Алгоритмы текстового поиска

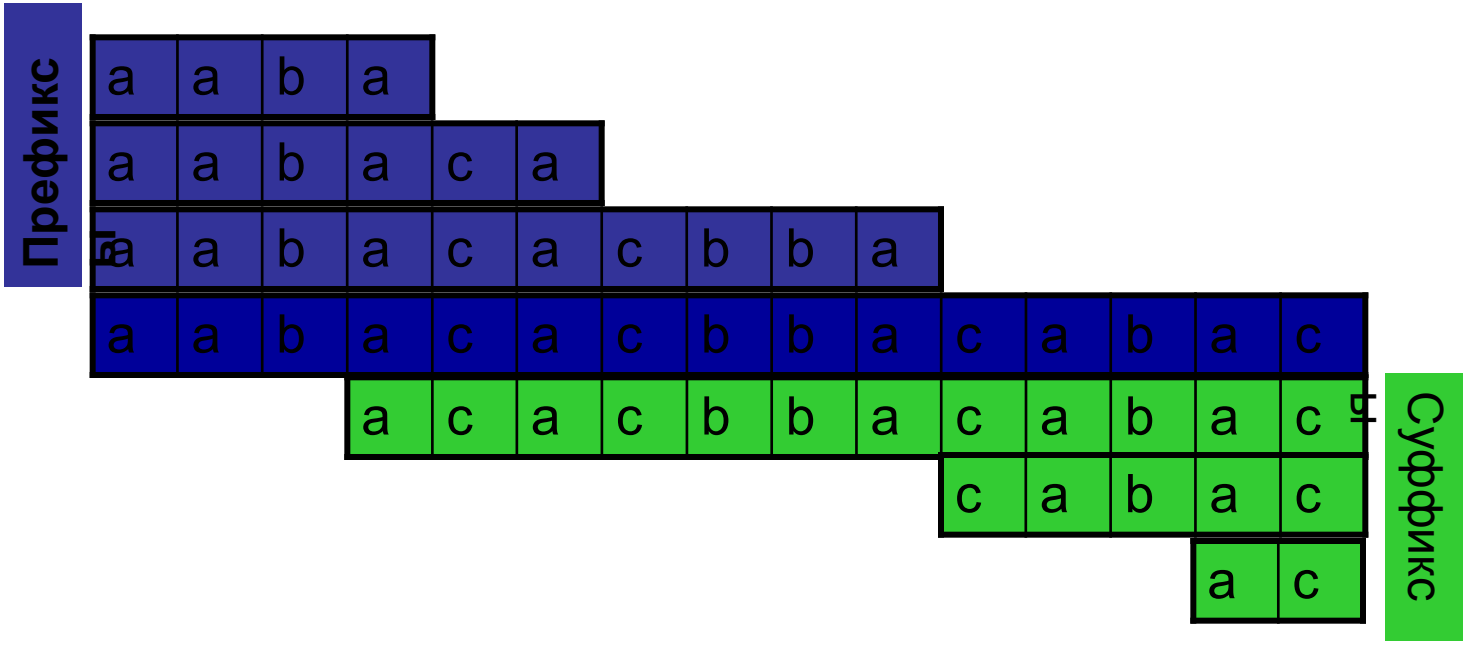
Алгоритмы точного поиска образца в тексте

Условные обозначения

- s – количество символов алфавита
- T – строка, в которой происходит поиск
- n- длина T
- p – образец, искомая строка
- m – длина p
- T[i..n]– суффикс слова T
- T[1..k]- префикс слова T, любая подстрока $1 \le k \le n$

Префикс и суффикс

- Префикс – любое подслово, начинающееся с 0
- Суффикс – окончание слова



Постановка задачи

Дан текст T *и паттерн* p *такие, что элементы этих строк — символы из конечного алфавита* s . *Требуется проверить, входит ли паттерн в текст* .

Алгоритмы поиска подстроки в строке

1. «Наивный» алгоритм

о	б	а		о	б	о	б	р	а	л	и		о	б	о	и		б	о	б	р	а

о	б	о	и

Число сравнений символов:

3 +1 +1 +1 +4 +1 +3 +1 +1 +1 +1 +1 +1 +4

Для каждого символа текста T **с** индексом **i** от **0** до **(n-m)** проверяем на сравнение с началом образца p.

Если **T[i] == p[0],** то сравниваем **T[i + 1]** и **p[1]** и т.д.

Алгоритмы поиска подстроки в строке

1. «Наивный» алгоритм

```
public static int simpleSearch(String where, String what) {
    int n = where.length();
    int m = what.length();
    extLoop:    // Внешний цикл поиска в исходной строке
    for (int i = 0; i <= n-m; i++) {
        // Внутренний цикл сравнения:
        for (int j = 0; j < m; j++) {
            if (where.charAt(i+j) != what.charAt(j))
                continue extLoop; }
        return i; }
return -1;
}
```

Для каждого символа текста T **с** индексом **i** от **0** до **(n-m)** проверяем на сравнение с началом образца p.

Если **T[i] == p[0],** то сравниваем **T[i + 1]** и **p[1]** и т.д.

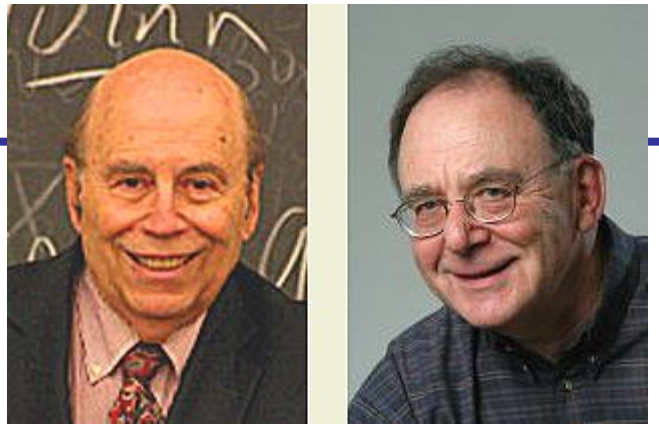
Наивный алгоритм

- ✔ Нет необходимости использовать дополнительную память.
- ✔ Нет дополнительных временных затрат.
- ✔ Худшее время поиска O((n-m)*m)≈O(n*m)

Худший случай:

```
simpleSearch("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaab"
, "aaaaaaab");
```

Алгоритм Рабина – Карпа



Два американских математика, Ричард Карп (справа) и Майкл Рабин (слева), предложили в 1987 г. очень интересный метод поиска образца в строке, «почти линейной» трудоемкости.

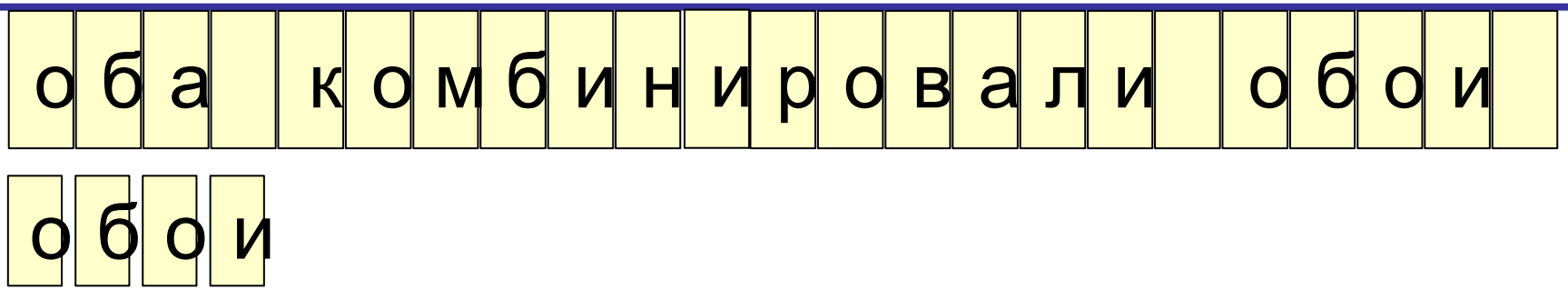
- Использование хеш-функции и сравнение чисел вместо строк. Быстрый пересчет значения хеш-функции для text[i + 1..i + m] из её значения для text[i..i + m − 1].

Примеры хеш-функций:

- сумма кодов символов;
- произведение кодов символов;
- интерпретация строки как числа в некоторой системе счисления.

Время работы в среднем — O(m + n).

Алгоритм Рабина — Карпа

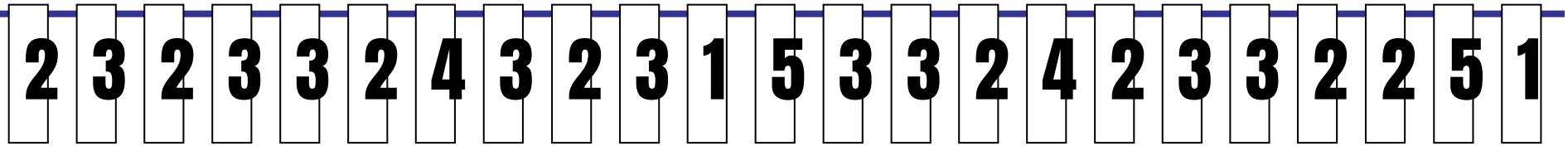



```
hash("обои") = 15 + 2 + 15 + 9 = 41

hash("оба ") = 15 + 2 + 1 + 0 = 18
hash("ба к") = 2 + 1 + 0 + 11 = 14
hash("а ко") = 1 + 0 + 11 + 15 = 27
 hash(" ком") = 0 + 11 + 15 + 13 = 39
       ...
hash(" обо") = 0 + 15 + 2 + 15 = 32
 hash("обои") = 15 + 2 + 15 + 9 = 41

hash("обои") = 15 + 2 + 15 + 9 = hash("комб") = 11 + 15 + 13 + 2 = 41
hash(" ком") = hash("а ко") + code('м') - code('а') = 27 + 13 - 1 = 39
```

2. Алгоритм Рабина – Карпа



Функция: $\sum_{i=1}^{m} s_i$ = 11

Число сравнений символов:

0 + 3 + 0 + 0 + 0 + 1 + 0 + 0 + 1 + 0 + 0 + 0 + 0 + 4 = 9

Значения функции на подстроках:

10 11 10 12 12 11 12 9 11 12 12 13 12 11

Алгоритм Рабина — Карпа.

Идея состоит в том, чтобы строкам сопоставлять значения хеш-функций и сравнивать не символы строк, а значения хеш-функций. Например, сопоставим каждой букве ее номер в алфавите, а пробелу — число 0.

Конечно, может оказаться, что одному и тому же значению хеш-функции будут соответствовать разные строки (коллизия). Так что в случае совпадения значений строки надо все же сравнивать посимвольно. Например,

```
hash("обои") = 15 + 2 + 15 + 9 = hash("комб") = 11 + 15 +
13 + 2 = 41
```

Существенно, что значения хеш-функции для следующей строки не надо перевычислять заново. Для получения нового значения надо лишь добавить код одного символа и вычесть код другого символа. Например,

```
hash(" ком") = hash("а ко") + code('м') - code('а') = 27 + 13 - 1
= 39
```

На практике хеш-функцию делают чуть более сложной, чтобы зависимость была не только от самих символов, но и их положения в строке, а также, чтобы не допустить переполнения.

Алгоритм Рабина-Карпа

Шаг 1

- Прикладываем левый край образца к левому краю текста, К = 0
- Вычисляем хэш-значения hq и hs для q и для s[0…M-1]

Шаг 2

- Если hq == hs, то проверяем, входит ли образец в текст, начиная с К-й позиции, последовательным сравнением символов образца q[j] с символами текста s[K+j] слева направо,j=0…M-1

Шаг 3

- Если имеем M совпадений, то образец в тексте найден – конец работы
- Если K+M >= N, то образец в тексте не найден – конец работы
- Иначе вычисляем hs для s[K+1…K+M], используя hs для s[K…K+M-1], K = K+1 и переходим к шагу 2

В худшем случае O((N - M)*M) сравнений

В "среднем" O(N) сравнений, зависит от выбора хэш-функции

Алгоритм Рабина — Карпа (оценка)

Если количество холостых срабатываний невелико, то время работы алгоритма будет пропорционально длине строки, в которой производится поиск. $t = O(n)$

Легко привести пример строк, в котором холостые срабатывания будут происходить очень часто, и, соответственно, время работы ухудшается до $t = O(n*m)$

Пример случая, в котором «наивный» алгоритм будет работать быстрее, чем алгоритм Рабина — Карпа (здесь предположим, что и хеш-функция достаточно «наивна»):

а	б	а	б	а	б	а	б	а	б	а	б	а	б	а	б	а	б	а	б	а	а	б	б	а
а	а	б	б																					

Здесь для почти каждого из четырехбуквенных фрагментов значение хеш-функции будет совпадать со значением хеш-функции строки поиска, но все «срабатывания» будут холостыми.

Алгоритм Рабина — Карпа удобно применять в следующих случаях:

- сравнение элементов «дорого», а вычисление хеш-функции - «дешево»;
- элементы разнообразны, так что вероятность холостых срабатываний низка;
- идет поиск сразу нескольких подстрок, каждая из которых имеет свое значение хеш-функции.

Алгоритм Бойера-Мура

- Алгоритм Бойера — Мура Сравнение с конца образца.
- Сильное/слабое правило плохого символа.
- Сильное/слабое правило хорошего суффикса.
- Каждое правило говорит, на сколько позиций сдвинуть образец. Выбираем из двух сдвигов наибольший. Время на препроцессинг образца — O(m)
- Одно слабое правило плохого символа дает время поиска O(n/m) в лучшем случае, но в худшем те же O(mn).
- Два правила вместе в сильном виде дают время поиска O(n) в худшем случае.

Алгоритм Бойера-Мура

Алгоритм Бойера-Мура состоит из следующих шагов:

1) Строится таблица смещений для искомой подстроки.
2) Далее совмещается начало исходной строки и искомого образца и начинается проверка с последнего символа образца .

3)Если последний символ образца и соответствующий ему символ строки совпадают, то проверяем предпоследний символ образца и т.д.

Если символ образца и соответствующий ему символ строки не совпадают, то образец сдвигается по одному из 4 правил (на следующем слайде)

Если все символы образца совпали с соответствующими символами строки, то вхождение искомой подстроки найдено.

Алгоритм Бойера-Мура

Пример 1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ТЕКСТ	с	у	п	е	р	а	б	р	а	к	а	д	а	б	р	а

ОБРАЗЕЦ

Таблица 1

3	2	1	0
б	р	а	к

Таблица 2

а	б	с	д	е	п	к	р	у
1	3	4	4	4	4	4	2	4

Если последняя буква образца не совпадает с символом с, то сдвиг выполняется по правилам:

1) Если символа **с** в образце нет, то **сдвиг = длине образца**

2) Если символа **с** в образце есть и не последняя буква образца, то **сдвиг = компоненте из таблицы 2 для символа строки.**

3) Если **с** – последний символ образца и среди остальных **m – 1** символов образца такого символа больше нет, то сдвиг должен быть подобен сдвигу в случае **1** – образец следует сдвинуть на всю длину **m**

Если вначале совпадения образца и символов строки есть, а на символе с прервалось, то

Если символ с в образце есть (символа с в образце нет), то **сдвиг = компоненте из таблицы 2 (или длине образца) - количество символов до несовпадения.**

Алгоритм Бойера-Мура

Пример 1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ТЕКСТ	с	у	п	е	р	а	**б**	**р**	**а**	**к**	а	д	а	б	р	а
ОБРАЗЕЦ	б	р	а	к												
					б	р	а	к								
							б	р	а	к						

3	2	1	0
б	р	а	к

а	б	с	д	е	п	к	р	у
1	3	4	4	4	4	4	2	4

Пример2 СЧИТАЕМ СДВИГ

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ТЕКСТ	с	у	п	е	р	а	б	р	а	к	а	д	а	б	р	а

ОБРАЗЕЦ	3	2	1	0
	д	а	б	р

Таблица **1**

а	б	с	д	е	п	к	р	у
2	1	4	3	4	4	4	4	4

Таблица **2**

1) Если символа **с** в образце нет, то **сдвиг = длине образца**

2) Если символа **с** в образце есть и не последняя буква образца, то **сдвиг = компоненте из таблицы 2 для символа строки.**

3) Если символ с в образце есть, но он не последний в строке, то **сдвиг = длине образца - количество символов до несовпадения.**

4) Если с – последний символ образца и среди остальных **m – 1** символов образца такого символа больше нет, то сдвиг должен быть подобен сдвигу в случае **1** – образец следует сдвинуть на всю длину **m**

Пример 2

Таблица 1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ТЕКСТ	с	у	п	е	р	а	б	р	а	к	а	д	а	б	р	а
ОБРАЗЕЦ	д	а	б	р												
					д	а	б	р								
						д	а	б	р							
								д	а	б	р					
										д	а	б	р			
											д	а	б	р		

3	2	1	0
д	а	б	р

а	б	с	д	е	п	к	р	у
2	1	4	3	4	4	4	4	4

Алгоритм Боуера-Мура

- Худшее время работы алгоритма – O(n * m)
  - Среднее – O(n + m)
- Дополнительные затраты на память O(m + s)
- Время на предобработку O(m + s)
- *3n* сравнений в худшем случае при поиске первого совпадения с непериодничным образцом

ВИДЫ ПРЕПРОЦЕССИНГА:

1. Префикс-функция
2. Z-функция
3. Бор
4. Суффиксы массив

Z-функция Гасфилда

- ***Z-функция — это вектор длин наибольшего общего префикса строки с ее суффиксом. .***

№	Суффикс	Z
1	абракадабра	8
2	бракадабра	0
3	ракадабра	0
4	акадабра	1
5	кадабра	0
6	адабра	1
7	дабра	0
8	абра	4
9	бра	0
10	ра	0
11	а	1

Поиск подстроки в строке с помощью Z-функции

n — длина текста. **m** — длина образца.

Образуем строку s = pattern + # + text, где # — символ, не встречающийся ни в text, ни в pattern.

Вычисляем Z-функцию от этой строки. В полученном массиве, в позициях в которых значение Z-функции равно |pattern|, по определению начинается подстрока, совпадающая с pattern.

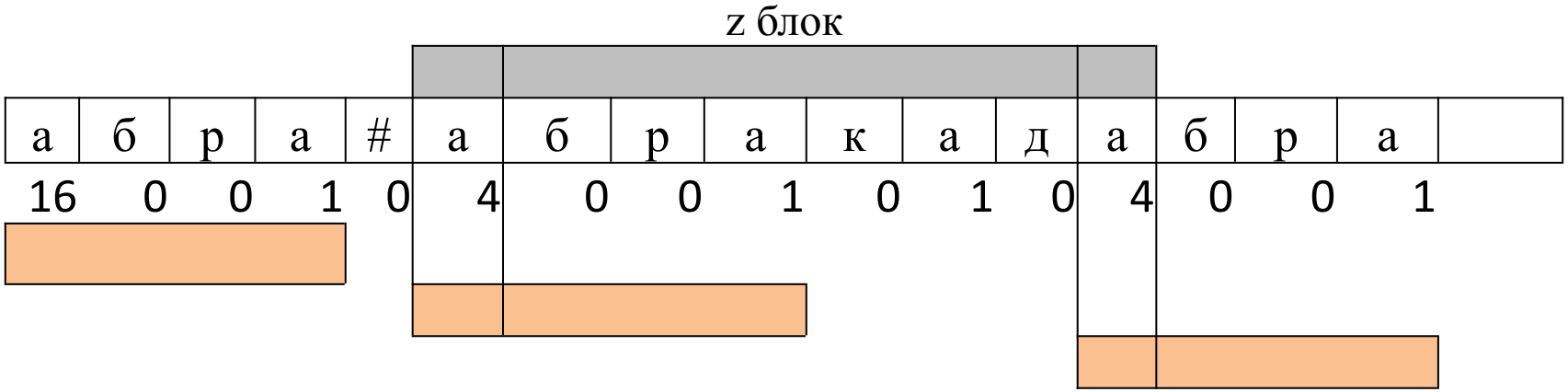
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	z
а	б	р	а	#	а	б	р	а	к	а	д	а	б	р	а	16
	б	р	а	#	а	б	р	а	к	а	д	а	б	р	а	0
		р	а	#	а	б	р	а	к	а	д	а	б	р	а	0
			а	#	а	б	р	а	к	а	д	а	б	р	а	1
				#	а	б	р	а	к	а	д	а	б	р	а	0
					а	б	р	а	к	а	д	а	б	р	а	4
						б	р	а	к	а	д	а	б	р	а	0
							р	а	к	а	д	а	б	р	а	0
								а	к	а	д	а	б	р	а	1
									к	а	д	а	б	р	а	0
										а	д	а	б	р	а	1
											д	а	б	р	а	0
												а	б	р	а	4
													б	р	а	0
														р	а	0
															а	1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	z
а	б	р	а	#	а	б	р	а	к	а	д	а	б	р	а	16
	б	р	а	#	а	б	р	а	к	а	д	а	б	р	а	0
		р	а	#	а	б	р	а	к	а	д	а	б	р	а	0
			а	#	а	б	р	а	к	а	д	а	б	р	а	1
				#	а	б	р	а	к	а	д	а	б	р	а	0
					а	б	р	а	к	а	д	а	б	р	а	4
						б	р	а	к	а	д	а	б	р	а	0
							р	а	к	а	д	а	б	р	а	0
								а	к	а	д	а	б	р	а	1
									к	а	д	а	б	р	а	0
										а	д	а	б	р	а	1
											д	а	б	р	а	0
												а	б	р	а	4
													б	р	а	0
														р	а	0
															а	1

z блок

а	б	р	а	#	а	б	р	а	к	а	д	а	б	р	а	
16	0	0	1	0	4	0	0	1	0	1	0	4	0	0	1	

Поиск подстроки в строке с помощью Z-функции



Сравнивать символы надо только правее самого правого **Z** блока

Псевдокод

```
int substringSearch(text : string, pattern : string):
  int[] zf = zFunction(pattern + '#' + text)
  for i = m + 1 to n + 1
    if zf[i] == m
      return i
```

- *Строке «**абракадабра»***
- *Образец «**рак**».*
- *Конкатенируем строки «**рак$абракадабра**».*
- *Вектор Z-функции выглядит для такой строки так:*

р	а	к	$	а	б	р	а	к	а	д	а	б	р	а
15	о	о	о	о	о	з	о	о	1	о	4	о	о	1

Поиск сводится к нахождению компонента z-функции равного длине образца.

Префикс-функция

Определение

- Дана строка s[0 ..n-1].
- Требуется вычислить для неё префикс-функцию, т.е. массив чисел π[0 .. n-1],
- где π[i] - наибольшая длина наибольшего собственного суффикса подстроки s[0.. i], совпадающего с её префиксом (собственный суффикс-значит не совпадающий со всей строкой).
- В частности, π[0] =0

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max_{k=0\ldots i} \{\ k\ :\ s[0\ldots k-1] = s[i-k+1\ldots i]\ \}.$$

Пример

Префиксная функция -в какой мере образец совпадает сам с собой после сдвигов.

Например, для строки "abcabcd" префикс-функция равна: [0, 0, 0, 1, 2, 3, 0], что означает:

- у строки "a" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "ab" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abc" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abca" префикс длины 1 совпадает с суффиксом a;
- у строки "abcab" префикс длины 2 совпадает с суффиксом ab;
- у строки "abcabc" префикс длины 3 совпадает с суффиксом abc;
- у строки "abcabcd" нет нетривиального префикса, совпадающего с суффиксом.

$\pi[1..7]=[0,0,0,1,2,3,0]$

a	b	c	a	b	c	d
0						
	0					
		0				
			1			
				2		
					3	
						0

Пример префикс -функции

Шаблон "ABABACA”

- A, нет совпадений 0
- AB, нет совпадений 0
- ABA, одно совпадение: a a
- ABAB, два совпадения: ab ab
- ABABA, три совпадения: aba aba
- ABABAC, нет совпадений:
- ABABACA, одно совпадение: a a

$\pi[1..7]=[0,0,1,2,3,0,1]$

Алгоритм Кнута- Морриса-Пратта

Эта задача является классическим применением префикс-функции

Рис. 2: Дональд Кнут (род. 1938), Джеймс Моррис (род. 1941), Вон Пратт (род. 1944).

Пример:
Текст ***«ababcabcacab»***
ищем ***«abca».***

Конкатенированный вариант ***«abca$ababcabcacab».***

Префикс-функция выглядит так:

a	b	c	a	$	a	b	a	b	c	a	b	c	a	c	a	b
0	0	0	1	0	1	2	1	2	3	4	2	3	4	0	1	2

Все вхождения подстроки оканчиваются на позициях четверок.

Алгоритм Кнута- Морриса-Пратта

- **Текст b b a b a b a b a a b a b a b b a**
- **Образец a b a b a b b a**

Шаг**1** Вычислим префикс функцию

k	1	2	3	4	5	6	7	8
Образец	a	b	a	b	a	b	b	a
π(k)	0	0	1	2	3	4	0	1

Шаг **2** Поиск подстроки в тексте с первой позиции. Если нет совпадений, то смещаем на **1**.

b	b	a	b	a	b	a	b	a	a	b	a	b	a	b	b	a
		a	b	a	b	a	b	b	a							

Если образец не совпал с текстом на **7** позиции, то вычисляем сдвиг:

$$\Delta = 6 - \pi(6) = 2$$

Алгоритм Кнута- Морриса-Пратта

b	b	a	b	a	b	a	b	a	a	b	a	b	a	b	b	a
		a	b	a	b	a	b	b	a							

$\Delta = 6 - \pi(6) = 2$

b	b	a	b	a	b	a	b	a	a	b	a	b	a	b	b	a
				a	b	a	b	a	b	b	a					

$\Delta = 5 - \pi(5) = 2$

b	b	a	b	a	b	a	b	c	a	b	a	b	a	b	b	a
						a	b	a	b	a	b	b	a			

Алгоритм Кнута- Морриса-Пратта

Время на препроцессинг + поиск: O(m + n) в худшем случае

- Проход по тексту O(n)
- Построение таблицы префиксной функции O(m)

- Дополнительные затраты на память O(m)

Выводы

- Алгоритм Кнута-Мориса-Пратта:
  - Несмотря на лучшую оценку по сравнению с алгоритмом грубой силы, на практике показал себя не очень хорошо.
- Алгоритм Боуера-Мура на данном тесте показал себя лучше всех других алгоритмов.

Алгоритм Кнутта- Морриса-Пратта

```
public static int KnuthMorrisPratt(String where, String what) {
   int n = where.length();   // Длина строки, в которой происходит поиск
   int m = what.length();    // Длина подстроки
   // Формирование таблицы сдвигов
   int[] table = new int[m];
   table[0] = 0;
   int shift = 0;
   for (int q = 1; q < m; q++) {
      while (shift > 0 && what.charAt(shift) != what.charAt(q)) {
         shift = table[shift-1];
      }
      if (what.charAt(shift) == what.charAt(q)) shift++;
      table[q] = shift;
   }
   // Поиск с использованием таблицы сдвигов
   shift = 0;
   for (int i = 0; i < n; i++) {
      while (shift > 0 && what.charAt(shift) != where.charAt(i)) {
         shift = table[shift-1];
      }
      if (what.charAt(shift) == where.charAt(i)) shift++;
      if (shift == m) return i-m+1;   // подстрока найдена
   }
   return -1;    // подстрока не найдена
```

Алгоритм Рабина - Карпа

```
public static int RabinKarp(String where, String what) {
    int n = where.length();// Длина строки, в которой происходит поиск
    int m = what.length();    // Длина подстроки
    long h = 1; // Вычисляемый числовой показатель вытесняемой буквы
    for (int k = 1; k <= m-1; k++) {  h <<= 8; h %= q; }
 long p = 0;// Числовой показатель подстроки - вычисляется один раз
 long t = 0;// Изменяемый числовой показатель участка исходной строки
    for (int k = 0; k < m; k++) {
        p = ((p << 8) | (byte) what.charAt(k)) % q;
        t = ((t << 8) | (byte) where.charAt(k)) % q;
    } // Внешний цикл по исходной строке
    extLoop:for (int i = 0; i <= n-m; i++) {
        if (p == t) {
// Показатели строк совпали; проверяем,не холостое ли это срабатывание
            for (int j = 0; j < m; j++) {
                if (where.charAt(i+j) != what.charAt(j)) {
                    // символы не совпали - продолжаем поиск
                    continue extLoop;
                }     }  // подстрока найдена!
            return i;
        } else if (i < n-m) { // сдвиг по исходной строке
            t = (((t - h * (byte) where.charAt(i)) << 8) | (byte)
   where.charAt(i+m)) % q; }    }    return -1;}
```

Алгоритм Бойера-Мура

```
private static final int shLen = 256;
private static int hash(char c) { return c & 0xFF; }
public static int BoyerMoore(String where, String what) {
    int n = where.length();    // Длина исходной строки
    int m = what.length();     // Длина образца
int[] shifts = new int[shLen]; // Формирование массива сдвигов
    // Для символов, отсутствующих в образце, сдвиг равен длине образца
    for (int i = 0; i < shLen; i++) {
        shifts[i] = m;    }
 // Символов из образца сдвиг равен расстоянию от последнего вхождения в образец до конца for
 (int i = 0; i < m-1; i++) {
        shifts[hash(what.charAt(i))] = m-i-1;
    }    // Поиск с использованием таблицы сдвигов
    for (int i = 0; i <= n-m; ) {
        // Сравнение начинается с конца образца
        for (int j = m-1; j>=0; j--) {
            if (where.charAt(i+j) == what.charAt(j)) {
                if (j == 0) return i;
            } else {                break;            }
        }
        // Сдвиг производится в соответствии с кодом последнего из сравниваемых символов
        i += shifts[hash(where.charAt(i+m-1))];
    }
    return -1;}
```

Список литературы

- https://sites.google.com/site/dssearchtext/showcase