

# Алгоритм Бойера-Мура

# Алгоритм Бойера-Мура

- Алгоритм поиска строки Бойера — Мура — алгоритм общего назначения, предназначенный для поиска подстроки в строке. Разработан Робертом Бойером и Джейм Муром в 1977 году. Преимущество этого алгоритма в том, что ценой некоторого количества предварительных вычислений над шаблоном, шаблон сравнивается с исходным текстом не во всех позициях — часть проверок пропускается как заведомо не дающая результата.
- В настоящее время алгоритм используется в текстовых редакторах и браузерах в команде Ctrl+F.

# Основные идеи алгоритма

## **1. Сканирование слева направо, сравнение справа налево.**

Совмещается начало текста (строки) и шаблона, проверка начинается с последнего символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и выполняется поиск следующего вхождения подстроки.

Если же какой-то символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на несколько (вычисляется эвристиками) символов вправо, и проверка снова начинается с последнего символа.

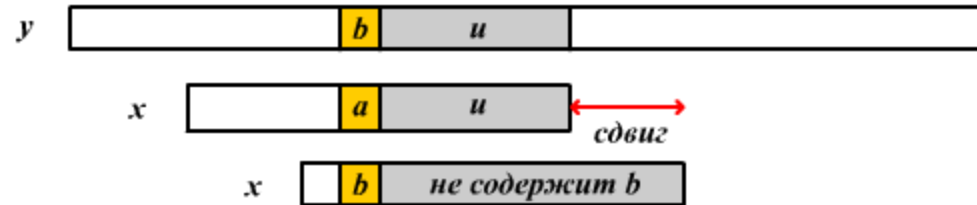
# Основные идеи алгоритма

## 2. Эвристика стоп-символа.

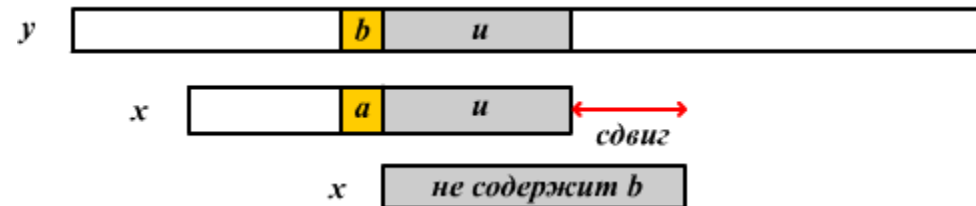
Создается массив размера  $|\Sigma|$  ( $\Sigma$  - алфавит) – это таблица стоп-символов (bad characters). В ней указывается последняя позиция в шаблоне (исключая последнюю букву) каждого из символов алфавита. Для всех символов, не вошедших в шаблон, ставится значение  $m$ . Предположим, что у нас не совпал символ  $s[i]$  текста на очередном шаге с символом из шаблона. Очевидно, что в таком случае мы можем сдвинуть шаблон до первого вхождения этого символа в шаблон, потому что совпадений других символов точно не может быть. Если в шаблоне такого символа нет, то можно сдвинуть весь шаблон полностью.

# Эвристика стоп-символа

- 1-ый случай: не совпавший символ присутствует в подстроке



- 2-ой случай: не совпавший символ отсутствует в подстроке



# Основные идеи алгоритма

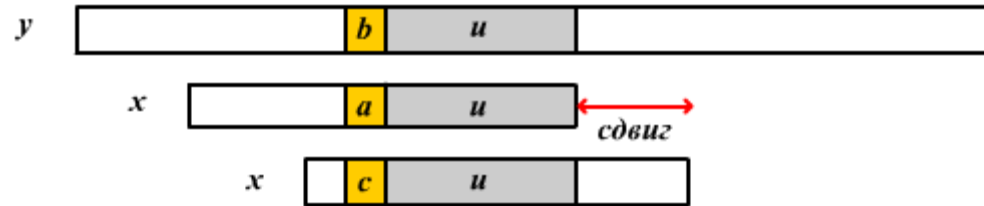
## 3. Эвристика хорошего суффикса.

Если при сравнении текста и шаблона совпало один или больше символов, шаблон сдвигается в зависимости от того, какой суффикс совпал.

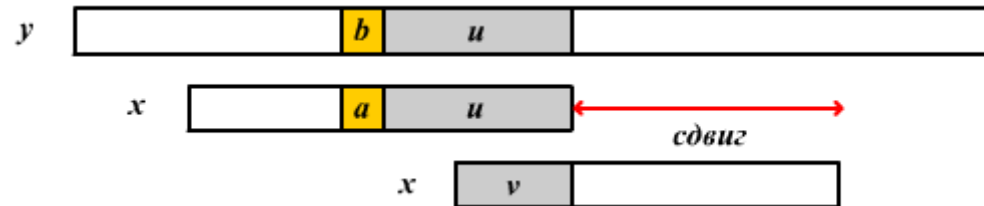
Если существуют такие подстроки равные  $u$ , что они полностью входят в  $x$  и идут справа от символов, отличных от  $x[i]$ , то сдвиг происходит к самой правой из них, отличной от  $u$ .

# Эвристика хорошего символа

1-ый случай: суффикс  $u$  присутствует еще раз в строке слева



2-ой случай: существует префикс  $v$  – подстрока суффикса  $u$



# Реализация

```
private static int[] preBmBc(String x) {  
    int[] table = new int[26];  
  
    for (int i = 0; i < 26; i++) {  
        table[i] = x.length();  
        iterations++;  
    }  
  
    for (int i = 0; i < x.length() - 1; i++) {  
        table[x.charAt(i) - 'a'] = x.length() - 1 - i;  
        iterations++;  
    }  
  
    return table;  
}
```

- Создание таблицы стоп-символов
- Асимптотика:  $O(m+|\Sigma|)$  [в данном случае  $|\Sigma| = 26$ ]



# Реализация

```
private static boolean isPrefix(String x, int p) {  
    int j = 0;  
    for (int i = p; i < x.length(); i++) {  
        if (x.charAt(i) != x.charAt(j)) {  
            return false;  
        }  
        j++;  
        iterations++;  
    }  
    return true;  
}
```

- Функция, проверяющая, что подстрока  $x[p \dots m-1]$  является префиксом шаблона  $x$ .
- Асимптотика:  $O(m-p)$

# Реализация

```
private static int suffixLength(String x, int p) {  
    int len = 0;  
    int i = p;  
    int j = x.length() - 1;  
    while (i >= 0 && x.charAt(i) == x.charAt(j)) {  
        len++;  
        i--;  
        j--;  
        iterations++;  
    }  
    return len;  
}
```

- Функция, возвращающая для позиции  $p$  длину максимальной подстроки, которая является суффиксом шаблона  $x$ .
- Асимптотика:  $O(m-p)$

# Реализация

```
private static int[] preBmGs(String x) {
    int[] table = new int[x.length()];
    int lastPrefixPos = x.length();
    for (int i = x.length() - 1; i >= 0; i--) {
        if (isPrefix(x, i + 1)) {
            lastPrefixPos = i + 1;
        }
        table[x.length() - 1 - i] = lastPrefixPos - i + x.length() - 1;
        iterations++;
    }
    for (int i = 0; i < x.length() - 1; i++) {
        int slen = suffixLength(x, i);
        table[slen] = x.length() - 1 - i + slen;
        iterations++;
    }
    return table;
}
```

- Функция для вычисления сдвигов хороших суффиксов. Требуется  $O(m)$  времени, несмотря на циклы в вызываемых функциях, из-за того, что каждый внутренний цикл в худшем случае будет выполняться на каждой позиции  $i$  не больше, чем  $i$  раз. Получается натуральный ряд, сумма  $m$  первых членов которого  $m \cdot (m-1)/2$ .
- Асимптотика:  $O(m^2)$  в худшем случае

# Реализация и оценка асимптотики

```
public static ArrayList<Integer> algorithm(String y, String x) {
    ArrayList<Integer> answer = new ArrayList<>();
    iterations = 0;
    if (x.length() == 0 || y.length() < x.length()) {
        answer.add(-1);
        iterations = 1;
        return answer;
    }

    int[] bmBc = preBmBc(x);
    int[] bmGs = preBmGs(x);
    for (int i = x.length() - 1; i < y.length(); ) {
        int j = x.length() - 1;
        while (x.charAt(j) == y.charAt(i)) {
            if (j == 0) {
                answer.add(i);
                iterations++;
                break;
            }
            --i;
            --j;
            iterations++;
        }
        i += Math.max(bmGs[x.length() - 1 - j], bmBc[y.charAt(i) - 'a']);
    }
    if (answer.isEmpty()) {
        answer.add(-1);
        iterations++;
    }
    return answer;
}
```

- Фаза предварительных вычислений требует  $O(m^2 + \sigma)$  времени и памяти.
- В худшем случае поиск требует  $O(m \cdot n)$  сравнений, так как сравнения будут произведены по всей строке.
- В лучшем случае требует  $\Omega(n/m)$  сравнений.
- $m$  — длина подстроки
- $n$  — длина исходной строки

# Графики



Входные данные: 1-ая строка (текст) размером от 100 до 10000, состоящая из прописных латинских букв; 2-ая строка из 1-7 символов (подстрока, или шаблон)

# Выводы

## Преимущества:

- Алгоритм Бойера-Мура на хороших данных очень быстр, а вероятность появления плохих данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск.

## Недостатки:

- Алгоритм Бойера-Мура не расширяется до приблизительного поиска и поиска любой строки из нескольких.
- На больших алфавитах (например, Юникод) может занимать много памяти. В таких случаях либо обходятся хэш-таблицами, либо дробят алфавит, рассматривая, например, 4-байтовый символ как пару двухбайтовых.
- На искусственно подобранных неудачных текстах скорость алгоритма Бойера-Мура серьёзно снижается. (Пример: Исходный текст **bb...bb** и шаблон **ab...ab**.)

Ссылка на репозиторий

