

Алгоритмы и Алгоритмические Языки

Семинар #10:

1. Наивный алгоритм поиска подстроки.
2. Алгоритм Рабина-Карпа.
3. Алгоритм Кнута-Морриса-Пратта.
4. Сравнение алгоритмов поиска подстроки.

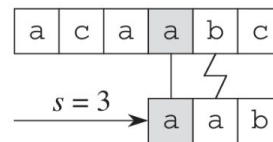
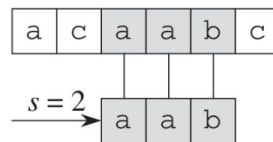
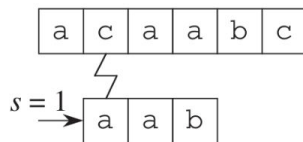
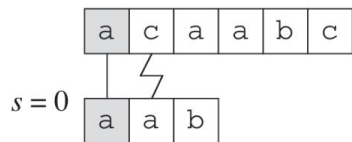
Наивный алгоритм поиска подстроки



Наивный алгоритм

NAIVE-STRING-MATCHER(T, P)

```
1   $n = T.length$ 
2   $m = P.length$ 
3  for  $s = 0$  to  $n - m$ 
4      if  $P[1..m] == T[s + 1..s + m]$ 
5          print “Pattern occurs with shift”  $s$ 
```



Алгоритм Рабина – Карпа



Идея алгоритма Рабина – Карпа

Для простоты: пусть $\text{char} = \{0, 1, \dots, 9\}$

Подготовка – вычисление хэша от `needle`-подстроки:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10P[1]) \dots))$$

На каждом шаге – обновление текущего хэша от фрагмента подстроки `haystack`:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

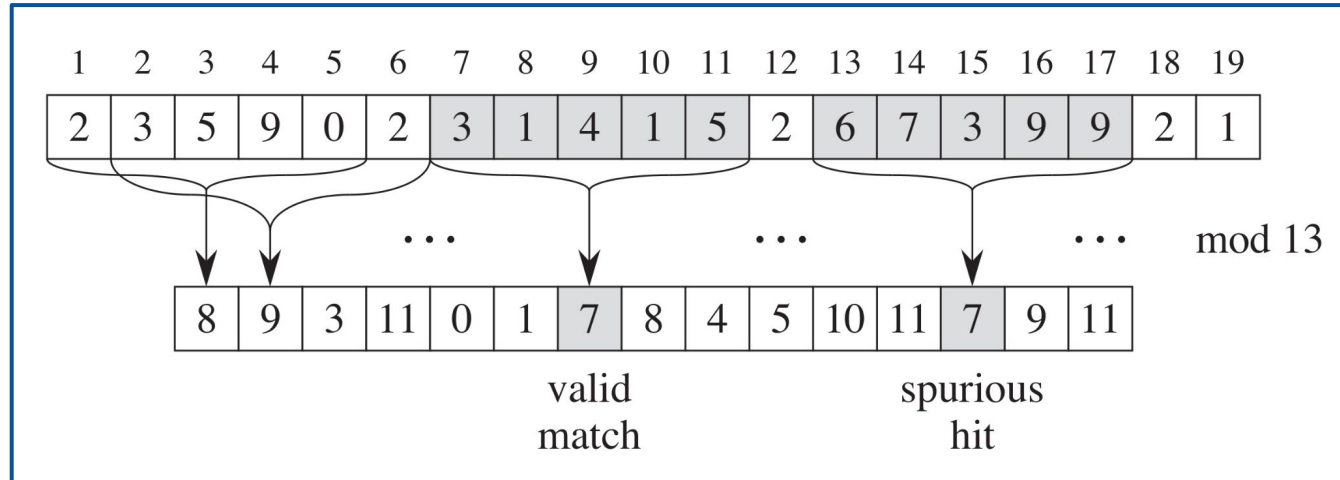
Честное сравнение – только в случае совпадения хэша.

Рассмотрение хэша по модулю

Для ограничения значения хэша сверху нужен модуль:

$$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$$

(здесь $h = d^{m-1} \bmod q$)



Псевдокод алгоритма

RABIN-KARP-MATCHER(T, P, d, q)

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $h = d^{m-1} \bmod q$ 
4   $p = 0$ 
5   $t_0 = 0$ 
6  for  $i = 1$  to  $m$            // preprocessing
7       $p = (dp + P[i]) \bmod q$ 
8       $t_0 = (dt_0 + T[i]) \bmod q$ 
9  for  $s = 0$  to  $n - m$        // matching
10     if  $p == t_s$ 
11         if  $P[1..m] == T[s + 1..s + m]$ 
12             print "Pattern occurs with shift"  $s$ 
13     if  $s < n - m$ 
14          $t_{s+1} = (d(t_s - T[s + 1])h + T[s + m + 1]) \bmod q$ 
```

Сложность алгоритма в лучшем/худшем случае = ???

Алгоритм Кнута – Морриса – Пратта



Неформальное рассмотрение

Будем искать подстроку «методом прикладывания»:

a b c a b c a c a b

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

^

Так как символы не совпали, то окно сдвигается.

_ a b c a b c a c a b

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

^

Если сравнение успешное, то указатель сдвигается внутри окна.

a b c a b c a c a b

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

^

Неформальное рассмотрение

Легко заметить, что оптимальный сдвиг окна равен четырём:

```
  _ _ _ _ a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
      ^
```

Рассмотрим следующее неуспешное сравнение:

```
  a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
      ^
```

При выборе сдвига важна рекурсивная структура самоповторений:

```
  _ _ _ a b c a b c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
      ^
```

Неформальное рассмотрение

b a b c b a b c a b c a b c a b c a b c a b c
 _ _ _ a b c a b c a b c
 ^

Снова 7 совпавших символов и 8-ой несовпавший:

[illegible]

И снова первый сдвиг окна – на три символа:

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

Финал – совпадение найдено.

Префикс-функция строки

Определение

Пусть дана строка str .

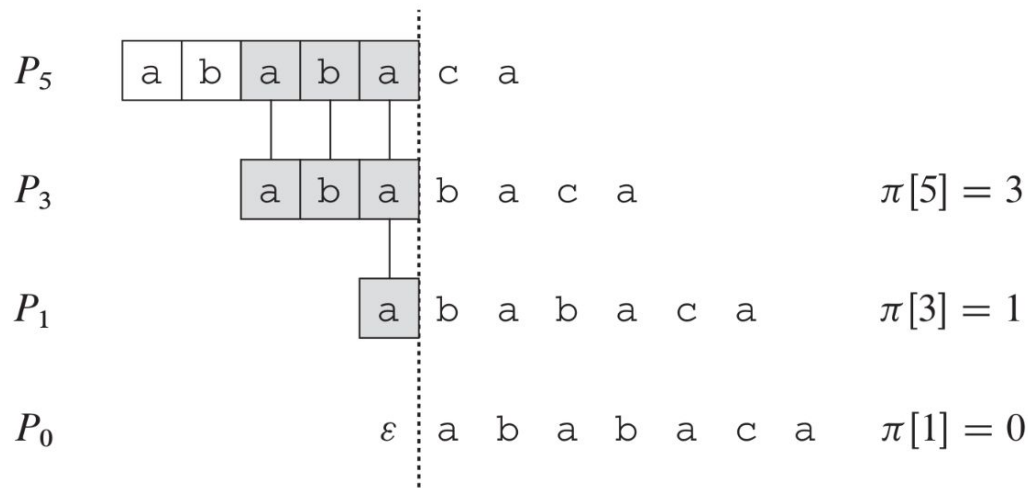
Префикс-функция строки str – такой массив чисел π , что $\pi[i]$ равен длине наибольшего префикса строки $str[1, i]$, являющегося также и суффиксом строки $str[1, i]$ и при этом не совпадающего с самой строкой $str[1, i]$.

$$\pi[i] = \max\{k : 0 \leq k < i \wedge str[1, k] = str[i - k + 1, i]\}$$

Вычисление префикс-функции

$$\pi[i] = \max\{k : 0 \leq k < i \wedge str[1, k] = str[i - k + 1, i]\}$$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1



Задание: вычислить COMPUTE-PREFIX-FUNCTION("ababaca")

Вычисление префикс-функции

COMPUTE-PREFIX-FUNCTION(P)

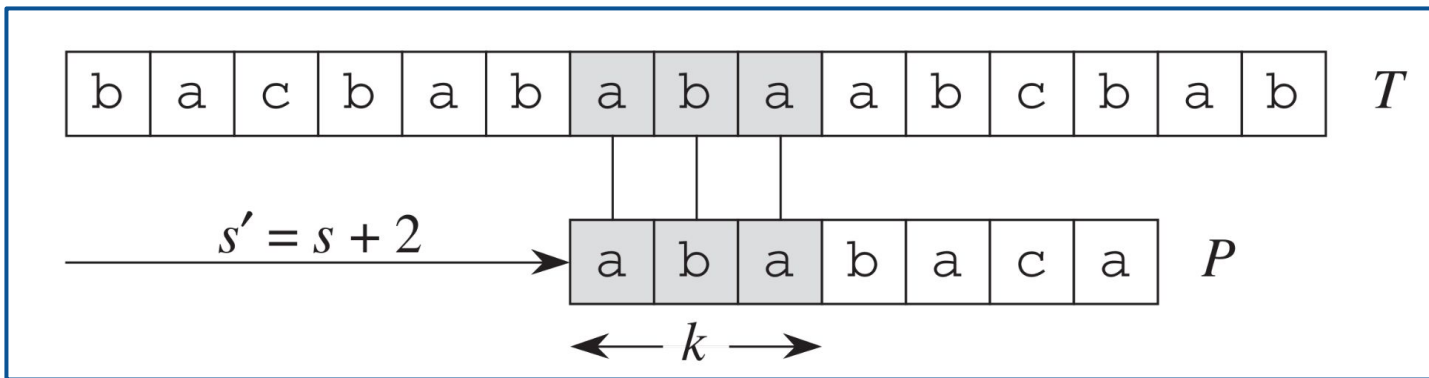
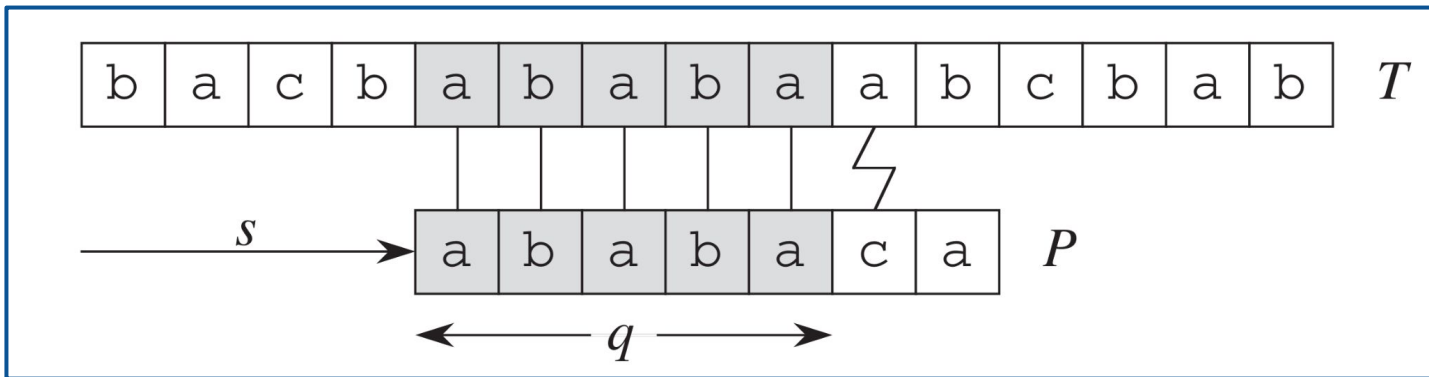
```
1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6      while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
7           $k = \pi[k]$ 
8      if  $P[k + 1] == P[q]$ 
9           $k = k + 1$ 
10      $\pi[q] = k$ 
11 return  $\pi$ 
```

Алгоритм КМП

KMP-MATCHER(T, P)

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$  // number of characters matched
5  for  $i = 1$  to  $n$  // scan the text from left to right
6      while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7           $q = \pi[q]$  // next character does not match
8      if  $P[q + 1] == T[i]$ 
9           $q = q + 1$  // next character matches
10     if  $q == m$  // is all of  $P$  matched?
11         print "Pattern occurs with shift"  $i - m$ 
12          $q = \pi[q]$  // look for the next match
```

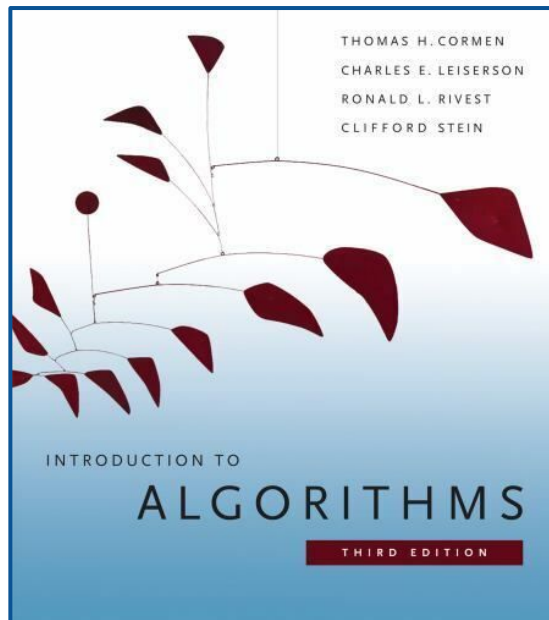
Алгоритм КМП



Задание: вычислить KMP-MATCHER(“bacbabababcbab”, “ababaca”)

Рекомендуемая литература

Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein
Introduction to Algorithms



Сравнение алгоритмов поиска подстроки



Замеры времени исполнения

```
// Запускаем измерение времени
clock_t ticks_start = clock();

compute_algorithm();

// Завершаем измерение времени
clock_t ticks_end = clock();

double ticks_delta = ticks_end - ticks_start;
double seconds = ticks_delta / CLOCKS_PER_SEC;

printf("Number of occurrences: %lu\n", occurrences);
printf("Time: %10.6lfs\n", seconds);
```

Замер сразу нескольких запусков

```
// Запускаем измерение времени
clock_t ticks_start = clock();

for (size_t iterations = 0; iterations < NUM_ITERATIONS; ++iterations)
{
    compute_algorithm(...);
}

// Завершаем измерение времени
clock_t ticks_end = clock();

double ticks_delta = ticks_end - ticks_start;
double seconds = ticks_delta / CLOCKS_PER_SEC;
```

Зачем повторять эксперимент NUM_ITERATIONS раз подряд?

Сравнение без перекомпиляции

```
// Функции алгоритмов поиска подстроки
const char* (*algorithms[])(const char*, const char*) =
{
    &strstr_naive,
    &strstr_rabin_karp,
    &strstr_knuth_morris_pratt,
    (const char* (*)(const char*, const char*)) &strstr
};

// Названия алгоритмов поиска подстроки
const char* algorithm_names[] =
{
    "Наивный алгоритм",
    "Алгоритм Рабина-Карпа",
    "Алгоритм Кнута-Морриса-Пратта",
    "Библиотечная реализация strstr"
};
```

Оценка системы алгоритм+задача

Какой алгоритм оптимальный? Зависит от данных!

```
// Оценку производительности алгоритмов поиска строк целесообразно оценивать
// совместно с корпусом слов, на котором алгоритм будет работать.
const char* needles[] =
{
    "любовь", "счастье", "добродетель", "гнев", "смерть",
    "время", "память", "слово", "дело", "смысл", "разум",
    "Андрей", "Наташа", "Николай", "Соня", "Пьер", "Элен", "дуэль",
    "салон", "madame", "mon cher", "обман",
    "Россия", "война", "Кутузов", "Наполеон", "друг", "враг", "гвардия", "бог",
    "есть только две добродетели: деятельность и ум",
    "есть только два источника людских пороков: праздность и суеверие",
    "Нездоровы, брат, бывают только дураки да развратники",
};
```

Для разных данных разные алгоритмы будут оптимальными!

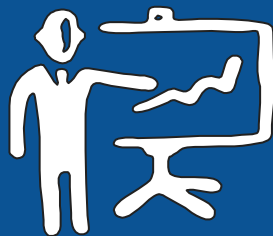
Что французу за здоровье – русскому за упокой!

Результаты измерений

```
> ./build/search  
Наивный алгоритм:  
      4.085240s  
Алгоритм Рабина-Карпа:  
      0.520688s  
Алгоритм Кнута-Морриса-Пратта:  
      2.503532s  
Библиотечная реализация strstr:  
      0.175467s
```

Библиотечная реализация вне конкуренции!

Вопросы?



Красивые иконки взяты с сайта handdrawngoods.com