Строки и хеши

Напоминание

Обычно хеш-функции принимают значения из большого диапазона так, что вероятность коллизии на элементах крайне мала.

Верно ли, что из этого факта можно построить сравнение строк за $O(1)$

```
const std::hash<std::string> h;
h(a) == h(b)
```

Сравнение с помощью хешей

```
const std::hash<std::string> h;
h(a) == h(b)
```

Верно ли, что сравнение выше выполняется за $O(1)$?

Вычисление хеш-функции занимает $O(n)$ времени, где n - длина строки.

Но!

Можно предподсчитать хеши строк заранее 1 раз при создании и хранить их вместе со строкой.

Если строки неизменяемы, то каждое сравнение обойдется за $O(1)$.

Хеширование строк в языках программирования

Существует несколько способов решения проблемы долгого вычисления хеш функций.

- В некоторых ЯП (например, Python) хеш-значения хранятся непосредственно вместе со строкой, что обеспечивает вычисление за $O(1)$.
- В C++ (gcc) ~~все как обычно~~ хеш-таблицы (`unordered_set` , `unordered_map`) используют служебный класс `__is_fast_hash` , который в случае долгих хеш-функций кэширует их значения и хранит непосредственно вместе с самими элементами.

Алгоритм Рабина-Карпа

Задача

Даны две строки s и t длины n и m соответственно.

Требуется найти все вхождения строки t в строку s.

Решение (?)

```
def search(s, t):
    for i in range(len(s) - len(t) + 1):
        if hash(s[i:i+len(t)]) == hash(t):
            print(i)
```

Решение (нет)

```
s = input()
t = input()

for i in range(len(s) - len(t) + 1):
    if hash(s[i:i+len(t)]) == hash(t):
        print(i)
```

Взятие подстроки и вычисление ее хеша выполняется за $O(n)$, где n - длина подстроки.

Поэтому, в худшем случае, алгоритм работает за $O(n^2)$.

Решение: идея

Хотелось бы придумать алгоритм, который бы мог вычислять хеш-значение произвольной подстроки s за $O(1)$.

Пусть $hash_s(l, r)$ - хеш-значение подстроки $s[l : r]$, которое вычисляется за $O(1)$.

```
s = input()
t = input()

# preprocess hash_s...

for i in range(len(s) - len(t) + 1):
    if hash_s(i, i+len(t)) == hash(t):
        print(i)
```

Алгоритм Рабина-Карпа

Возьмем в качестве хеш-функции $hash_s(l, r)$ следующую функцию:

$$hash_s(l, r) = \sum_{i=l}^{r-1} s[i] \cdot p^{r-1-i} \mod m,$$

, m - большое простое число, $2 \leq p \leq m - 1$.

Чем хороша эта хеш-функция?

Алгоритм Рабина-Карпа

Посчитаем хеш функцию $hash_s(r)$ для префикса строки s длины r:

$$hash_s(r) = s[0] \cdot p^{r-1} + s[1] \cdot p^{r-2} + s[2] \cdot p^{r-3} + ... + s[r-1] \mod m = \\ = \sum_{i=0}^{r-1} s[i] \cdot p^{r-1-i} \mod m = hash_s(r-1) * p + s[r-1] \mod m.$$

Тогда, хеш-значение подстроки $s[l:r]$ можно вычислить за $O(1)$:

$$hash_s(l, r) = hash_s(r) - hash_s(l) \cdot p^{r-l} \mod m.$$

```
def preprocess(s, p, m):
    deg = [1] * (len(s) + 1)
    hs = [0] * (len(s) + 1)
    for i in range(len(s)):
        hs[i + 1] = (hs[i] * p + ord(s[i])) % m
        deg[i + 1] = (deg[i] * p) % m
    return hs, deg
```

Алгоритм Рабина-Карпа

```
def preprocess(s, p, m):
    deg = [1] * (len(s) + 1)
    hs = [0] * (len(s) + 1)
    for i in range(len(s)):
        hs[i + 1] = (hs[i] * p + ord(s[i])) % m
        deg[i + 1] = (deg[i] * p) % m
    return hs, deg
```

```
def hash_s(l, r):
    return (hs[r] - hs[l] * deg[r - l]) % m
```

```
s = input()
t = input()
for i in range(len(s) - len(t) + 1):
    if hash_s(i, i + len(t)) == hash(t):  # хеш строки t достаточно посчитать 1 раз
        print(i)
```

Коллизии полиномиального хеша

Утверждение. При случайном равновероятном выборе p вероятность коллизии полиномиального хеша для двух произвольных строк s и t длины n не превосходит $\frac{n-1}{m}$.

Доказательство.

$$\sum_{i=0}^{n-1} s[i] \cdot p^i \mod m == \sum_{i=0}^{n-1} t[i] \cdot p^i \mod m$$

$$\sum_{i=0}^{n-1} (s[i] - t[i]) \cdot p^i \mod m == 0$$

По теореме Безу число корней уравнения не превосходит $n - 1$.

Так как p выбирается из множества размера m, вероятность попадания в корень $\frac{n-1}{m}$. ■