Алгоритм Рабина-Карпа

11.12.2024

Богданова Арина

Алгоритм Рабина-Карпа

- Один из алгоритмов поиска подстроки длиной m в строке длиной n
- Редко используется для поиска одиночного шаблона, но эффективен в поиске совпадений множественных шаблонов одинаковой длины
- Применим в антиплагиате

Алгоритм Рабина-Карпа

- Основан на сравнении чисел
- Использует хеширование
- Сильно зависит от качества хеш-функции:
  - В лучшем случае выполняется за $O(n)$
  - В худшем – за $O(nm)$

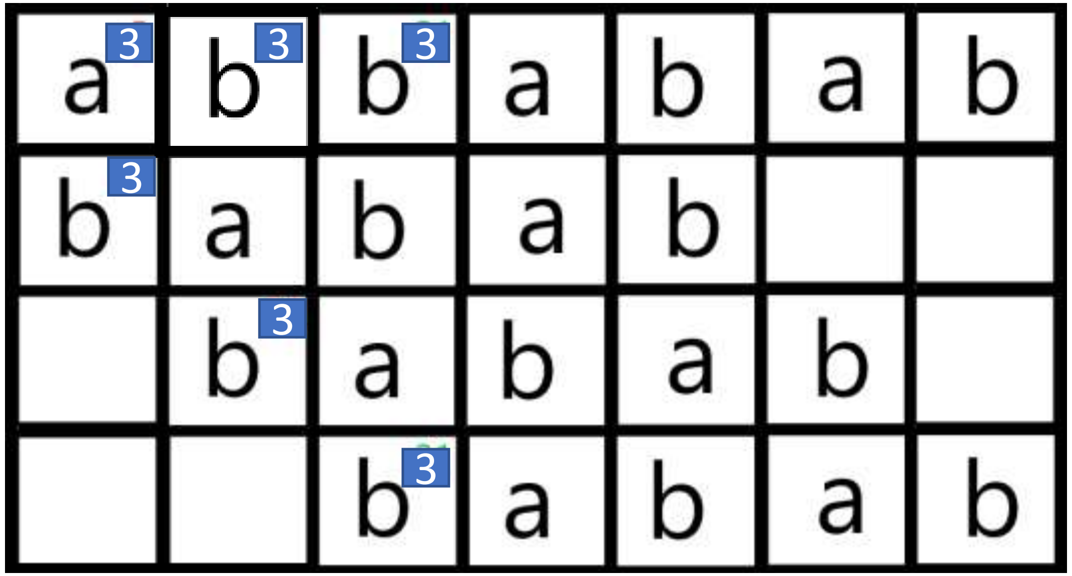
Идея наивного алгоритма первая

- Начиная с каждого места в строке сравнивать ее с образцом

```
function NaiveSearch(string s[1..n], string sub[1..m])
    for i from 1 to n-m+1
        for j from 1 to m
            if s[i+j-1] ≠ sub[j]
                перейти к следующей итерации внешнего цикла
        return i
    return not found
```

Идея наивного алгоритма вторая

- Производим хеширование:
  - Каждому символу – число
  - Суммируем полученные числа
- Хешируем образец (его длина m)
- Хешируем кусок строки длиной m
- Сравниваем хеш-значения
- Равны?
  - Да – сравниваем посимвольно
  - Нет – идем дальше



a = 0, b = 1

H(P) = 1 + 0 + 1 + 0 + 1 = 3

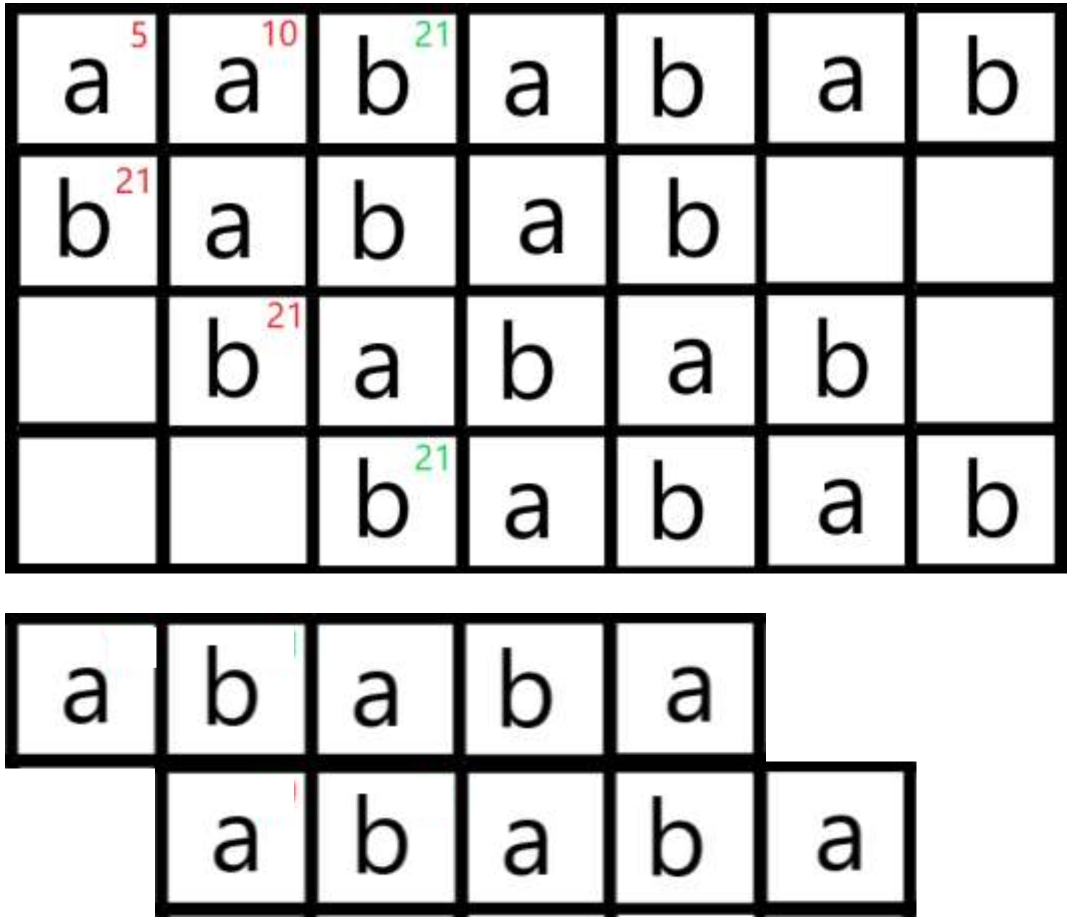
H(T[0:4]) = 0 + 1 + 1 + 0 + 1 = 3

H(T[1:5]) = H(T[0:4]) - T[0] + T[5] = 3

H(T[2:6]) = H(T[1:5]) - T[1] + T[6] = 3

Идея алгоритма

- Хешируем образец (его длина m)
- Хешируем кусок строки длиной m
- Сравниваем хеш-значение
- Равны?
  - Да – сравниваем посимвольно
  - Нет – идем дальше



Хеш-функция

- Для реализации подойдет любая кольцевая хеш-функция
- Если хеш-функция присваивает числа каждому символу и возвращает сумму - крах
- Выход – полином:
  - Возведение в степень:
    - Схема Горнера

$$H(P) = p[0] * x^{m-1} + p[1] * x^{m-2} + \ldots + p[m-1] * x^0$$

$$H(P) = p[m-1] * x^{m-1} + p[m-2] * x^{m-2} + \ldots + p[0] * x^0 = (\ldots(p[m-1] * x + p[m-2]) * x + \ldots + p[1]) * x + p[0]$$

a 5	a 10	b 21	a	b	a	b
b 21	a	b	a	b		
	b 21	a	b	a	b	
		b 21	a	b	a	b

$$H(P) = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 21$$

$$H(T[0,4]) = 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5$$

$$H(T[1,5]) = (H(T[0,4]) - 0 * 2^4) * 2 + 0 * 2^0 = 10$$

$$H(T[2,6]) = (H(T[1,5]) - 0 * 2^4) * 2 + 1 * 2^0 = 21$$

Модульная арифметика в хеш-функции

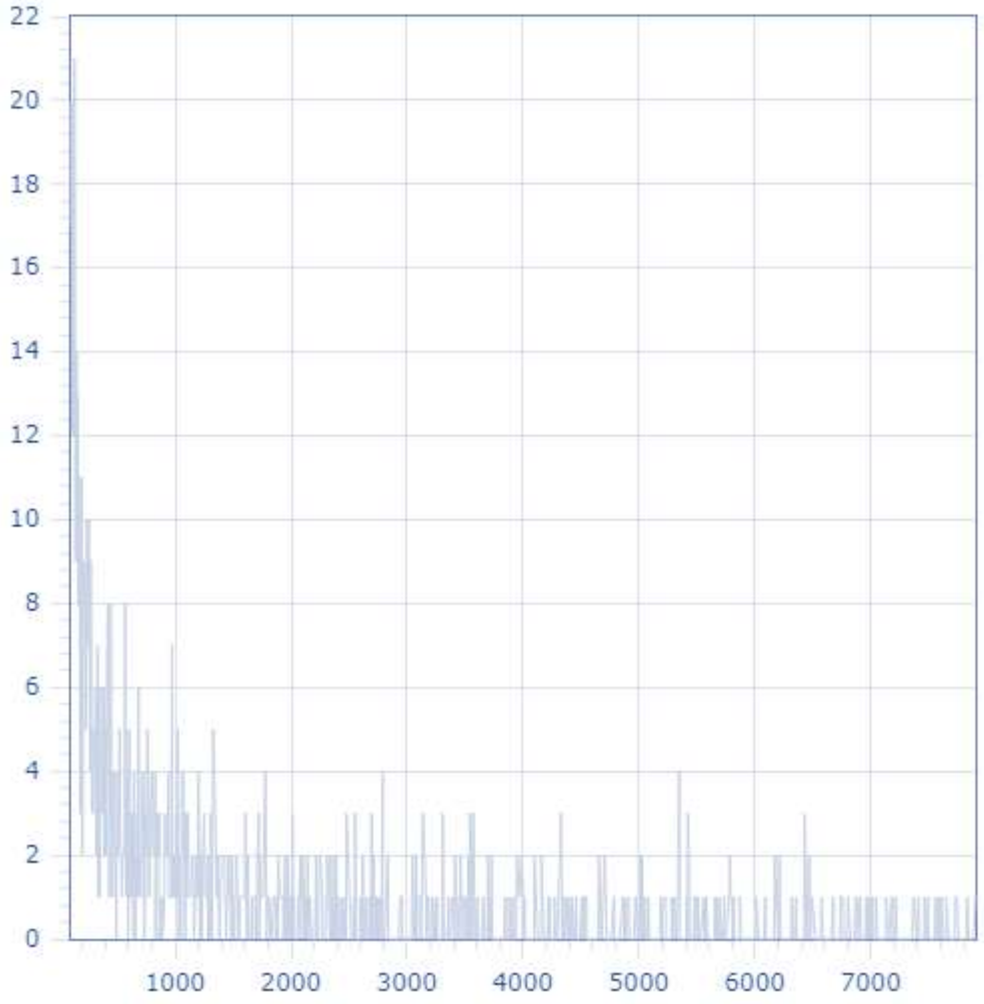
- Число **q** должно быть большим и простым
- Для текста *T[0:n]* длиной *len = n + 1.* возьмем *q > len^c*, где *c > 2,* то вероятность коллизий будет меньше, чем *1/n^(c-2)*

$$H(P) = p[0] * x^{m-1} + p[1] * x^{m-2} + \ldots + p[m-1] * x^0$$

$$H(P) = p[m-1] * x^{m-1} + p[m-2] * x^{m-2} + \ldots + p[0] * x^0 = (\ldots(p[m-1] * x + p[m-2]) * x + \ldots + p[1]) * x + p[0]$$

$$H(P) = (((\ldots((p[m-1] * x + p[m-2]) \mod q) * x + \ldots + p[1]) \mod q) * x + p[0]) \mod q$$

$$p_{max} * (x+1).$$



Пример

```
export function getSubstringRK(text: string, pattern: string): number[] {
    const result = [];

    const alphabetSize = 256;
    const mod = 9973;

    let patternHash = pattern.charCodeAt(0) % mod;
    let textHash = text.charCodeAt(0) % mod;

    let firstIndexHash = 1;

    for(let i = 1; i < pattern.length; i++)
    {
        patternHash *= alphabetSize;
        patternHash += pattern.charCodeAt(i);
        patternHash %= mod;

        textHash *= alphabetSize;
        textHash += text.charCodeAt(i);
        textHash %= mod;

        firstIndexHash *= alphabetSize;
        firstIndexHash %= mod;
    }
```

```
    for (let i = 0; i <= text.length - pattern.length; i++) {
        if (patternHash === textHash && compareText(text, i, pattern)) {
            result.push(i);
        }

        if (i === text.length - pattern.length) break;

        textHash -= (text.charCodeAt(i) * firstIndexHash) % mod;
        textHash += mod;
        textHash *= alphabetSize;
        textHash += text.charCodeAt(i + pattern.length);
        textHash %= mod;
    }

    return result;
}

function compareText(text: string, index: number, pattern: string): boolean {
    for (let i = 0; i < pattern.length; i++) {
        if (pattern[i] !== text[index + i]) {
            return false;
        }
    }

    return true;
}
```