Высшая школа экономики
Факультет компьютерных наук
Департамент программной инженерии

Алгоритмы и алгоритмические языки
Лекция 8

11 ноября 2025 г.

Дробные двоичные числа

Что такое 1011.101_2?

Дробные двоичные числа

Что такое 1011.101_2?

$$1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}=11\frac{5}{8}=11.625.$$

Дробные двоичные числа

Что такое 1011.101_2?

$$1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}=11\frac{5}{8}=11.625.$$

$0.1111\ldots_2 = 1.0-\varepsilon(\varepsilon\to 0)$, т.к. $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}\to 1$ при $n\to\infty$.

Точно можно представить только числа вида $\frac{x}{2^k}$.

Осстальные рациональные числа представляются периодическими двоичными дробями: $\frac{1}{5}=0.(0011)_2$.

Иррациональные числа представляются апериодическими двоичными дробями и могут быть представлены только приближенно.

Представление чисел с плавающей точкой (IEEE 754)

Числа с плавающей точкой представляются в *нормализованной* форме:

$$(-1)^s M 2^e, \text{ где}$$

- s — код знака числа (он же знак мантиссы);
- M — мантисса ($1 \leqslant M < 2$);
- e — (двоичный) порядок.

Первая цифра мантиссы в нормализованном представлении всегда 1. В стандарте принято решение не записывать в представление числа эту единицу (тем самым мантисса как бы увеличивается на разряд).

В представление числа записывается не M, а $frac = M - 1$.

Представление чисел с плавающей точкой (IEEE 754)

Чтобы не записывать отрицательных чисел в поле порядка, вводится *смещение* $bias = 2^{k-1} - 1$, где k — количество бит в поле для записи порядка, и вместо порядка e записывается *код* порядка exp, связанный с e соотношением $e = exp - bias$.

Нормализованное число $(-1)^s M 2^e$ упаковывается в машинное слово с полями s, $frac$ и exp.

s	exp (код порядка)	$frac$ (код мантиссы)

Ширина поля s всегда равна 1. Ширина полей exp и $frac$ зависит от точности числа.

Типы плавающей арифметики (точность)

Одинарная точность (32 бита): exp — 8 бит, $frac$ — 23 бита.
$bias = 127$, $-126 \leqslant e \leqslant 127$, $1 \leqslant exp \leqslant 254$.

Двойная точность (64 бита): exp — 11 бит, $frac$ — 52 бита.
$bias = 1023$, $-1022 \leqslant e \leqslant 1023$, $1 \leqslant exp \leqslant 2046$.

Повышенная точность (80 бит): exp — 15 бит, $frac$ — 64 бита.

Пример

```
float f = 15213.0;
```

$15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$.

$M = 1.\underline{1101101101101}_2$,
$frac = \underline{1101101101101}0000000000_2$.

$e = 13, bias = 127, exp = 140 = 10001100_2$.

Результат:

0	10001100	11011011011010000000000
s	exp	$frac$

Представление нуля

Для типа `float` код порядка *exp* изменяется от 00000001 до 11111110 (значению 00000001 соответствует порядок $e = -126$, значению 11111110 — порядок $e = 127$).

Код $exp = 00000000$, $frac = 000\ldots0$ представляет нулевое значение; в зависимости от значения знакового разряда s это либо $+0$, либо -0.

А какое значение представляют коды
$exp = 00000000$, $frac \neq 000\ldots0$ и
$exp = 11111111$?

Большие числа

Пусть $exp = 111\ldots1$.

Если при этом $frac = 000\ldots0$, то коду будет соответствовать значение ∞ (со знаком s).

Если же $frac \neq 000\ldots0$, то код не будет представлять *никакое* число («значение», представляемое таким кодом, так и называется: «не число» — NaN — Not a number).

Денормализованные числа

Это числа, представляемые кодами $exp = 000\ldots0$, $frac \neq 000\ldots0$.

exp вносит в значение такого числа постоянный вклад 2^{-k-2}, $frac$ меняется от $000\ldots1$ до $111\ldots1$ и рассматривается уже не как мантисса, а как значение, умножаемое на exp.

Пример: 8-разрядные числа

s	*exp*	*frac*
1 бит	4 бита	3 бита

	s	*exp*	*frac*	E	значение	
Денормализованные числа	0	0000	000	-6	0	Близкие к 0
	0	0000	001	-6	$1/8 \times 1/64 = 1/512$	
	0	0000	010	-6	$2/8 \times 1/64 = 2/512$	
				...		
	0	0000	110	-6	$6/8 \times 1/64 = 6/512$	
	0	0000	111	-6	$7/8 \times 1/64 = 7/512$	Наибольшее денормализованное
Нормализованные числа	0	0001	000	-6	$8/8 \times 1/64 = 8/512$	Наименьшее нормализованное
	0	0001	001	-6	$9/8 \times 1/64 = 9/512$	
				...		
	0	0110	110	-1	$14/8 \times 1/2 = 14/16$	
	0	0110	111	-1	$15/8 \times 1/2 = 15/16$	Ближайшее к 1 снизу
	0	0111	000	0	$8/8 \times 1 = 1$	
	0	0111	001	0	$9/8 \times 1 = 9/8$	Ближайшее к 1 сверху
				...		
	0	1110	110	7	$14/8 \times 128 = 224$	
	0	1110	111	7	$15/8 \times 128 = 240$	Наибольшее нормализованное
	0	1111	000		$+\infty$	

Важные частные случаи

Что	*exp*	*frac*	Численное значение `float`	Численное значение `double`
Ноль	00...00	00...00	`0.0`	
Наименьшее положительное денормализованное	00...00	00...01	$2^{-23} \times 2^{-126}$	$2^{-52} \times 2^{-1022}$
Наибольшее положительное денормализованное	00...00	11...11	$(1-\varepsilon) \times 2^{-126}$	$(1-\varepsilon) \times 2^{-1022}$
Единица	01...11	00...00	`1.0`	
Наибольшее положительное нормализованное	11...10	11...11	$(2-\varepsilon) \times 2^{127}$	$(2-\varepsilon) \times 2^{1023}$

Операции над числами с плавающей точкой

$$x +_{FP} y = \text{Round}(x + y)$$

$$x \times_{FP} y = \text{Round}(x \times y),$$

где *Round* означает округление.

Выполнение операции:

- сначала вычисляется точный результат (получается более длинная мантисса, чем запоминаемая, иногда в два раза);
- потом фиксируется исключение (например, переполнение);
- потом результат округляется, чтобы поместиться в поле *frac*.

Умножение чисел с плавающей точкой

$$(-1)^{s^1} M_1 2^{e^1} \times (-1)^{s^2} M_2 2^{e^2}$$

Точный результат: $(-1)^s M 2^e$, где

- $s = s_1 \wedge s_2$,
- $M = M_1 \times M_2$,
- $e = e_1 + e_2$.

Преобразование:

- если $M \geqslant 2$, сдвиг M вправо с одновременным увеличением e;
- если e не помещается в поле *exp*, фиксируется переполнение;
- округление M, чтобы оно поместилось в поле *frac*.

Основные затраты на перемножение мантисс.

Сложение чисел с плавающей точкой

$$(-1)^{s^1} M_1 2^{e^1} + (-1)^{s^2} M_2 2^{e^2}, \text{ где } e_1 > e_2.$$

Точный результат: $(-1)^s M 2^e$.

- Порядок суммы — e_1.
- К мантиссе M_1 прибавляется $e_1 - e_2$ старших разрядов мантиссы M_2.

Преобразование:

- если $M \geqslant 2$, сдвиг M вправо с одновременным увеличением e;
- если $M < 1$, сдвиг M влево на k позиций с одновременным вычитанием k из e;
- если e не помещается в поле *exp*, фиксируется переполнение;
- округление M, чтобы оно поместилось в поле *frac*.

Пример. Сложение чисел «типа» `float`

Мантисса — 6 десятичных цифр, порядок — 2 десятичных цифры.
$0.231876 * 10^{02} + 0.645391 * 10^{-03} + 0.231834 * 10^{-01} + 0.245383 * 10^{-02} + 0.945722 * 10^{-03}$.

Сложение по порядку: $0.2321\mathbf{4}7 * 10^{02}$.

$23.1876 + 0.000645391 = 23.188245391 = 23.1882 = 0.231882 * 10^{02}$;
$23.1882 + 0.0231834 = 23.2113834 = 23.2114 = 0.232114 * 10^{02}$;
$23.2114 + 0.00245383 = 23.21385383 = 23.2138 * 10^{02}$;
$23.2138 + 0.000945722 = 23.214745722 = 23.2147 = 0.232147 * 10^{02}$.

Сложение по размеру: $0.2321\mathbf{5}7 * 10^{02}$.

$0.000645391 + 0.000945722 = 0.001591113 = 0.00159111 = 0.159111 * 10^{-02}$;
$0.00159111 + 0.00245383 = 0.00494493 = 0.494493 * 10^{-02}$;
$0.00494493 + 0.0231834 = 0.02812833 = 0.0281283 = 0.281283 * 10^{-01}$;
$0.0281283 + 23.1876 = 23.2157283 = 23.2157 = 0.232157 * 10^{02}$.

Выводы по операциям

При вычислении суммы чисел с одинаковыми знаками необходимо упорядочить слагаемые по возрастанию и складывать, начиная с наименьших слагаемых.

При вычислении суммы чисел с разными знаками необходимо сначала сложить все положительные числа, потом — все отрицательные числа и в конце выполнить одно вычитание.

Вычитание (сложение чисел с противоположными знаками) часто приводит к потере точности, которая у чисел с плавающей точкой определяется количеством значащих цифр в мантиссе (при вычитании двух близких чисел мантисса «исчезает», что ведет к резкой потере точности).
Итак, чем меньше вычитаний, тем точнее результат.

Вещественные типы языка Си

`float, double, long double`.

Традиционные арифметические операции.

Внимание: в функциях с переменным числом параметров `float` автоматически преобразуется в **`double`** (в части переменных параметров).

Режимы gcc для работы с плавающей точкой

https://gcc.gnu.org/wiki/FloatingPointMath

Детальное резюме того, что бывает в gcc, и таблица преобразований, влияющих на результат вычислений.

- `-ffast-math`: считать максимально быстро, но, возможно, нарушать стандарт IEEE-754

  Полезно для тестирования, но не распространения финальной версии программы
- `-fno-math-errno`: не устанавливать переменную `errno` как результат ошибочного выполнения математических функций

  Можно обойтись и без этого, но зависит от библиотеки Си.

  Компилятор может заменять вызовы функций инструкциями процессора (например, `sqrt`).

David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23, 1 (March 1991), 5-48.
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Режимы gcc для работы с плавающей точкой

https://gcc.gnu.org/wiki/FloatingPointMath

Детальное резюме того, что бывает в gcc, и таблица преобразований, влияющих на результат вычислений.

> `-fno-trapping-math`: считать, что вычисления с плавающей точкой не могут вызывать исключений процессора (traps)
>
> Т.е. вы гарантируете отсутствие в своем коде ситуаций, вызывающих деления на ноль, переполнения, некорректные операции.
>
> Компилятор может более свободно комбинировать, переставлять, удалять операции с плавающей точкой.

David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23, 1 (March 1991), 5-48.
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Сложность алгоритмов

Размер входа: числовая величина, характеризующая количество входных данных (например, длина битовой записи чисел-параметров алгоритма).

Сложность в наихудшем случае: функция размера входа, отражающая максимум затрат на выполнение алгоритма для данного размера:

- временнАя сложность,
- пространственная сложность (затраты памяти);
- часто оценивают не все затраты, а только самые «дорогие» операции.

Сложность в среднем: функция размера входа, отражающая средние затраты на выполнение алгоритма для входа данного размера (учет вероятностей входа).

Асимптотические оценки сложности: O-нотация (оценка сверху), точная O-оценка, Θ-оценка.

Формальная постановка задачи поиска по образцу

Даны *текст* — массив $T[N]$ длины N и *образец* — массив $P[m]$ длины $m \leqslant N$, где значениями элементов массивов T и P являются символы некоторого алфавита A.

Говорят, что образец P входит в текст T со сдвигом s, если $0 \leqslant s \leqslant N - m$ и для всех $i = 1, 2, \ldots, m$ $T[s + i] = P[i]$.

Сдвиг $s(T, P)$ называется *допустимым*, если P входит в T со сдвигом $s = s(T, P)$, и *недопустимым* в противном случае.

Задача поиска подстрок состоит в нахождении множества допустимых сдвигов $s(T, P)$ для заданного текста T и образца P.

Терминология

Пусть строки $x, y, w \in A^*, \varepsilon \in A^*$ — пустая строка.

$|x|$ — длина строки x;

xy — *конкатенация* строк x и y; $|xy| = |x| + |y|$;

если $x = wy$, то w — *префикс* (начало) x, обозначение $w \prec x$;

если $x = yw$, то w — *суффикс* (конец) x, обозначение $w \succ x$;

если w — префикс или суффикс x, то $|w| \leqslant |x|$;

отношения префикса и суффикса *транзитивны*.

Для любых $x, y \in A^*$ и любого $a \in A$ соотношения $x \succ y$ и $xa \succ ya$ равносильны.

Если $S = S[r]$ — строка длины r, то её префикс длины k, $k \leqslant r$ будет обозначаться $S_k = S[k]$; ясно, что $S_0 = \varepsilon, S_r = S$.

Лемма о двух суффиксах

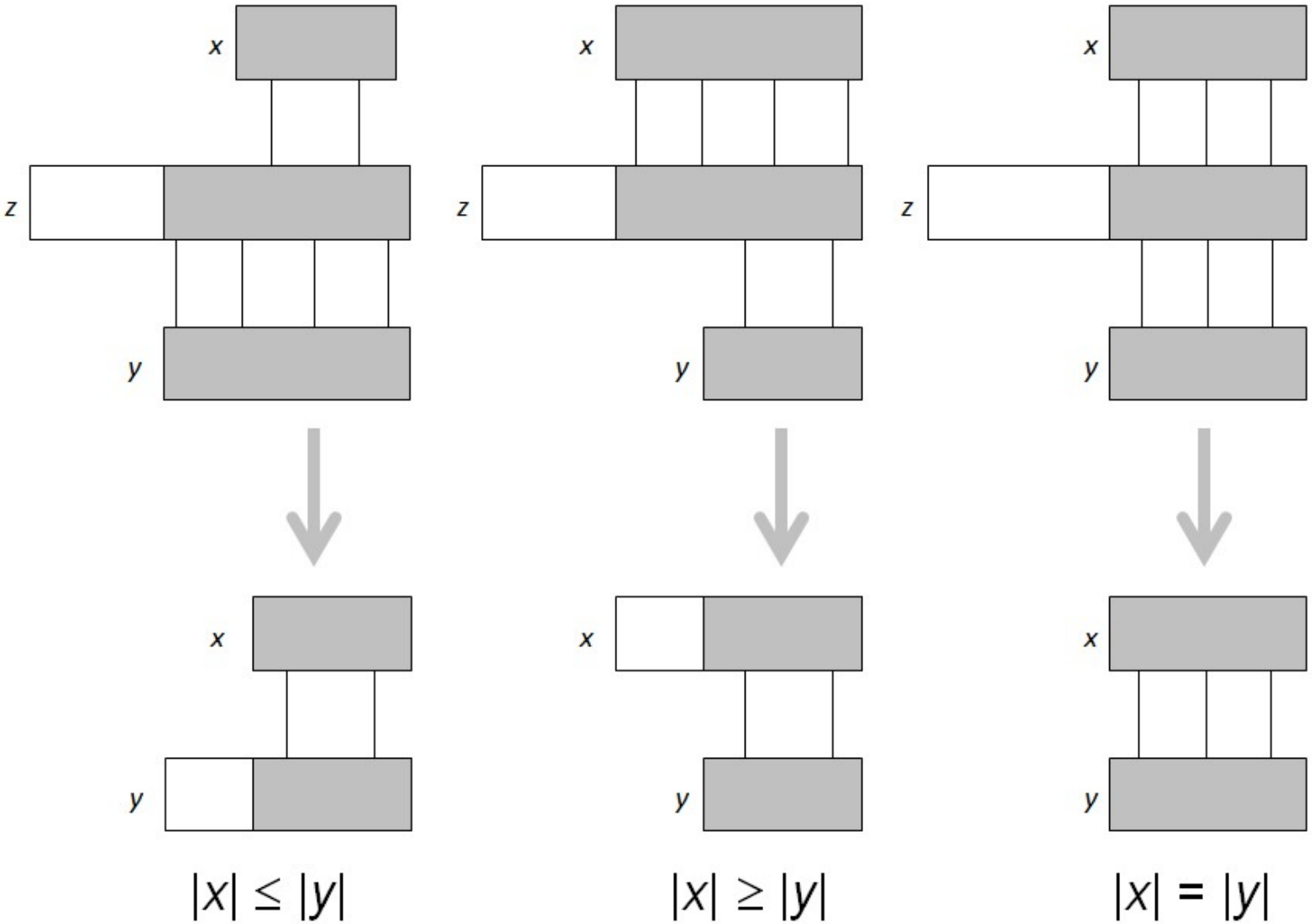
Пусть x, y и z — строки, для которых $x \succ z$ и $y \succ z$. Тогда:

если $|x| \leqslant |y|$, то $x \succ y$,

если $|x| \geqslant |y|$, то $y \succ x$,

если $|x| = |y|$, то $x = y$.

Лемма о двух суффиксах



Простой алгоритм

Проверка совмещения двух строк: посимвольное сравнение слева направо, которое прекращается (с отрицательным результатом) при первом же расхождении.
Оценка скорости сравнения строк x и y — $\Theta(t+1)$, где t — длина наибольшего общего префикса строк x и y.

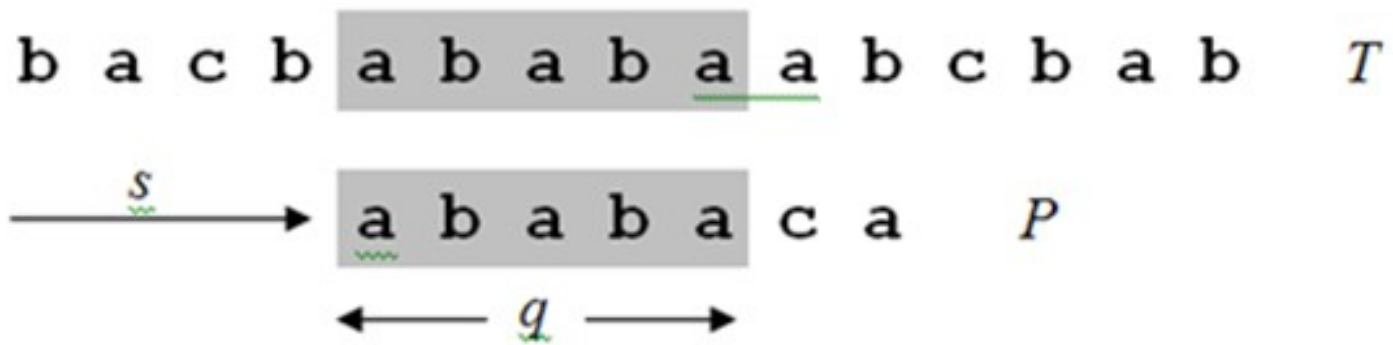
```
for (s = 0; s <= n - m; s++) {
  for (i = 0; i < m && P[i] == T[s + i]; i++)
    ;
  if (i == m)
    printf ("%d\n", s);
}
```

Время работы в худшем случае $\Theta((n-m+1)m) \sim \Theta(nm)$.
Причина: информация о тексте T, полученная при проверке сдвига s, никак не используется при проверке следующих сдвигов. Например, если для образца `dddc` сдвиг $s = 0$ допустим, то сдвиги $s = 1, 2, 3$ недопустимы, так как `T[3] == c`.

Алгоритм Кнута–Морриса–Пратта. Идея

Префикс-функция, ассоциированная с образцом P, показывает, где в строке P повторно встречаются различные префиксы этой строки. Если это известно, можно не проверять заведомо недопустимые сдвиги.

Пример. Пусть ищутся вхождения образца $P = ababaca$ в текст T. Пусть для некоторого сдвига s оказалось, что первые q символов образца совпадают с символами текста. Значит, символы текста от $T[s+1]$ до $T[s+q]$ известны, что позволяет заключить, что некоторые сдвиги заведомо недопустимы.



Алгоритм Кнута–Морриса–Пратта. Идея

Пусть $P[1..q] = T[s+1..s+q]$; каково минимальное значение сдвига $s' > s$, для которого $P[1..k] = T[s'+1..s'+k]$, где $s'+k = s+q$?

- Число s' — минимальное значение сдвига, большего s, которое совместимо с тем, что $T[s+1..s+q] = P[1..q]$. Следовательно, значения сдвигов, меньшие s', проверять не нужно.
- Лучше всего, когда $s' = s+q$, так как в этом случае не нужно рассматривать сдвиги $s+q-1, s+q-2, \ldots, s+1$.
- Кроме того, при проверке нового сдвига s' можно не рассматривать первые его k символов образца: они заведомо совпадут.

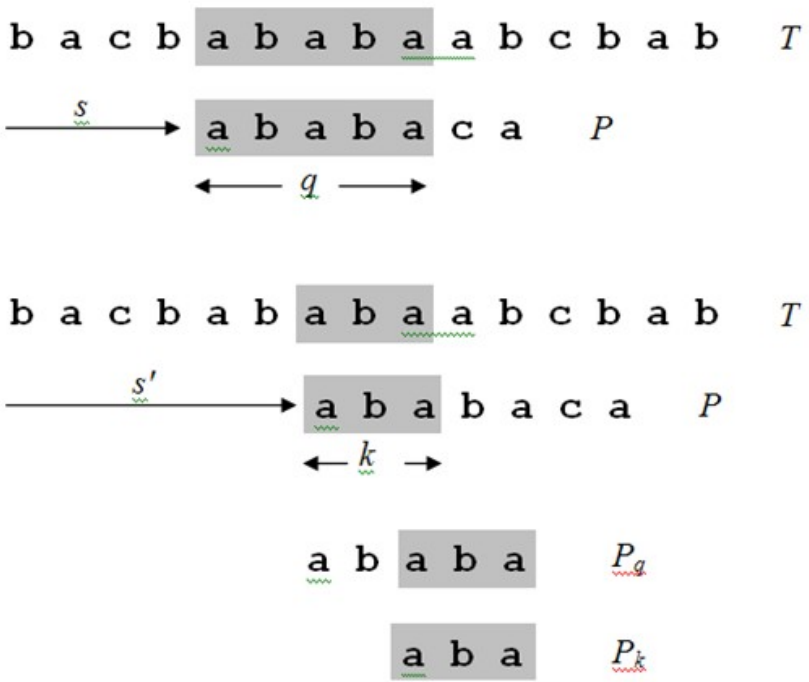
Чтобы найти s', достаточно знать образец P и число q:
$T[s'+1..s'+k]$ — суффикс P_q, поэтому k — это наибольшее число, для которого P_k является суффиксом P_q. Зная k (число символов, заведомо совпадающих при проверке нового сдвига s'), можно вычислить по формуле $s' = s + (q-k)$.

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

Определение. Префикс-функцией, ассоциированной со строкой $P[1..m]$, называется функция $\pi : 1, 2, \ldots, m \to 0, 1, \ldots, m-1$, определённая следующим образом:

$$\pi[q] = \max\{k : k < q \wedge P_k \succ P_q\}.$$

Иными словами, $\pi[q]$ — длина наибольшего префикса P, являющегося суффиксом P_q.



Алгоритм Кнута–Морриса–Пратта. Префикс-функция

```
void prefix_func (char *pat, int *pi, int m) {
  int k, q;

  /*  Считаем, что pat и pi нумеруются от 1.  */
  pi[1] = 0; k = 0;
  for (q = 2; q <= m; q++) {
    while (k > 0 && pat[k + 1] != pat[q])
      k = pi[k];
    if (pat[k + 1] == pat[q])
      k++;
    pi[q] = k;
  }
}
```

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

Лемма 1. Обозначим $\pi^*[q] = \{q, \pi[q], \pi^2[q], \ldots, \pi^t[q]\}$, где $\pi^i[q]$ есть i-я итерация префикс-функции, $\pi^t[q] = 0$. Пусть P — строка длины m с префикс-функцией π. Тогда для всех $q = 1, 2, \ldots, m$ имеем $\pi^*[q] = \{k : P_i \succ P_q\}$.
Лемма показывает, что при помощи итерирования префикс-функции можно для данного q найти все такие k, что P_k является суффиксом P_q.

Доказательство. Во-первых, покажем, что если i принадлежит $\pi^*[q]$, то P_i является суффиксом P_q.
Действительно, $P_{\pi[i]} \succ P_i$ по определению префикс-функции, так что каждый следующий член последовательности $P_i, P_{\pi[i]}, P_{\pi[\pi[i]]}, \ldots$ является суффиксом всех предыдущих.

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

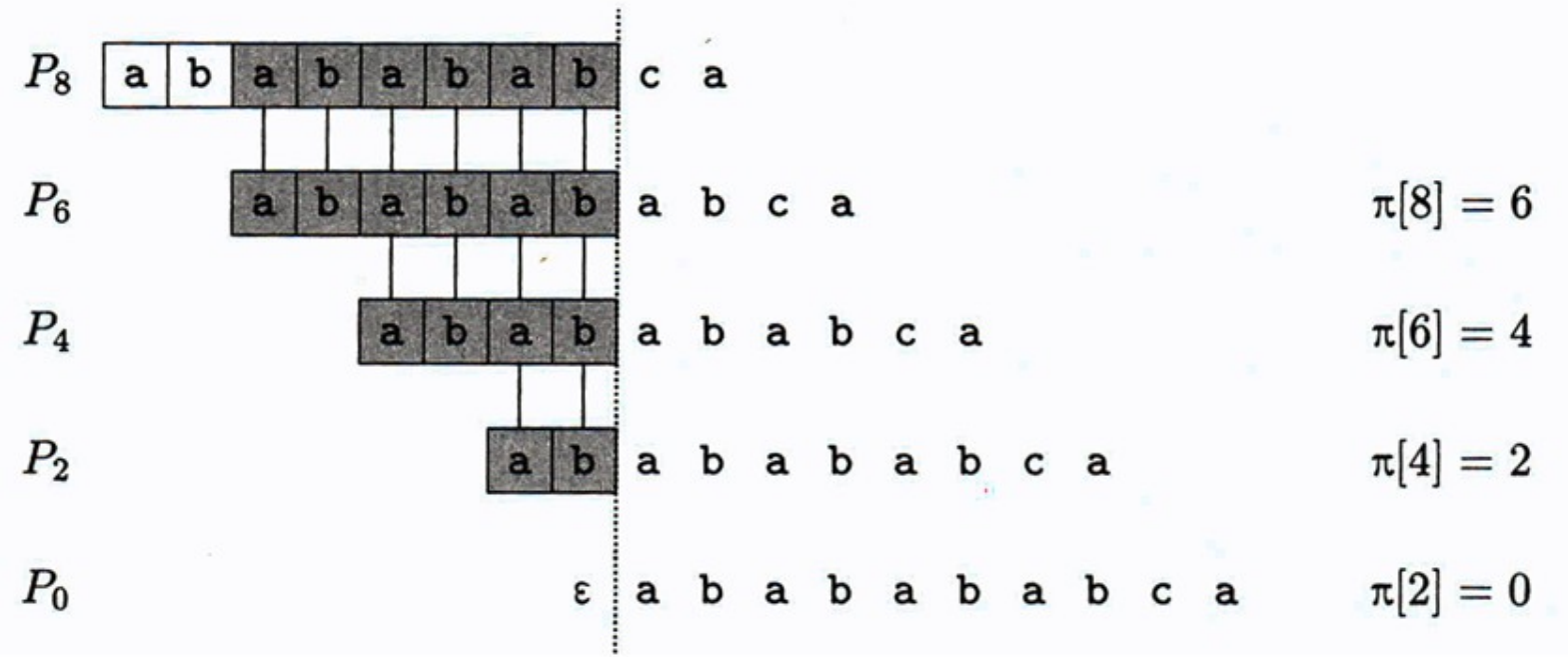
Покажем, что наоборот, если P_i является суффиксом P_q, то i принадлежит $\pi^*[q]$.

Расположим все P_i, являющиеся суффиксами P_q, в порядке уменьшения i (длины): $P_{i_1}, P_{i_2}, \ldots$. Покажем по индукции, что $P_{i_k} = \pi^k[q]$.

> База индукции ($k = 1$): для максимального префикса P_i, являющегося суффиксом P_q, по определению $i = \pi[q]$.
>
> Шаг индукции: если $P_{i_k} = \pi^k[q]$, то по определению $j = \pi[\pi^k[q]]$ соответствует максимальный префикс P_j, который является суффиксом P_{i_k}. Обе строки P_j и P_{i_k} есть суффиксы P_q по построению. Таким максимальным префиксом из оставшихся $P_{i_{k+1}}, P_{i_{k+2}}, \ldots$ по построению является префикс $P_{i_{k+1}}$, то есть $j = i_{k+1}$.

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

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



$$\pi^*[8] = \{8, 6, 4, 2, 0\}$$

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

Лемма 2. Пусть P — строка длины m с префикс-функцией π. Тогда для всех $q = 1, 2, \ldots, m$, для которых $\pi[q] > 0$, имеем $\pi[q] - 1 \in \pi^*[q-1]$.

Доказательство. Если $k = \pi[q] > 0$, то P_k является суффиксом P_q по определению префикс-функции. Следовательно, P_{k-1} является суффиксом P_{q-1}.
Тогда по Лемме 1 $k - 1 \in \pi^*[q-1]$, т.е. $\pi[q] - 1 \in \pi^*[q-1]$.

Определим множества E_{q-1} как

$$E_{q-1} = \{k : k \in \pi^*[q-1] \wedge P[k+1] = P[q]\}.$$

Множество E_{q-1} состоит из таких k, что P_k является суффиксом P_{q-1}, и за ними идут одинаковые буквы $P[k+1]$ и $P[q]$.
Из определения вытекает, что P_{k+1} есть суффикс P_q.

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

Следствие 1. Пусть P — строка длины m с префикс-функцией π. Тогда для всех $q = 2, 3, \ldots, m$

$$\pi[q] = \begin{cases} 0, \text{если } E_{q-1} \text{ пусто;} \\ 1 + \max\{k \in E_{q-1}\}, \text{если } E_{q-1} \text{не пусто.} \end{cases}$$

Доказательство. Если $r = \pi[q] \geqslant 1$, то $P[r] = P[q]$ и по Лемме 2 $r - 1 = \pi[q] - 1 \in \pi^*[q-1]$. Раз $P[r] = P[q]$, то $P[(r-1)+1] = P[q]$. Поэтому $r - 1 \in E_{q-1}$ из определения E_{q-1}, и из $\pi[q] \geqslant 1$ следует непустота E_{q-1}.

Следовательно, если E_{q-1} пусто, то $\pi[q] = 0$ (от противного).

Если же $k \in E_{q-1}$, то P_{k+1} есть суффикс P_q (по определению), тем самым $\pi[q] \geqslant k + 1$ и $\underline{\pi[q] \geqslant 1 + \max\{k \in E_{q-1}\}}$.

То есть, если E_{q-1} не пусто, то префикс-функция положительна. Но тогда $\pi[q] - 1 \in E_{q-1}$, и $\pi[q] - 1$ не больше максимума из E_{q-1}, то есть $\underline{\pi[q] \leqslant 1 + \max\{k \in E_{q-1}\}}$.

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

```
1  void prefix_func (char *pat, int *pi, int m) {
2    int k, q;
3
4    /*  Считаем, что pat и pi нумеруются от 1.  */
5    pi[1] = 0; k = 0;
6    for (q = 2; q <= m; q++) {
7      while (k > 0 && pat[k + 1] != pat[q])
8        k = pi[k];
9      if (pat[k + 1] == pat[q])
10       k++;
11     pi[q] = k;
12   }
13 }
```

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

Теорема 1. Функция `prefix_func` правильно вычисляет префикс-функцию π.

Доказательство.
Покажем, что при входе в цикл функции $k = \pi[q-1]$.

База индукции. При `q = 2 k = 0`, `pi[q-1] = pi[1] = 0`.

Шаг индукции. Пусть при входе в цикл функции $k = \pi[q-1]$.
Код на строках 7-8

```
while (k > 0 && pat[k + 1] != pat[q])
        k = pi[k];
```

находит наибольший элемент E_{q-1} (т.к. цикл перебирает в порядке убывания элементы из $\pi^*[q-1]$ и для каждого проверяет условие `pat[k + 1] != pat[q]`).

Алгоритм Кнута–Морриса–Пратта. Префикс-функция

Теорема 1. Функция `prefix_func` правильно вычисляет префикс-функцию π.

Доказательство.
После выхода из цикла на строках 7-8

```
while (k > 0 && pat[k + 1] != pat[q])
  k = pi[k];
```

если `pat[k + 1] == pat[q]`, то выполняется код на строке 10

```
k++;
```

что из Следствия 1 даёт нам $\pi[q]$;

если `pat[k + 1] != pat[q]`, то `k == 0`, множество E_{q-1} пусто и $\pi[q] = 0$.

Алгоритм Кнута–Морриса–Пратта. Функция kmp

```
void kmp (char *text, char *pat, int m, int n) {
  int q;
  int pi[m + 1];  /* VLA-массив */

  /*  Считаем, что pat и pi нумеруются от 1. */
  prefix_func (pat, pi, m);
  q = 0;
  for (i = 1; i <= n; i++) {
    while (q > 0 && pat[q + 1] != text[i])
      q = pi[q];
    if (pat[q + 1] == text[i])
      q++;
    if (q == m) {
      printf ("образец␣входит␣со␣сдвигом␣%d\n", i - m);
      q = pi[q];
    }
}
```

Алгоритм Кнута–Морриса–Пратта. Функция `kmp`

Алгоритм КМП для подстроки P и текста эквивалентен вычислению префикс-функции для строки $Q = P\#T$, где $\#$ — символ, заведомо не встречающийся в обеих строках.

Длина максимального префикса Q, являющегося её суффиксом (т.е. значение префикс-функции), не превосходит длины P.

Допустимый сдвиг обнаруживается в тот момент, когда очередное вычисленное значение префикс-функции совпадает с длиной подстроки P (условие `if (q == m)`).

В явном виде объединённая строка не строится!

Алгоритм Кнута–Морриса–Пратта. Функция kmp

Теорема 2. Функция kmp работает правильно.
Формальное доказательство осуществляется по аналогии с доказательством Теоремы 1, где множества, подобные E_{q-1}, строятся для строки-текста, а не строки-образца.

Свойства префикс-функции часто используются и в других задачах (кроме поиска подстроки в строке).
Полезной оказывается Лемма 1: итерированием префикс-функции можно найти все префиксы строки, являющиеся её суффиксами.

Алгоритм Кнута–Морриса–Пратта. Время работы

Функция `prefix_func` выполняет $\leqslant (m-1)$ итераций цикла `for`. Стоимость каждой итерации можно считать равной $O(1)$, а стоимость всей процедуры $O(m)$.

> Каждая итерация цикла `while` (строки 7-8) уменьшает k.
>
> Увеличивается k только в строке 10 не более одного раза на итерацию цикла `for` (строки 6-11).
>
> Следовательно, операций уменьшения не больше, чем итераций цикла `for`, то есть $\leqslant (m-1)$ на весь цикл и $O(1)$ на итерацию в среднем.

Аналогично, функция `kmp` выполняет $\leqslant (n-1)$ итераций, и её стоимость (без учета вызова `prefix_func`) есть $O(n)$. Следовательно, время выполнения всей процедуры — $O(m+n)$.