***Задача поиска подстроки в строке.***

1. ***Алгоритм Рабина-Карпа.***

Рассмотрим задачу поиска *подстроки в строке*. Существует несколько подходов и алгоритмов ее решения, рассмотрим *2* из них.

Пусть нам дана строка символов из некоторого *алфавита* . Длина строки – *n*. И задана подстрока, *шаблон*, из этого же алфавита длиной *m.* Требуется указать *индекс* первого вхождения подстроки в строку ( если такой случай существует).

При решении такой задачи *«в лоб»* нам придется посимвольно сравнивать *шаблон* со всеми позициями в тексте, что дает оценку *O(nm).*

Рассмотрим алгоритмы, которые решают эту задачу за *линейное время*.

В алгоритме *Рабина-Карпа* происхдит сравнение *хеш-функций* *шаблона* и подстрок длины *m строки.*

Этапы алгоритма выглядят так:

* + - 1. Вычисляем *хеш шаблона,* который имеет длину *m.*
      2. Вычисляем *хеш* подстроки в строке, начиная с индекса 0 до *m-1*.
      3. Сравниваем *хеш шаблона* с *хешем* подстроки.
      4. Если *хеши* совпадают, то значит мы нашли скорее всего вхождение (но если возможны коллизии , производим посимвольное сравнение *шаблона* с найденной подстрокой).
      5. Если *хеши* не совпадают, сдвигаемсяпо строке на один *индекс* вправо (на один символ) и вычисляем *хеш* для новой подстроки длиной *m.*
      6. Повторяем пункты *4*, *5* до тех пор пока не будут обработаны все символовстроки.

*Для быстрого вычисления хеша алгоритм использует полиномиальный хеш, который позволяет вычислить за постоянное время следующий хеш подстроки в строке из предыдущего, независимо от длины подстроки.*

Для подстроки длиной *m полиномиальный хеш* в общем виде будет иметь вид:

, *(1)*

где , – символы подстроки, которые трактуются как числа, *p* – целое положительное число (желательно *простое* для уменьшения *коллизий* при модульных операциях), большее мощности нашего алфавита *А*.

Фактически, мы предстваляем подстроку, для которой вычисляем *хеш*, числом по основанию *p*.

Теперь построим *хеш* для подстроки длиной *4* и будем идти с начала строки.

. *(2)*

Предположим, построенный *хеш* не совпал с *хешем* шаблона длинной *4*, в этом случае мы сдвигаемся на символ по строке вправо и строим новый *хеш* для новой подстроки длиной *m*:

.

А теперь сравним эти два *хеша*.

Нам не надо полностью строить второй *хеш*, егомы можем получить из первого, отняв вначале от первого , затем умножив полученный хеш на *p* и, наконец прибавив к нему . Обозначим через исходний *хеш*, тогда алгоритм построения последующего, нового, хеша может выглядеть так:

1. .
2. .
3. .

В общем случае вычисляем новый хеш из предыдущего по формуле:

, *(3)*

где – предыдущий стартовый символ­ (*previous*) ,

– новый символ (*new*).

Поскольку число *m* может быть достаточно большим, вычисление *хеша* по формулам *1–3* может приводить к *целочисленному переполнению*. Для устранения таких ситуаций используются вычисления *хеша* по *модулю.*

И, как уже говорилось, при совпадении *хешов* *шаблона* и подстроки для исключения *коллизий*, сравниваем их *посимвольно*.

Рассмотрим алгоритм *Рабина-Карпа* в применении к задаче

**28. Implement strStr()** ресурса <https://leetcode.com/>.

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.

**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()](_blank).

**Example 1:**

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

**Output:** 2

**Example 2:**

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

**Output:** -1

**Example 3:**

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

**Output:** 0

Ниже приведена реализация алгоритма *Рабина-Карпа* на ***java.***

private static int strStr(String haystack, String needle) {  
 if (needle.equals("")) {  
 return 0;  
 }  
 if (haystack.length() < needle.length()) {  
 return -1;  
 }  
 char[] needleChar = needle.toCharArray();  
 char[] hayStackChar = haystack.toCharArray();  
  
 int p = 29;  
 int module = Integer.*MAX\_VALUE*;  
 int needleHash = needleChar[0] - 'a' + 1;  
 int hayHash = hayStackChar[0] - 'a' + 1;  
 int mp = 1;  
  
 for (int i = 1; i < needleChar.length; i++) {  
 needleHash = (((needleHash \* p) & module) + needleChar[i] - 'a' + 1) & module;  
 hayHash = (((hayHash \* p) & module) + hayStackChar[i] - 'a' + 1) & module;  
 mp = (mp \* p) & module;  
 }  
 if (hayHash == needleHash) {  
 return 0;  
 }  
 int end = 0;  
 for (int j = 1; j <= hayStackChar.length - needleChar.length; j++) {  
 hayHash -= ((hayStackChar[j - 1] - 'a' + 1) \* mp) & module;  
 hayHash = (hayHash \* p) & module;  
 end = j + needleChar.length - 1;  
 hayHash = (hayHash + hayStackChar[end] - 'a' + 1) & module;  
  
 if (hayHash == needleHash) {  
 boolean result = true;  
 for (int i = 0, k = j; i < needleChar.length; i++, k++) {  
 if (needleChar[i] != hayStackChar[k]) {  
 result = false;  
 break;  
 }  
 }  
 if (result) {  
 return j;  
 }  
 }  
 }  
 return -1;  
}

1. В данной реализации вместо операции *взятия по модулю* используется операция *поразрядного «и»* с числом , в двоичном виде представляющее собой *31* разрядное двоичное число, состоящие из одних единиц. *Поразрядное* *«и» целого положительного числа* c таким двоичным числом дает нам остаток по модулю .  
Проверим это утверждение на примере.  
Найдем остаток от деления чисел *21* и *26* по модулю *.* Остатки будут *5* и *2* соответственно. Берем *поразрядное «и»* с числом . Взять остаток по модулю , значит отбросить в двоичном представлении числа все разряды, соответствующие степени : . Выделить единицы в остатке и позволяет операция *поразрядного «и».*21: 00010101 26: 00011010  
 00000111 00000111  
 00000101 (число 5) 00000010 (число *2*)

И к еще одному примеру, получить остаток числа *123* по модулю *10*, значит отбросить разряды :  
 .  
Остаток *3.*

1. Один из способов вычисления *полиномиального* *хеша* подстроки может быть следующим:  
   • вычисляем *хеш* *первого* символа – он равен самому первому символу : ;  
   • далее вычисляем *хеш* первых *двух* символов , . Подставив значение , мы бы получили *полином первой степени*: .  
   • далее вычисляем *хеш* первых *трех* символов, .  
   подставляя значения и , мы получаем *полиномиальный хеш*, *полином второй* *степени*: ;  
   и т. д. для всех символов для которых мы хотим построить *хеш*. Таким образом, текущий *хеш* мы получаем из предыдущего, этот способ позволяет вычислять *полиноминальный хеш* не вычисляя отдельно степени *p*.

Такая схема вычисления значений полинома называется *схемой Горнера*.

***Вопросы по обработке алгоритма Рабина-Карпа ( реализация java).***

*(Ответы в письменном виде)*

1. Какова временная сложнось алгоритма?
2. Какова *пространственная* (*емкостная*) сложность алгоритма?
3. Что такое *коллизии* и какие методы позволяют их избежать при данной реализации алгоритма?
4. Что такое схема *Горнера* и для чего она используется, есть ли она в данной реализации?
5. В каких задачах применяется алгоритм *Рабина-Карпа*?
6. \*Реализовать один из предложенных алгоритмов на ресурсе https://leetcode.com.
7. ***Алгоритм Кнута-Мориса-Пратта***

public int strStr(String haystack, String needle) {  
 if (needle.equals("")) {  
 return 0;  
 }  
 if (haystack.length() < needle.length()){  
 return -1;  
 }  
 char[] needleChar = needle.toCharArray();  
 char[] hayStackChar = haystack.toCharArray();  
  
 int[] prf = new int[needle.length()];  
 prf[0] = 0;  
 int len = 0;  
 int index = 1;  
 while (index < prf.length) {  
  
 if (needleChar[index] == needleChar[len]) {  
 len++;  
 prf[index] = len;  
 index++;  
 } else {  
 if (len != 0) {  
 len = prf[len - 1];  
 } else {  
 prf[index] = len;  
 index++;  
 }  
 }  
 }  
  
 int i = 0;  
 int j = 0;  
 while (i < hayStackChar.length) {  
  
 if (hayStackChar[i] == needleChar[j]) {  
 i++;  
 j++;  
 if (j >= needleChar.length) {  
 return i - j;  
 }  
 } else {  
  
 if (j != 0) {  
 j = prf[j - 1];  
 } else {  
 i++;  
 }  
 }  
 }  
 return -1;  
}