**Поиск подстроки в строке. Алгоритм грубой силы и использование хэшей**.

**Постановка задачи**

Дана строка *S* длины *п* и строка *Т* длины *m*. В дальнейшем, не оговаривая каждый раз отдельно, будем предполагать, что *n≥m*. Будем нумеровать символы строк начиная с 1. Обозначим через *S* [*i*.. j], 1 ≤*i ≤ j* ≤ n, подстроку строки S, начинающуюся в *i*-й позиции и заканчивающуюся в *j* -й позиции. Требуется найти все вхождения строки *Т* в строку S, т. е. указать все позиции строки S, начиная с которых читается строка *Т.* Множество вхождений, которое надо построить, формально описывается так: *М* = {*i* |1 ≤ *i* ≤ *п-т+1,* S[*i*.. *i+m-1*] = Т}.

Например, пусть *S = abcabdecab*, *Т =* cab, тогда *М* = {3; 8}. Или такой пример: пусть *S = abababacaba*, *Т* = *aba*, тогда *М* = {1; 3; 5; 9}. Таким образом, мы допускаем ≪перекрытие≫ — ситуацию, когда расстояние между двумя соседними вхождениями меньше длины строки *Т.*

**Метод решения: алгоритм грубой силы**

Наиболее очевидным и простым является алгоритм грубой силы (англ. brute force). Он состоит в последовательном посимвольном сравнении строк.

Начало шаблона *Т* совмещается с началом строки *S.* Проверка начинается с первого символа шаблона Т. Если символы совпали, то производится сравнение следующего символа и т. д. Возможна одна и только одна из двух ситуаций:

1. Если все символы шаблона совпали с соответствующими символами подстроки, то подстрока найдена; надо 1) сделать соответствующую запись об этом вхождении и 2) сдвинуть шаблон на 1 символ вправо и начать новый поиск.
2. Если какой-то символ шаблона не совпал с соответствующим символом строки, то надо сдвинуть шаблон на 1 символ вправо и начать новый поиск.

Проверку осуществлять до тех пор, пока есть куда сдвигать шаблон, т. е. пока правый конец шаблона не выйдет за правый край строки *S* в результате очередного сдвига.

Внимательное рассмотрение этого описания позволяет увидеть, что сдвиг шаблона на 1 символ вправо производится независимо от того, все символы шаблона совпали с соответствующими символами строки *S* или нет.

****

**Метод решения: использование хэшей.**

Каждому символу, входящему в строку, можно поставить в соответствие число, например, это может быть код символа в таблице ASCII (разумеется, если символ принадлежит

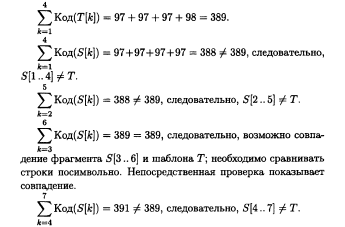
этой таблице). Так, например, для английских букв Код(А)=65, Koд(Z)=90, Код(а)=97, Код(z)=122, для русских букв Код(А)=192, Код(Я)=223, Код(а)=224, Код(я)=255.

Утверждение 1. Если две строки *S1* и *S2* совпадают, то сумма кодов символов строки *S1*совпадает с суммой кодов символов строки *S2.*

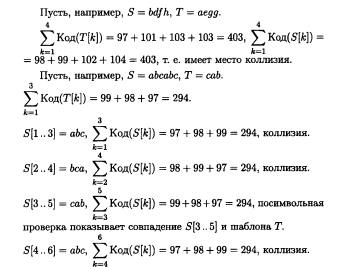
Утверждение 2. Если сумма кодов символов строки *S1* не совпадает с суммой кодов символов строки *S2* , то строки *S1* и *S2* не совпадают.

Если использовать это простое наблюдение при поиске подстроки в строке, то можно избежать целого ряда проверок, которые заведомо покажут несоответствие шаблона и фрагмента строки.

Пусть, например, *S=аааааЬс*, *Т =аааЬ.*



Разумеется, возможна ситуация, когда суммы кодов символов совпадут, а сами строки будут отличаться. Такая ситуация называется *коллизией.*



Для дальнейшего нам потребуется понятие хэш-функции.

Хэш-функцией *h*(s), определенной на строке s, мы будем называть произвольную ограниченную целочисленную функцию этой строки. Сумма ASCII кодов символов строки являет собой пример простейшей хэш-функции. Часто также рассматривают

сумму кодов символов строки по модулю большого числа.

Приступим к построению алгоритма поиска подстроки в строке с использованием хэшей. Главное отличие от метода brute force состоит в том, что на шаге, где проводится сопоставление *S*[ *i*.. *i+m-1*] с Т, сперва вычисляется хэш *h(S[i*.. *і+m-1*]), который потом сравнивается с хэшем h(Т), и только в случае равенства хэшей осуществляется посимвольная проверка, как в brute force. Если хэш шаблона можно вычислить однажды, а именно до начала всех проверок, то хэши фрагментов строки *S* необходимо вычислять отдельно для каждого исследуемого фрагмента *S*[ *i*.. *i+m-1*], 1 ≤ *i ≤п*-*т*+1, т. е. *п - т +* 1 раз. Возможны различные подходы к вычислению хэшей фрагментов *S*[ *i*.. *i+m-1*]; сравним два способа.

Первый способ вычисления:

*h(S*[ *i*.. *i+m-1*]), 1 ≤ *i ≤п*-*т*+1.

hs=0

k=0

Пока (k<=m-l)

Начало\_цикла

hs+=Koд\_ символа (S[i+k])

k++

Конец\_цикла

Второй способ вычисления:

*h(S*[ *i*.. *i+m-1*]), 1 < *i ≤п*-*т*+1, при этом предполагается, что *h(S*[ *1*.. *m*])будет вычислено заранее.

hs=hs + Код\_символа(S[i+m-1]) - Код\_символа(S[i-1])

Если в первом способе хэш фрагмента каждый раз вычисляется с нуля, то во втором используются ранее полученные результаты, чтобы после ≪небольшой≫ поправки получить требуемое. Эта оптимизация, называемая *рекуррентное вычисление*,

дает особенно ощутимый эффект, если длина шаблона *т* велика. Вычисление хэш-функции должно производиться быстро, независимо от *т.* Собственно только в этом случае использование хэшей дает преимущество над методом грубой силы!

Наряду с рассмотренной ранее хэш-функцией, вычисляемой как сумма кодов символов, можно предложить и другие.

Например, сумма квадратов кодов также допускает рекуррентное

вычисление:

*h2 (S*[ *i*.. *i+m-1*]) = *h2 (S*[ *i-1*.. *i+m-2*]) + Код2(S[i+m-1]) – Код2 (S[i-1])

Возникает естественный вопрос — какую предпочесть? При прочих равных условиях отдавать предпочтение надо той хэш- функции, которая дает меньше коллизий.

Ранее были рассмотрены примеры строк, доставляющих коллизию хэш-функции типа ≪сумма кодов символов≫:

*S=bdfh*, *Т=аеgg* и *S* = *abc*, *Т* = *cab*. Проверим, имеет ли место коллизия для хэш-функции типа ≪сумма квадратов кодов символов≫ *h2 (bdfh) =* 40828 *≠ h2 (aegg)* = 40625,

*h2 (abc)* = *h2* (*cab*). То есть в первом случае нам удалось избежать коллизии, а во втором нет, так как строки *abc* и *cab* отличаются только порядком букв, а сумма квадратов от порядка слагаемых не зависит.

Нельзя ли предложить хэш-функцию, которая дает различные значения для строк, отличающихся лишь порядком символов, как то: *abc* и *cab*? Рабином и Карпом была предложена функция ,где p – длина строки str*.* Эта функция допускает

рекуррентное вычисление

Практика показывает, что на среднестатистическом тексте хэш-функция Рабина-Карпа дает меньше коллизий, чем хэш-функция типа ≪сумма кодов≫ или ≪сумма квадратов кодов символов≫.

В методических целях еще раз назовем свойства, которым должна удовлетворять ≪хорошая≫ хэш-функция.

1. Возможность рекуррентного вычисления; как следствие, высокая скорость нахождения значений функции, не зависящая от длины строки.

2. Низкая вероятность коллизий.

**Задание.**

В текстовом файле input.txt записана на первой строке - строка *S*, на второй строке - строка *Т*. Требуется найти все вхождения строки *Т* в строку S, т. е. указать все позиции строки S, начиная с которых читается строка *Т:*

1. алгоритмом грубой силы;
2. использование хэш-функции - сумма кодов символов;
3. использование хэш-функции Рабина и Карпа.

(Обратите внимание, значения хэш-функций совпали, это не значит, что строки совпали, т.е. надо проверить посимвольно совпадение строк.)

Попробовать на примере:

S: Весёлый, насмешливый хохот. | Громкий раскатистый хохот. | Оглушительный, заразительный хохот. | Дьявольский, злобный, жуткий, дикий хохот. | Взрыв общего, дружного хохота. | Закатываться, заливаться хохотом. | Гости с хохотом вывалились на улицу. | Хохот стих. | Со всех сторон поднялся хохот. хохохохоттт 265593138170205224293304

T: хохот

S: Вся природа радовалась вместе со мной. автомобиль  Имеет смысл поспешить с решением. Тебе уже не восемнадцать. За две недели пребывания в Таормине я значительно продвинулась в итальянском языке и без труда объяснялась с этим выдающимся макаронником. автомобиль с лёгкой запинкой произнёс он. 404 автомобиль

T: автомобиль

**Алгоритм грубой силы**

fso = new ActiveXObject ("Scripting.FileSystemObject");

fh = fso.OpenTextFile("d:\\11.txt");

S = fh.Readline();

T = fh.Readline();

fh.Close();

(node js

let fs = require('fs');

inText = fs.readFileSync('output.txt');

line = str.split('\n') – разбиваем файл по разделителям строк, получаем массив

S = line[0];

T = line[1];

)

n=длина S

m=длина T

i=0

Пока (i<=n-m)

Начало.цикла

j=0

Пока (S[i+j]==T[j] и j<m)

Начало.цикла

j++

Конец\_цикла

Если (j==m)

Начало.если

Вывести (i+1)

Конец\_если

i++

Конец\_цикла

T[j] означает рассматриваем в строке T символ на позиции j. Чтобы получить символ, используйте вызов charAt(позиция), например str.charAt(0) (первый символ имеет позицию 0).

**Использование хэшей.**

Хэш-функция - сумма кодов символов строки.

S = fh.Readline();

T = fh.Readline();

n=длина S

m=длина T

ht=0

hs=0

k=0

Пока (k<=m-1)

Начало\_цикла

ht+=Koд\_ символа (T[k])

hs+=Koд\_ символа (S[k])

k++

Конец\_цикла

і=0

Пока (і<=n-m)

Начало.цикла

Если (hs==ht)

Начало.если

j=0

Пока (S[і+j]==T[j] и j<m)

Начало.цикла

j++

Конец\_цикла

Если (j==m)

Начало.если

Вывести (і+1)

Конец\_если

Конец\_если

і++

Если (і<= n-m)

Начало.если

hs=hs + Код\_символа(S[і+m-1]) - Код\_символа(S[і-1])

Конец\_если

Конец\_цикла

По хэш-функции Рабина и Карпа.

,где p– длина строки str.

Т.е. .

Лучше вычислять не с первого символа, а с последнего, и накапливать степень двойки.

…

k=m-1

st2=1

Пока (k>=0)

Начало\_цикла

…

k=k-1

st2=st2\*2

Конец\_цикла

Пересчет.