# KMP and Rabin Karp

## 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.

## Rabin Karp

Rabin karp is another algorithm for searching the substring, the idea of Rabin karp is to calculate the substring checksum and then during the search check the checksum first, if they match then check the substring.

It is tricky to calculate the checksum, you can not simply add them up because the duplication will be too many, so you should give each position a pow, normally we use 26, but at the same time such number will be too big and be beyond range of 64 bits, so you need to do a mode, we can choose a large prime number, 1000000007 is a good choice.

## 28. Implement strStr()

Easy

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

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/" \t "_blank) and Java's [indexOf()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \l "indexOf(java.lang.String)" \t "_blank).

### **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;

}

## 1071. Greatest Common Divisor of Strings

Easy

For strings S and T, we say "T divides S" if and only if S = T + ... + T  (T concatenated with itself 1 or more times)

Return the largest string X such that X divides str1 and X divides str2.

**Example 1:**

**Input:** str1 = "ABCABC", str2 = "ABC"

**Output:** "ABC"

**Example 2:**

**Input:** str1 = "ABABAB", str2 = "ABAB"

**Output:** "AB"

**Example 3:**

**Input:** str1 = "LEET", str2 = "CODE"

**Output:** ""

**Note:**

1. 1 <= str1.length <= 1000
2. 1 <= str2.length <= 1000
3. str1[i] and str2[i] are English uppercase letters.

### **Analysis**

**First we use KMP to get the shortest repeated string for each string, then if they are not same, we do not have common divisor, if they are same, we will divide each string length by the common repeated string length, the calculate common divisor. Then repeat the common repeated string.**

/// <summary>

/// Leet code #1071. Greatest Common Divisor of Strings

/// </summary>

string LeetCodeString::getRepeatedOfStrings(string str)

{

size\_t size = str.size();

vector<int> lps(size);

int len = 0;

size\_t i = 1;

while (i < str.size())

{

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

{

len++;

lps[i] = len;

i++;

}

else if (len > 0)

{

len = lps[len - 1];

}

else

{

lps[i] = 0;

i++;

}

}

if ((len > 0) && (size % (size-len) == 0))

{

return str.substr(0, size-len);

}

else

{

return str;

}

}

/// <summary>

/// Leet code #1071. Greatest Common Divisor of Strings

///

/// For strings S and T, we say "T divides S" if and only if S = T + ... + T

/// (T concatenated with itself 1 or more times)

///

/// Return the largest string X such that X divides str1 and X divides str2.

///

/// Example 1:

///

/// Input: str1 = "ABCABC", str2 = "ABC"

/// Output: "ABC"

///

/// Example 2:

///

/// Input: str1 = "ABABAB", str2 = "ABAB"

/// Output: "AB"

///

/// Example 3:

///

/// Input: str1 = "LEET", str2 = "CODE"

/// Output: ""

///

/// Note:

///

/// 1. 1 <= str1.length <= 1000

/// 2. 1 <= str2.length <= 1000

/// 3. str1[i] and str2[i] are English uppercase letters.

/// </summary>

string LeetCodeString::gcdOfStrings(string str1, string str2)

{

string re1 = getRepeatedOfStrings(str1);

string re2 = getRepeatedOfStrings(str2);

if (re1 != re2 || re1 == "") return "";

size\_t a = str1.size() / re1.size();

size\_t b = str2.size() / re2.size();

while (a \* b != 0)

{

if (a < b) swap(a, b);

a = a % b;

}

string result;

for (size\_t i = 0; i < b; i++)

{

result.append(re1);

}

return result;

}

## 214. Shortest Palindrome

Hard

Given a string ***s***, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

**Example 1:**

**Input:** "aacecaaa"

**Output:** "aaacecaaa"

**Example 2:**

**Input:** "abcd"

**Output:** "dcbabcd"

### **Analysis**

**We can use KMP to track the string and match from back to front until two pointers met, the back position deduct the front position is the characters we want to patch.**

/// <summary>

/// Leet code #214. Shortest Palindrome

///

/// Given a string S, you are allowed to convert it to a palindrome by

/// adding characters in front of it. Find and return the shortest palindrome

/// you can find by performing this transformation.

///

/// For example:

/// Given "aacecaaa", return "aaacecaaa".

/// Given "abcd", return "dcbabcd".

/// Given an array of n positive integers and a positive integer s,

/// </summary>

string LeetCodeString::shortestPalindrome(string s)

{

string result;

int n = s.size();

vector<int> kmp(n);

int i = 1;

int j = 0;

while (i < n)

{

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

{

j++;

kmp[i] = j;

i++;

}

else if (j == 0)

{

i++;

}

else

{

j = kmp[j - 1];

}

}

int s\_i = 0, r\_i = 0;

while (s\_i + r\_i < n)

{

if (s[s\_i] == s[n-1-r\_i])

{

s\_i++;

r\_i++;

}

else

{

if (s\_i > 0)

{

s\_i = kmp[s\_i-1];

}

else r\_i++;

}

}

r\_i -= s\_i;

for (int i = 0; i < r\_i; i++)

{

result.push\_back(s[n-1-i]);

}

result.append(s);

return result;

}

## 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, 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();

}

## 1044. Longest Duplicate Substring

Hard

Given a string S, consider all *duplicated substrings*: (contiguous) substrings of S that occur 2 or more times.  (The occurrences may overlap.)

Return **any** duplicated substring that has the longest possible length.  (If S does not have a duplicated substring, the answer is "".)

**Example 1:**

**Input:** "banana"

**Output:** "ana"

**Example 2:**

**Input:** "abcd"

**Output:** ""

**Note:**

1. 2 <= S.length <= 10^5
2. S consists of lowercase English letters.

### **Analysis**

**For this problem, we cannot use KMP, because KMP will lead to O(n^2) cause TLE. We should use Rabin-Karp, first we guess the longest duplicate substring as binary search, then given the longest duplicate substring, we calculate the checksum for the substring and check everything which has a duplicate hit count.**

/// <summary>

/// Leet code #1044. Longest Duplicate Substring

///

/// Given a string S, consider all duplicated substrings: (contiguous)

/// substrings of S that occur 2 or more times. (The occurrences may overlap.)

///

/// Return any duplicated substring that has the longest possible length.

/// (If S does not have a duplicated substring, the answer is "".)

///

/// Example 1:

///

/// Input: "banana"

/// Output: "ana"

///

/// Example 2:

///

/// Input: "abcd"

/// Output: ""

///

/// Note:

///

/// 1. 2 <= S.length <= 10^5

/// 2. S consists of lowercase English letters.

/// </summary>

string LeetCodeString::longestDupSubstring(string S)

{

unsigned long long M = 1000000007;

int first = 1;

int last = S.size();

string result;

while (first < last)

{

int middle = first + (last - first) / 2;

unsigned long long pow = 1;

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

{

pow = pow \* 26 % M;

}

bool match = false;

unordered\_map<int, vector<int>> hash;

unsigned long long sum = 0;

for (int i = 0; i < (int)S.size(); i++)

{

if (i < middle - 1)

{

sum = (sum \* 26 %M + S[i] - 'a') % M;

}

else

{

if (i == middle - 1)

{

sum = (sum \* 26 % M + S[i] - 'a') % M;

}

else

{

sum = (sum \* 26 % M + S[i] - 'a' - (((unsigned long long)S[i - middle] - 'a') \* pow % M) + M) % M;

}

int key = (int)sum;

if (hash.count(key) > 0)

{

match = false;

string target = S.substr(i - middle + 1, middle);

for (size\_t j = 0; j < hash[key].size(); j++)

{

string source = S.substr(hash[key][j], middle);

if (source == target)

{

match = true;

break;

}

}

if (match == true)

{

result = target;

break;

}

}

hash[key].push\_back(i - middle + 1);

}

}

if (match) first = middle + 1;

else last = middle;

}

return result;

}

## 972. Equal Rational Numbers

Hard

Given two strings S and T, each of which represents a non-negative rational number, return **True** if and only if they represent the same number. The strings may use parentheses to denote the repeating part of the rational number.

In general a rational number can be represented using up to three parts: an integer part, a non-repeating part, and a repeating part. The number will be represented in one of the following three ways:

* <IntegerPart> (e.g. 0, 12, 123)
* <IntegerPart><.><NonRepeatingPart>  (e.g. 0.5, 1., 2.12, 2.0001)
* <IntegerPart><.><NonRepeatingPart><(><RepeatingPart><)> (e.g. 0.1(6), 0.9(9), 0.00(1212))

The repeating portion of a decimal expansion is conventionally denoted within a pair of round brackets.  For example:

1 / 6 = 0.16666666... = 0.1(6) = 0.1666(6) = 0.166(66)

Both 0.1(6) or 0.1666(6) or 0.166(66) are correct representations of 1 / 6.

**Example 1:**

**Input:** S = "0.(52)", T = "0.5(25)"

**Output:** true

**Explanation:**

Because "0.(52)" represents 0.52525252..., and "0.5(25)" represents 0.52525252525..... , the strings represent the same number.

**Example 2:**

**Input:** S = "0.1666(6)", T = "0.166(66)"

**Output:** true

**Example 3:**

**Input:** S = "0.9(9)", T = "1."

**Output:** true

**Explanation:**

"0.9(9)" represents 0.999999999... repeated forever, which equals 1. [[See this link for an explanation.](https://en.wikipedia.org/wiki/0.999...)]

"1." represents the number 1, which is formed correctly: (IntegerPart) = "1" and (NonRepeatingPart) = "".

**Note:**

1. Each part consists only of digits.
2. The <IntegerPart> will not begin with 2 or more zeros.  (There is no other restriction on the digits of each part.)
3. 1 <= <IntegerPart>.length <= 4
4. 0 <= <NonRepeatingPart>.length <= 4
5. 1 <= <RepeatingPart>.length <= 4

### **Analysis**

**For this problem one of focus is to normalize the repeating portion of a decimal, because the repeating pattern starting from the position zero, we can use another simple form to calculate it, we simply rotate the character from left most to the right most, until we have the same string again, we will see the duplicated string (the shift length should be divisible against the total length).**

/// <summary>

/// Leet code #972. Equal Rational Numbers

/// </summary>

string LeetCodeString::parseRationalInteger(string S, size\_t &index)

{

string result;

while (index < S.size())

{

if (S[index] == '.') break;

result.push\_back(S[index]);

index++;

}

if (result.empty()) result.push\_back('0');

return result;

}

/// <summary>

/// Leet code #972. Equal Rational Numbers

/// </summary>

string LeetCodeString::parseRationalDecimal(string S, size\_t &index)

{

string non\_repeating\_part;

string repeating\_part;

bool is\_repeat = false;

while (index < S.size())

{

if (S[index] == '.')

{

}

else if (S[index] == '(')

{

is\_repeat = true;

}

else if (S[index] == ')')

{

is\_repeat = false;

}

else if (is\_repeat)

{

repeating\_part.push\_back(S[index]);

}

else

{

non\_repeating\_part.push\_back(S[index]);

}

index++;

}

// shift non-repeating part to repeating part

while (!non\_repeating\_part.empty() && !repeating\_part.empty() &&

(non\_repeating\_part.back() == repeating\_part.back()))

{

repeating\_part.insert(repeating\_part.begin(), non\_repeating\_part.back());

non\_repeating\_part.pop\_back();

repeating\_part.pop\_back();

}

// shrink repeating part

for (int i = 1; i < (int)repeating\_part.size(); i++)

{

string next\_str = repeating\_part.substr(i) + repeating\_part.substr(0, i);

if (repeating\_part == next\_str)

{

repeating\_part = repeating\_part.substr(0, i);

break;

}

}

if (repeating\_part == "0") repeating\_part = "";

// (9) theory

if (repeating\_part == "9" && !non\_repeating\_part.empty())

{

non\_repeating\_part[non\_repeating\_part.size() - 1]++;

repeating\_part = "";

}

if (repeating\_part.empty())

{

while (!non\_repeating\_part.empty() && non\_repeating\_part.back() == '0')

{

non\_repeating\_part.pop\_back();

}

}

string result = non\_repeating\_part;

if (!repeating\_part.empty()) result += "(" + repeating\_part + ")";

return result;

}

/// <summary>

/// Leet code #972. Equal Rational Numbers

///

/// Given two strings S and T, each of which represents a non-negative

/// rational number, return True if and only if they represent the same

/// number. The strings may use parentheses to denote the repeating part

/// of the rational number.

///

/// In general a rational number can be represented using up to three parts:

/// an integer part, a non-repeating part, and a repeating part. The number

/// will be represented in one of the following three ways:

///

/// <IntegerPart> (e.g. 0, 12, 123)

/// <IntegerPart><.><NonRepeatingPart> (e.g. 0.5, 1., 2.12, 2.0001)

/// <IntegerPart><.><NonRepeatingPart><(><RepeatingPart><)> (e.g. 0.1(6),

/// 0.9(9), 0.00(1212))

/// The repeating portion of a decimal expansion is conventionally denoted

/// within a pair of round brackets. For example:

///

/// 1 / 6 = 0.16666666... = 0.1(6) = 0.1666(6) = 0.166(66)

///

/// Both 0.1(6) or 0.1666(6) or 0.166(66) are correct representations of 1 / 6.

///

/// Example 1:

/// Input: S = "0.(52)", T = "0.5(25)"

/// Output: true

/// Explanation:

/// Because "0.(52)" represents 0.52525252..., and "0.5(25)" represents

/// 0.52525252525..... , the strings represent the same number.

///

/// Example 2:

/// Input: S = "0.1666(6)", T = "0.166(66)"

/// Output: true

///

/// Example 3:

///

/// Input: S = "0.9(9)", T = "1."

/// Output: true

/// Explanation:

/// "0.9(9)" represents 0.999999999... repeated forever, which equals 1.

/// [See this link for an explanation.]

/// "1." represents the number 1, which is formed correctly:

/// (IntegerPart) = "1" and (NonRepeatingPart) = "".

///

/// Note:

///

/// 1. Each part consists only of digits.

/// 2. The <IntegerPart> will not begin with 2 or more zeros. (

/// There is no other restriction on the digits of each part.)

/// 3. 1 <= <IntegerPart>.length <= 4

/// 4. 0 <= <NonRepeatingPart>.length <= 4

/// 5. 1 <= <RepeatingPart>.length <= 4

/// </summary>

bool LeetCodeString::isRationalEqual(string S, string T)

{

size\_t index = 0;

string integer1 = parseRationalInteger(S, index);

string decimal1 = parseRationalDecimal(S, index);

if (decimal1 == "(9)")

{

integer1 = to\_string(stoi(integer1) + 1);

decimal1 = "";

}

index = 0;

string integer2 = parseRationalInteger(T, index);

string decimal2 = parseRationalDecimal(T, index);

if (decimal2 == "(9)")

{

integer2 = to\_string(stoi(integer2) + 1);

decimal2 = "";

}

if (integer1 == integer2 && decimal1 == decimal2)

{

return true;

}

else

{

return false;

}

}

## KMP vs Rabin-Karp

In some case, KMP and Rabin Karp are exchangeable, Rabin Karp is more useful when you have a known length of substring to search.

## 187. Repeated DNA Sequences

Medium

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

**Example:**

**Input:** s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

**Output:** ["AAAAACCCCC", "CCCCCAAAAA"]

### **Analysis**

**The DNA only contains 4 different characters, so we can simplify the Karp Rabin as a 32-bit integer. Each character use 2-bit, the length of 10 characters can be held in a integer.**

/// <summary>

/// Leet code #187. Repeated DNA Sequences

///

/// All DNA is composed of a series of nucleotides abbreviated

/// as A, C, G, and T, for example: "ACGAATTCCG".

/// When studying DNA, it is sometimes useful to identify repeated

/// sequences within the DNA.

/// Write a function to find all the 10-letter-long sequences

/// (substrings) that occur more than once in a DNA molecule.

///

/// For example,

/// Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

/// Return:

/// ["AAAAACCCCC", "CCCCCAAAAA"].

/// </summary>

vector<string> LeetCode::findRepeatedDnaSequences(string s)

{

vector<string> result;

if (s.size() < 10) return result;

unordered\_map<char, int> char\_map = { { 'A', 0 },{ 'C', 1 },{ 'G', 2 },{ 'T', 3 } };

unordered\_map<int, int> pos\_map;

unordered\_set<int> visited;

uint32\_t string\_code = 0;

for (size\_t i = 0; i < 10; i++)

{

string\_code = ((string\_code << 2) & 0xfffff) + char\_map[s[i]];

}

pos\_map[string\_code] = 0;

for (size\_t i = 10; i < s.size(); i++)

{

string\_code = ((string\_code << 2) & 0xfffff) + char\_map[s[i]];

if (pos\_map.find(string\_code) != pos\_map.end())

{

if (visited.find(string\_code) == visited.end())

{

result.push\_back(s.substr(pos\_map[string\_code], 10));

visited.insert(string\_code);

}

}

else

{

pos\_map[string\_code] = i - 9;

}

}

return result;

}