period string).

## 459. Repeated Substring Pattern

Easy

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

**Example 1:**

**Input:** "abab"

**Output:** True

**Explanation:** It's the substring "ab" twice.

**Example 2:**

**Input:** "aba"

**Output:** False

**Example 3:**

**Input:** "abcabcabcabc"

**Output:** True

**Explanation:** It's the substring "abc" four times. (And the substring "abcabc" twice.)

### **Analysis**

**We can use KMP to determine the minimum repeated substring length, which should be length - prefix length.**

/// <summary>

/// Leet code #459. Repeated Substring Pattern

/// Given a non-empty string check if it can be constructed by taking a

/// substring of it and appending multiple copies of the substring together.

/// You may assume the given string consists of lowercase English letters

/// only and its length will not exceed 10000.

///

/// Example 1:

/// Input: "abab"

/// Output: True

///

/// Explanation: It's the substring "ab" twice.

/// Example 2:

/// Input: "aba"

/// Output: False

///

/// Example 3:

/// Input: "abcabcabcabc"

/// Output: True

/// Explanation: It's the substring "abc" four times. (And the substring

/// "abcabc" twice.)

/// </summary>

bool LeetCodeString::repeatedSubstringPattern(string s)

{

vector<int> kmp(s.size());

int i = 1;

int j = 0;

while (i < s.size())

{

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

{

j++;

kmp[i] = j;

i++;

}

else if (j == 0)

{

i++;

}

else

{

j = kmp[j - 1];

}

}

int last = kmp.back();

int size = s.size();

if ((last != 0) && (size % (size - last) == 0))

{

return true;

}

else

{

return false;

}

}

## 686. Repeated String Match

Easy

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it. If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times (“abcdabcdabcd”), B is a substring of it; and B is not a substring of A repeated two times ("abcdabcd").

**Note:**  
The length of A and B will be between 1 and 10000.

### **Analysis**

**This problem also does not require kmp, we can build repeat A until longer than B and do string find. But we use KMP for fun. We build KMP on the string B and repeat A to match it, when first B character match the second instance of A, we stop search.**

/// <summary>

/// Leet code #686. Repeated String Match

///

/// Given two strings A and B, find the minimum number of times A has to

/// be repeated such that B is a substring of it. If no such solution,

/// return -1.

///

/// For example, with A = "abcd" and B = "cdabcdab".

/// Return 3, because by repeating A three times ("abcdabcdabcd"),

/// B is a substring of it; and B is not a substring of A repeated

/// two times ("abcdabcd").

///

/// Note:

/// The length of A and B will be between 1 and 10000.

/// </summary>

int LeetCodeString::repeatedStringMatch(string A, string B)

{

vector<int> kmp(B.size());

int i = 1;

int j = 0;

while (i < (int)B.size())

{

if (B[i] == B[j])

{

j++;

kmp[i] = j;

i++;

}

else if (j == 0)

{

i++;

}

else

{

j = kmp[j - 1];

}

}

int result = 0;

i = 0;

j = 0;

while (j < (int)B.size())

{

if (i == A.size())

{

result++;

i = 0;

}

else if (A[i] == B[j])

{

i++;

j++;

}

else

{

if ((result - 1) \* (int)A.size() + i > j)

{

return -1;

}

if (j == 0)

{

i++;

}

else

{

j = kmp[j - 1];

}

}

}

return result + 1;

}

## 1316. Distinct Echo Substrings

Hard

Return the number of **distinct** non-empty substrings of text that can be written as the concatenation of some string with itself (i.e. it can be written as a + a where a is some string).

**Example 1:**

**Input:** text = "abcabcabc"

**Output:** 3

**Explanation:** The 3 substrings are "abcabc", "bcabca" and "cabcab".

**Example 2:**

**Input:** text = "leetcodeleetcode"

**Output:** 2

**Explanation:** The 2 substrings are "ee" and "leetcodeleetcode".

**Constraints:**

* 1 <= text.length <= 2000
* text has only lowercase English letters.

### **Analysis**

**There is a little bit confusion in this problem, basically the target is to look for a substring which repeat itself. So we can use half size to check if it is divisible by minimum repeated substring length.**

/// <summary>

/// Leet code #1316. Distinct Echo Substrings

///

/// Hard

///

/// Return the number of distinct non-empty substrings of text that can be

/// written as the concatenation of some string with itself.

///

/// Example 1:

/// Input: text = "abcabcabc"

/// Output: 3

/// Explanation: The 3 substrings are "abcabc", "bcabca" and "cabcab".

/// Example 2:

/// Input: text = "leetcodeleetcode"

/// Output: 2

/// Explanation: The 2 substrings are "ee" and "leetcodeleetcode".

///

/// Constraints:

/// 1. 1 <= text.length <= 2000

/// 2. text has only lowercase English letters.

/// </summary>

int LeetCodeString::distinctEchoSubstrings(string text)

{

int n = text.size();

unordered\_set<string> result\_set;

for (int s = 0; s < (int)text.size(); s++)

{

vector<int> kmp(n);

int i = 1;

int j = 0;

while (s + i < n)

{

if (text[s + i] == text[s + j])

{

j++;

kmp[i] = j;

i++;

// must be even length

if (i % 2 == 1) continue;

// if duplicate as half half, the remaining prefix length

// must be divisible to total length

if ((i / 2) % (i - j) == 0)

{

result\_set.insert(text.substr(s, i));

}

}

else if (j == 0)

{

i++;

}

else

{

j = kmp[j - 1];

}

}

}

return result\_set.size();

}