БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

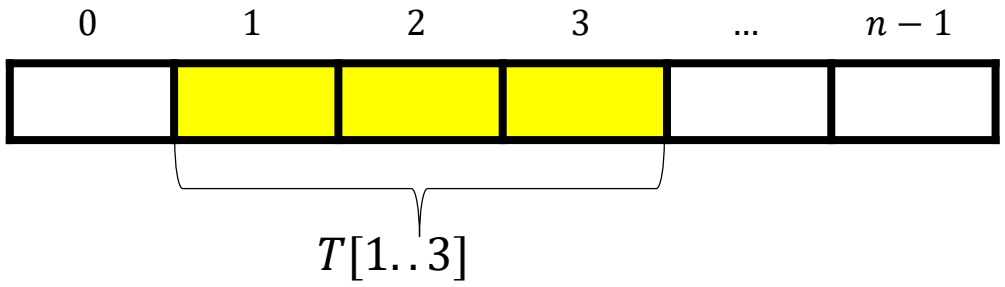
Строковые алгоритмы и структуры данных

Основные определения.

Задано некоторое конечное непустое множество символов Σ, называемое алфавитом ($|\Sigma|$ – мощность алфавита).

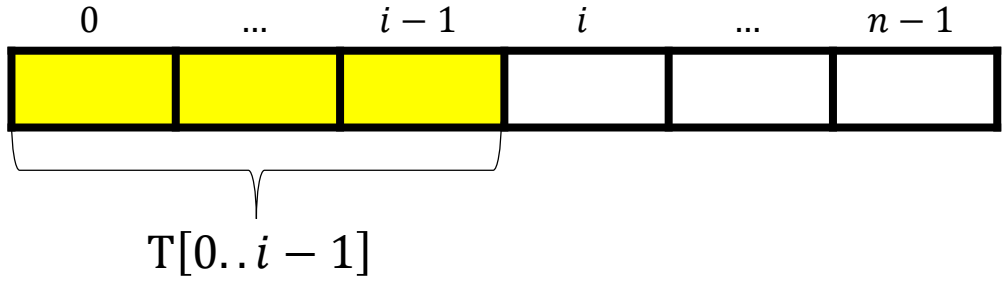
Строка - произвольная конечная последовательность символов из алфавита: $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1}), t_i \in \Sigma$.

Подстрока – непрерывная последовательность строки $\mathrm{T}[i..j] = (t_i, t_1, \ldots, t_j), t_i \in \Sigma$.

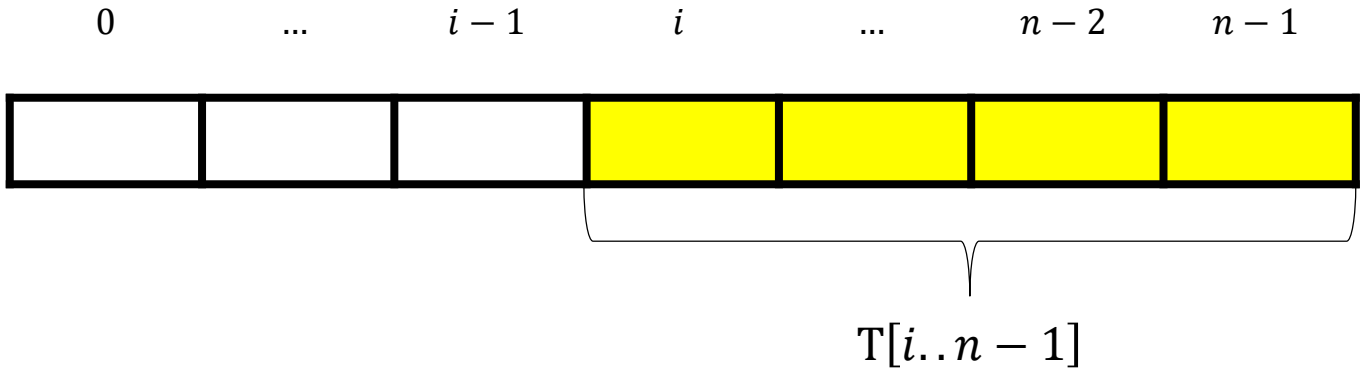


Если не оговорено иное, то при $i > j$ считаем, что подстроки $\mathrm{T}[i..j]$ не существует.

Подстрока $\mathrm{T}[0..i-1]$, которая состоит из первых i символов строки, называется **префиксом** ($\mathrm{T}[0..i-1]$ - префикс длины i).



Подстрока $\mathrm{T}[i..n-1]$, которая состоит из последних $n-i$ символов строки, называется i-ым **суффиксом**.



Собственный суффикс/префикс - не совпадающий со всей строкой.

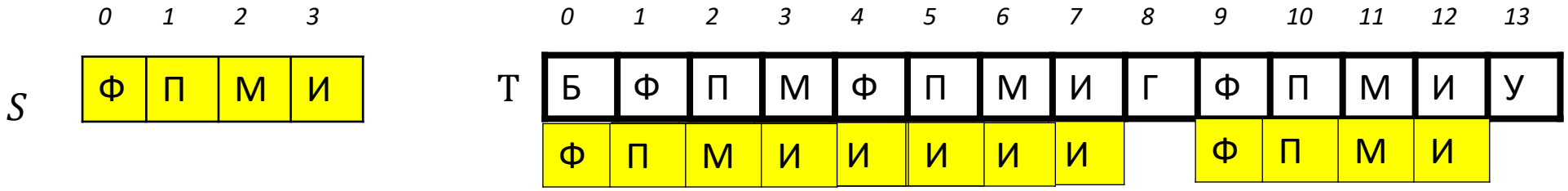
Задача поиска вхождений подстроки в строке

- Пусть есть непустая строка $\mathrm{T} = (t_0, t_1, \dots, t_{n-1})$, называемая **текстом**, и непустая строка $S = (s_0, s_1, \dots, s_{m-1})$, называемая **образцом**.

  Задача поиска вхождений подстроки в строке состоит в том, чтобы найти все подстроки строки T, которые совпадают с образцом S.

  - **полиномиальное хеширование строк**
  - **префиксная функция**
  - **Z-функция**

Задачу поиска всех подстрок строки $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1})$, которые совпадают с образцом $S = (s_0, s_1, \ldots, s_{m-1})$ можно решить непосредственно – последовательно перебирать в T начальный символ для подстроки, а затем последовательно сравнивать подстроку T с образцом S.



Трудоемкость поиска такого алгоритма - $\mathrm{O}(n \cdot m)$.

Существуют алгоритмы, которые решают эту задачу за время $\mathrm{O}(n + m)$, которые мы далее и рассмотрим.

Предположим, что на вход поступает строка:

$$S = (s_0, s_1, \dots, s_n)$$

Можно считать, что элементы строки S – целые числа от 1 до L,
где $L = |\Sigma|$ – размер алфавита:

> для этого каждый символ строки S заменяем его порядковым номером в алфавите $(s_i - 'a' + 1)$.

$$S = (a, b, c, d, b) \rightarrow (1,2,3,4,2)$$

Прямой полиномиальный хеш (хеширование **слева направо**)	**Обратный полиномиальный хеш** (хеширование **справа налево**)
это число → $\boldsymbol{h_{\text{пр.}}(S) = (s_0 + s_1 \cdot k^1 + s_2 \cdot k^2 + \cdots + s_n \cdot k^n) \bmod m}$	$\boldsymbol{h_{\text{обр.}}(S) = (s_0 \cdot k^n + s_1 \cdot k^{n-1} + s_2 \cdot k^{n-2} + \cdots + s_n) \bmod m}$

Рекомендуют выбирать следующие константы:

$\boldsymbol{k > L}$ (натуральное число, чуть больше размера алфавита $L = |\Sigma|$);

$\boldsymbol{m}$ – достаточно большое целое число, например, $m = 2^{64}$;

$\boldsymbol{k}$ и $\boldsymbol{m}$ должны быть взаимно просты, например, если m=2^{64}, то k – нечётное число.

Предположим, что алфавит небольшой, $L = |\Sigma| = 9$.

Пусть $k = 10$ и $m = 1\ 000\ 007$.

$\boldsymbol{h_{\text{пр.}}}\big((a,b,c,d,b) \rightarrow (1,2,3,4,2)\big)$=24 321	$\boldsymbol{h_{\text{обр.}}}\big((a,b,c,d,b) \rightarrow (1,2,3,4,2)\big)$=12 342

Две одинаковые строки будут иметь обязательно один хеш, а вероятность коллизий крайне мала.

Вычислить значения функций
$\boldsymbol{h}_{\text{пр.}}(S)$ и $\boldsymbol{h}_{\text{обр.}}(S)$
можно за линейное от длины строки S время
$\Theta(n)$.

1) **Прямое полиномиальное хеширование**:

$$\boldsymbol{h_{\text{пр.}}(S) = (s_0 + s_1 \cdot k^1 + s_2 \cdot k^2 + \cdots + s_n \cdot k^n) \bmod m.}$$

Выполним сначала за время $\Theta(n)$ предподсчёт нужных степеней.

Обозначим $\boldsymbol{k[i]} = k^i \bmod \boldsymbol{m}$,
тогда:

$$\begin{cases} \boldsymbol{k[0]} = 1 \\ \boldsymbol{k[i]} = (\boldsymbol{k[i-1]} \cdot \boldsymbol{k}) \bmod \boldsymbol{m},\ \boldsymbol{i = 1, \dots, n.} \end{cases}$$

Обозначим $\boldsymbol{h_{\text{пр.}}(0: i)}$ - хеш для префикса строки $S[\boldsymbol{0..i}]$:

$$\boldsymbol{h_{\text{пр.}}(0: i) = (\underbrace{s_0 + s_1 \cdot k^1 + s_2 \cdot k^2 + \cdots + s_i \cdot k^i}) \bmod m.}$$

Тогда

$$\begin{cases} \boldsymbol{h_{\text{пр.}}(0: 0) = s_0} \\ \boldsymbol{h_{\text{пр.}}(0: i) = \left(h_{\text{пр.}}(0: i-1) + s_i \cdot k[i]\right) \bmod m,} \quad \boldsymbol{i = 1, \dots, n} \end{cases}$$

Так как значение хеша каждого следующего префикса выражается через значение хеша предыдущего префикса, то линейным проходом по всей строке за время $\Theta(n)$ можно вычислить хеши для всех префиксов строки.

2) **Обратное полиномиальное хеширование**:

$$\boldsymbol{h}_{\text{обр.}}(\boldsymbol{s}) = (\boldsymbol{s_0} \cdot \boldsymbol{k^n} + \boldsymbol{s_1} \cdot \boldsymbol{k^{n-1}} + \boldsymbol{s_2} \cdot \boldsymbol{k^{n-2}} + \cdots + \boldsymbol{s_n}) \bmod \boldsymbol{m}$$

$\boldsymbol{h}_{\text{обр.}} = \boldsymbol{0}$

$\boldsymbol{i} = \boldsymbol{0}, \ldots, \boldsymbol{n}$

$\quad \boldsymbol{h}_{\text{обр.}} = \left(\boldsymbol{h}_{\text{обр.}} \cdot \boldsymbol{k} + \boldsymbol{s_i}\right) \bmod \boldsymbol{m},$

$\boldsymbol{h}_{\text{обр.}} = \boldsymbol{0}$

$i = 0 \quad \boldsymbol{h}_{\text{обр.}} = (0 \cdot k + s_0) \bmod m$

$i = 1 \quad \boldsymbol{h}_{\text{обр.}} = (s_0 \cdot k + s_1) \bmod m$

$i = 2 \quad \boldsymbol{h}_{\text{обр.}} = (s_0 \cdot k^2 + s_1 \cdot k^1 + s_2) \bmod m$

…

$i = n \quad \boldsymbol{h}_{\text{обр.}} = (s_0 \cdot k^n + \ldots + s_{n-1} \cdot k^1 + s_n) \bmod m$

Так как хеш – это значение многочлена, то для многих строковых операций можно быстро пересчитывать хеш результата.

Конкатенация двух строк

$\mathrm{X} = (\mathrm{x}_0, \mathrm{x}_1, \dots, \mathrm{x}_n)$

$Y = (y_0, y_1, \dots, y_r)$

Конкатенация строк X и Y, это строка, которая получается в результате приписывания последовательности символов строки Y после последовательности символов строки X:

$$\mathrm{X} \cdot Y = (\mathrm{x}_0, \mathrm{x}_1, \dots, \mathrm{x}_n, y_0, y_1, \dots, y_r)$$

$$h_{\text{пр.}}(\mathrm{X}) = (\mathrm{x}_0 + \mathrm{x}_1 \cdot k^1 + \mathrm{x}_2 \cdot k^2 + \cdots + \mathrm{x}_n \cdot k^n) \bmod m$$

$$h_{\text{пр.}}(Y) = (y_0 + y_1 \cdot k^1 + y_2 \cdot k^2 + \cdots + y_r \cdot k^r) \bmod m$$

$$h_{\text{пр.}}(\mathrm{X} \cdot Y) = (\underbrace{\mathrm{x}_0 + \mathrm{x}_1 \cdot k^1 + \mathrm{x}_2 \cdot k^2 + \cdots + \mathrm{x}_n \cdot k^n}_{} + \underbrace{y_0 \cdot k^{n+1} + y_1 \cdot k^{n+2} + \cdots + y_r \cdot k^{n+r}}_{}) \bmod m$$

$$h_{\text{пр.}}(\mathrm{X} \cdot Y) = (\mathrm{x}_0 + \mathrm{x}_1 \cdot k^1 + \mathrm{x}_2 \cdot k^2 + \cdots + \mathrm{x}_n \cdot k^n + y_0 \cdot k^{n+1} + y_1 \cdot k^{n+2} + \cdots + y_r \cdot k^{n+r}) \bmod m$$

$$h_{\text{пр.}}(\mathrm{X} \cdot Y) = (\underbrace{(\mathrm{x}_0 + \cdots + \mathrm{x}_n \cdot k^n) \bmod m}_{h_{\text{пр.}}(\mathrm{X})} + k^{n+1} \cdot \underbrace{(y_0 + \cdots + y_r \cdot k^r) \bmod m)}_{h_{\text{пр.}}(Y)}) \bmod m$$

$$\boxed{h_{\text{пр.}}(\mathrm{X} \cdot Y) = (h_{\text{пр.}}(\mathrm{X}) + k^{|\mathrm{X}|} \cdot h_{\text{пр.}}(Y)) \bmod m}$$

Хеш для подстроки $\mathrm{X}[l..r] = (x_l, x_{l+1}, \dots, x_r)$

$$h_{\text{пр.}}(l{:}\,r) = \mathrm{x}_{\boldsymbol{l}} + \mathrm{x}_{\boldsymbol{l+1}} \cdot \boldsymbol{k^1} + \cdots + \mathrm{x}_{\boldsymbol{r}} \cdot \boldsymbol{k^{r-l}}$$

как вычислить быстрее?

Предположим, что $m = 2^{64}$ и значение полинома помещается в 64 бита. Это равносильно тому, что на компьютере можно выполнять все операции над 64-битными числами и не задумываться о том, чтобы брать остаток.

Но в C++ для этого необходимо использовать не `int64_t` (обычно, синоним `long long`), а `uint64_t` (`unsigned long long`), потому что переполнение знаковых целочисленных типов в C++ — опасный пример неопределённого поведения.

$$h_{\text{пр.}}(l:r) = \mathrm{x}_{\boldsymbol{l}} + \mathrm{x}_{\boldsymbol{l+1}} \cdot \boldsymbol{k^1} + \cdots + \mathrm{x}_{\boldsymbol{r}} \cdot \boldsymbol{k^{r-l}}$$

$$h_{\text{пр.}}(l:r) \cdot k^l = \mathrm{x}_{\boldsymbol{l}} \cdot k^l + \mathrm{x}_{\boldsymbol{l+1}} \cdot \boldsymbol{k^{l+1}} + \cdots + \mathrm{x}_{\boldsymbol{r}} \cdot \boldsymbol{k^r}$$

$$h_{\text{пр.}}(\mathrm{X}) = \mathrm{x}_0 + \mathrm{x}_1 \cdot k^1 + \cdots + \mathrm{x}_{l-1} \cdot k^{l-1} + \mathrm{x}_{\boldsymbol{l}} \cdot \boldsymbol{k^l} + \mathrm{x}_{\boldsymbol{l+1}} \cdot \boldsymbol{k^{l+1}} + \cdots + \mathrm{x}_{\boldsymbol{r-1}} \cdot \boldsymbol{k^{r-1}} + \mathrm{x}_{\boldsymbol{r}} \cdot \boldsymbol{k^r} + \cdots + \mathrm{x}_n \cdot k^n$$

$$h_{\text{пр.}}(l:r) \cdot k^l = h_{\text{пр.}}(0:r) - h_{\text{пр.}}(0:l-1)$$

$$h_{\text{пр.}}(l:r) = \frac{h_{\text{пр.}}(0:r) - h_{\text{пр.}}(0:l-1)}{k^l}$$

Из формулы следует, что для вычисления хешей любой подстроки, необходимо знать лишь хеши всех префиксов этой строки.

Теперь, чтобы сравнить на равенство две любые подстроки строки S, можно сравнить соответствующие хеши этих подстрок. Так как предполагается, что значение полинома помещается в 64 бита, то считаем, что сравнение двух хешей будет выполнено за $O(1)$.

Для реализации математической формулы

$$h_{\text{пр.}}(l:r) = \left(\frac{h_{\text{пр.}}(0:r) - h_{\text{пр.}}(0:l-1)}{k^l}\right) \bmod m,$$

необходимо найти обратный элемент для k^l по модулю m.

Для того, чтобы найти обратный элемент для p по модулю m, нужно вычислить значение функции Эйлера $\varphi(m)$ (число взаимно простых с m чисел) и возвести p в степень $\varphi(m) - 1$ (https://ru.m.wikipedia.org/wiki/Обратное_по_модулю_число).

В случае полиномиального хеша с $m = 2^{64}$ обратным числу p будет число $p^{2^{63}-1}$.

Вычислить величину $p^{2^{63}-1}$ можно бинарным возведением в степень:

$$p^n = \begin{cases} (p^{n/2})^2, n - \text{чётное} \\ p \cdot (p^{n-1/2})^2, n - \text{нечётное} \end{cases}$$

Если предподсчитать степени обратного элемента по выбранному модулю m, то сравнение на равенство двух подстрок можно выполнять за O(1).

Так как деление достаточно сложная операция, то для сравнения подстрок на равенство на практике используют формулу без деления (приведение к одной степени).

Предположим, что $l_2 > l_1$, тогда

$$h_{\text{пр.}}(l_1:r_1) \stackrel{?}{=} h_{\text{пр.}}(l_2:r_2)$$

$$k^{l_2-l_1} \cdot (h_{\text{пр.}}(0:r_1) - h_{\text{пр.}}(0:l_1-1)) \stackrel{?}{=} (h_{\text{пр.}}(0:r_2) - h_{\text{пр.}}(0:l_2-1))$$

Так как деление достаточно сложная операция, то для сравнения подстрок на равенство на практике используют формулу без деления (приведение к одной степени).

Предположим, что $l_2 > l_1$, тогда для того, чтобы $h_{\text{пр.}}(l_1 : r_1) = h_{\text{пр.}}(l_2 : r_2)$

надо, чтобы $k^{l_2 - l_1} \cdot (h_{\text{пр.}}(0 : r_1) - h_{\text{пр.}}(0 : l_1 - 1)) = (h_{\text{пр.}}(0 : r_2) - h_{\text{пр.}}(0 : l_2 - 1))$

Действительно, по определению прямого хеширования

(1) $k^{l_1} \cdot h_{\text{пр.}}(l_1 : r_1) = h_{\text{пр.}}(0 : r_1) - h_{\text{пр.}}(0 : l_1 - 1)$

(2) $k^{l_2} \cdot h_{\text{пр.}}(l_2 : r_2) = h_{\text{пр.}}(0 : r_2) - h_{\text{пр.}}(0 : l_2 - 1)$

Домножим (1) на $k^{l_2 - l_1}$

$$k^{l_2 - l_1} \cdot k^{l_1} \cdot h_{\text{пр.}}(l_1 : r_1) = k^{l_2 - l_1} \cdot (h_{\text{пр.}}(0 : r_1) - h_{\text{пр.}}(0 : l_1 - 1))$$

(3) $$k^{l_2} \cdot h_{\text{пр.}}(l_1 : r_1) = k^{l_2 - l_1} \cdot (h_{\text{пр.}}(0 : r_1) - h_{\text{пр.}}(0 : l_1 - 1))$$

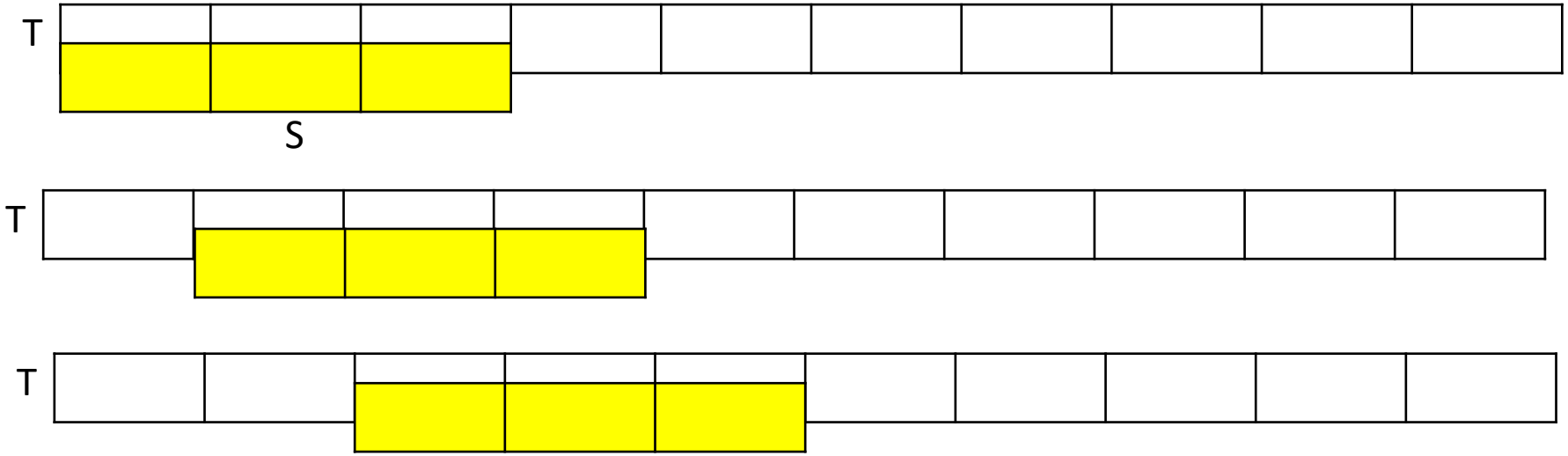
Из (2) и (3) получаем, что для того, чтобы $h_{\text{пр.}}(l_1 : r_1) = h_{\text{пр.}}(l_2 : r_2)$, надо, чтобы
$k^{l_2 - l_1} \cdot (h_{\text{пр.}}(0 : r_1) - h_{\text{пр.}}(0 : l_1 - 1)) = (h_{\text{пр.}}(0 : r_2) - h_{\text{пр.}}(0 : l_2 - 1))$

Применение хеширования строк

- ✓ сравнение двух подстрок на равенство: если выполнить предподсчёт всех префиксов, то сравнение на равенство выполняется за $O(1)$;

- ✓ лексикографически сравнить две строки на больше (меньше): подсчитаем хеши всех префиксов строки S, тогда можно за время $O(\log |S|)$ лексикографически сравнить две строки на больше (меньше): дихотомией ищут такой наибольший индекс i, что префиксы обеих строк длины i равны, после чего сравнивают i+1 символ соответствующих строк;

- ✓ за время $O(|T| + |S|)$ можно найти все вхождения образца S в текст T (алгоритм Рабина (Rabin)-Карпа (Karp), 1987 год): сначала для всех префиксов текста T вычислить хеши, вычислить хеш для образца S; затем, двигаясь слева направо по тексту T, пробуем накладывать окно длины $|S|$.



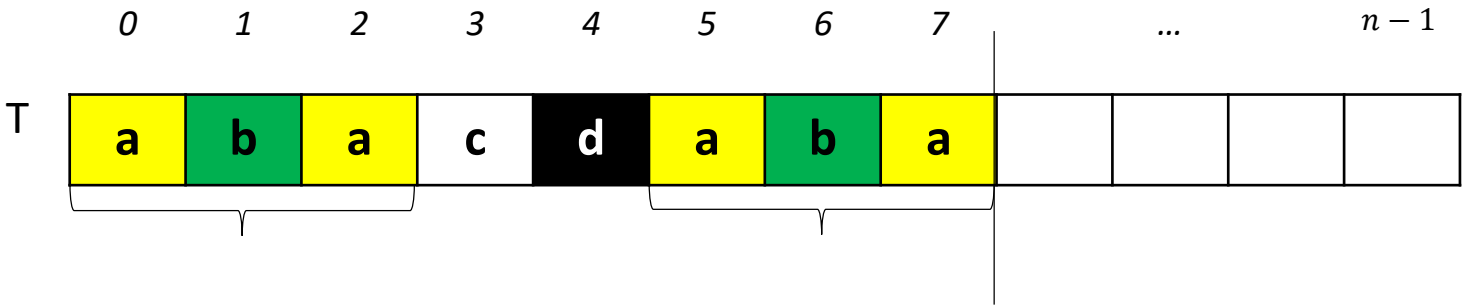
Префиксная функция

Префиксная функция P от строки $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1})$

это массив целых чисел $P[0..n-1]$,
где $p[i]$ – длина максимального префикса подстроки $T[0..i]$, который совпадает с суффиксом этой подстроки, но не равен всей подстроке

$$\begin{cases} p[i] = 0 \\ p[i] = \max\limits_{0 \le k \le i} \{k | T[0..k-1] = T[i-k+1..i]\}, i = 1,..,n-1 \end{cases}$$

другими словами - наибольшее число $k \le i$, такое, что первые k символов подстроки $\mathrm{T}[0..i]$ совпадают с ее последними k символами;



$p[7] = 3$

Вычислить префиксную функцию для всех позиций строки T:

0	1	2	3	4	5	6
a	b	a	c	a	b	a

i	0	0	1	0	1	2	0	1	2	3	0	1	2	3	4
T	a	a	b	a	b	a	a	b	a	c	a	b	a	c	a
P	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1

0	1	2	3	4	5
a	b	a	c	a	b
0	0	1	0	1	2

0	1	2	3	4	5	6
a	b	a	c	a	b	a
0	0	1	0	1	2	3

Наивный алгоритм вычисления префиксной функции для всех позиций строки T (в основе которого лежит сравнение подстрок) работает за время

$$\Theta(|T|^3) = \Theta(n^3).$$

Эффективный алгоритм вычисления префиксной функции для всех позиций строки $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1})$, который работает за линейное от длины строки время

(как основной элемент решения задачи поиска всех вхождений некоторого образца в текст)

был разработан Д.Э. **Кнуттом** и В.Р. **Праттом**,

независимо от них Д.Х. **Мориссом**.

Совместно тремя учёными алгоритм **Кнута, Морисса, Пратта** (**КМП**-алгоритм) был опубликован в **1977** году

Donald Knuth; James H. Morris, Jr, Vaughan Pratt. Fast pattern matching in strings (англ.) // SIAM Journal on Computing (англ.)рус. : journal. — 1977. — Vol. 6, no. 2. — P. 323—350.

Дональд Эрвин **Кнут**

Donald Ervin Knuth

Дата рождения	1938
Место рождения	США[1]
Научная сфера	математика, программирование, информатика
Место работы	Стэнфордский университет
Известен как	автор классических трудов «Искусство программирования», «Конкретная математика» и др.

Вон Рональд **Пратт**

Vaughan Ronald Pratt

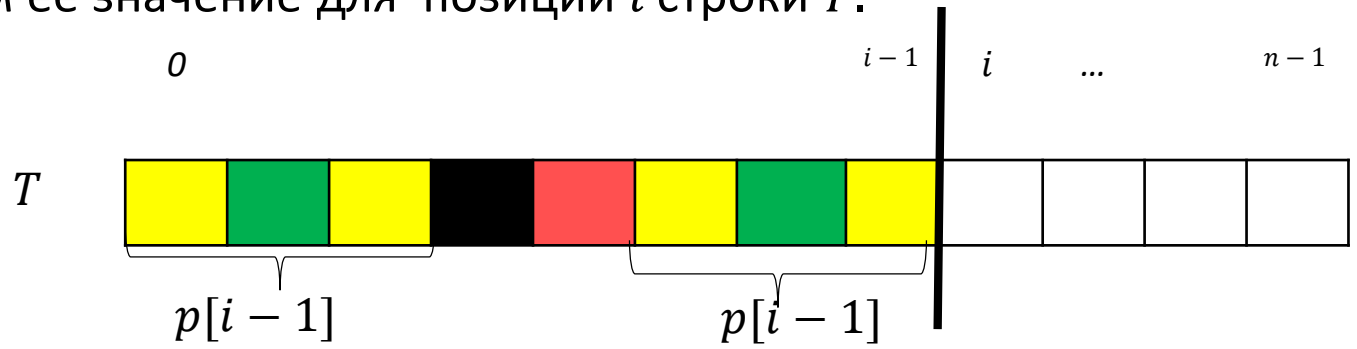
Дата рождения	1944
Место рождения	Австралия
Научная сфера	математика, программирование, информатика
Место работы	Стэнфордский университет
Известен как	автор сертификата простоты Пратта и синтаксического анализатора Пратта

Джеймс Хирам **Моррис**

James Hiram Morris

Дата рождения	1941
Место рождения	США
Место работы	•Университет Карнеги — Меллона

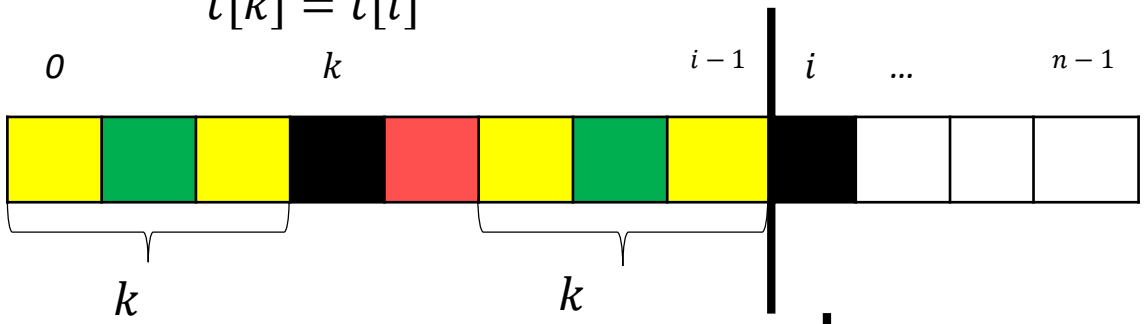
Предположим, что префиксная функция P вычислена для позиций $1, \ldots, i-1$ строки T.
Вычислим её значение для позиции i строки T.

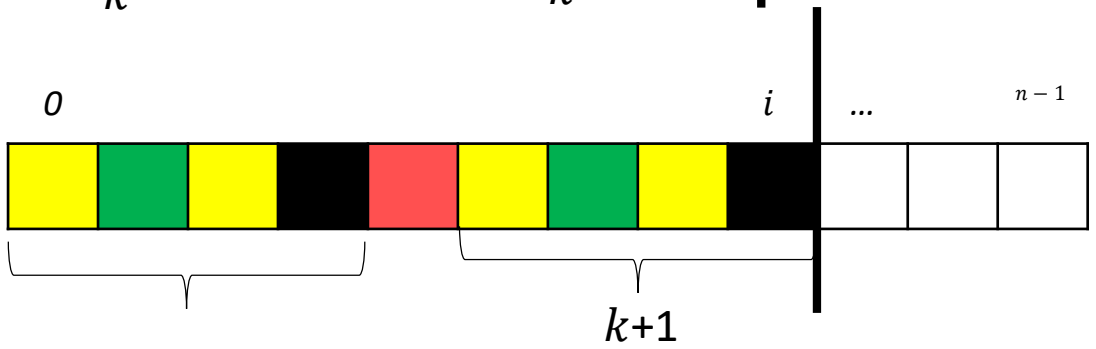


Случай 1.

$k = p[i-1]$

$t[k] = t[i]$





$$p[i] = p[i-1] + 1$$

Покажем, что большее значение для позиции i строки T получить нельзя.

Предположим, что можно получить большее значение префиксной функции в позиции i т.е.

$$p[i] > p[i-1] + 1.$$

Удалим символ i из подстроки $T[0..i]$, получим, что у подстроки $T[0..i-1]$ есть суффикс равный префиксу и его длина $p[i]-1$. По сделанному предположению

$$p[i] - 1 > p[i-1],$$

т.е. найдено лучшее значение префиксной функции для позиции $i-1$ строки T – противоречие.

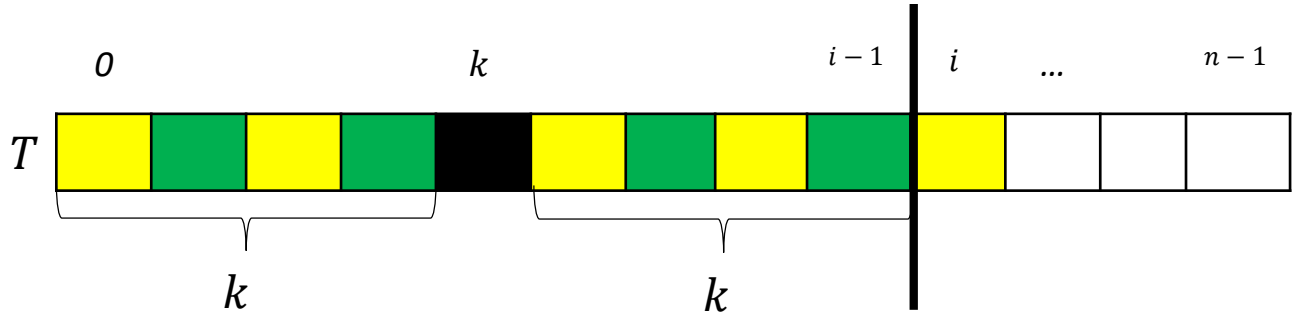
Утверждение 1.

Для каждой позиции i строки T значение префиксной функции может увеличиться максимум на 1 по сравнению с предыдущим значением.

Случай 2.

$k = p[i-1]$

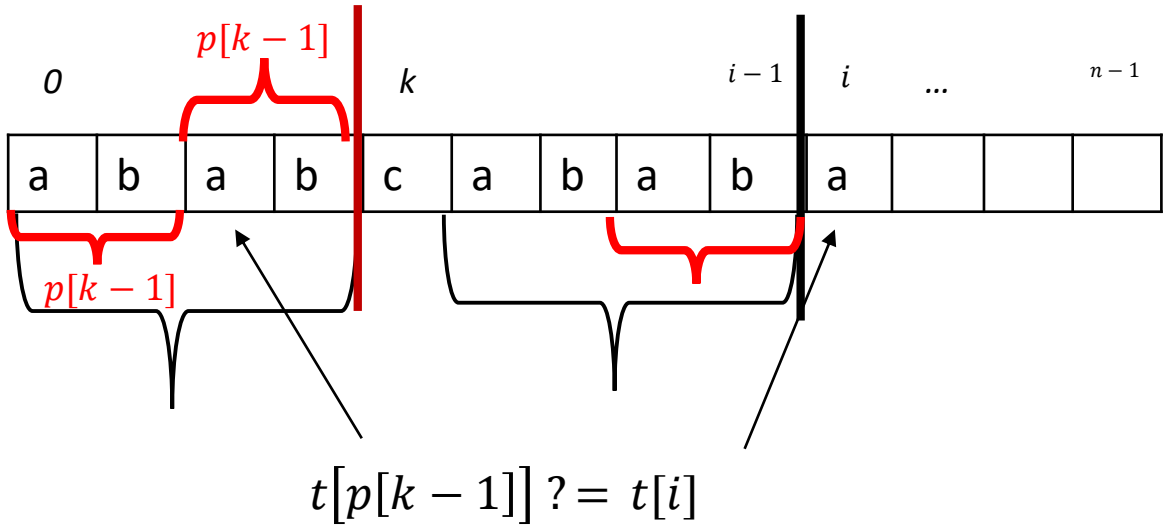
$t[k] \neq t[i]$



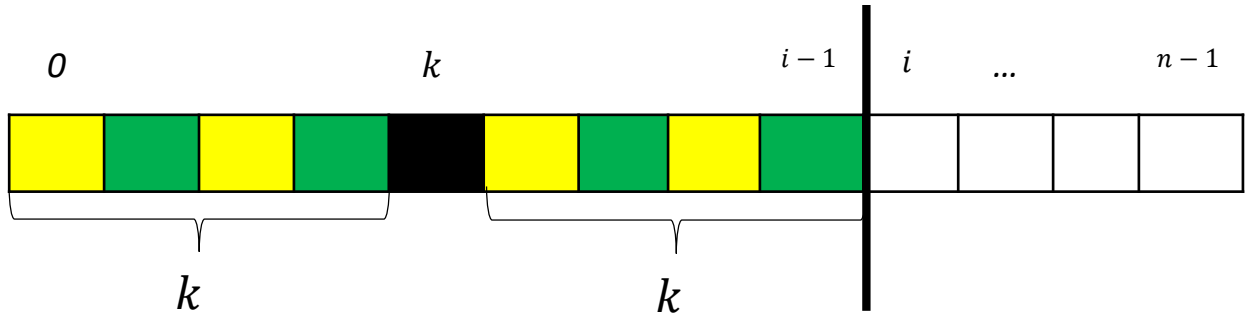
Если $p[i-1] = 0$, то $p[i] = 0$.

Рассмотрим случай, когда $p[i-1] > 0$:

значит для подстроки $\mathrm{T}[0..i-1]$ надо искать наибольший по длине префикс, который равен её суффиксу, но более короткий, чем k, а затем пробовать его продолжить



$t\big[p[k-1]\big] \,?=\, t[i]$



$p[0] := 0$

для всех $i = 1, .., n-1$

$k := p[i-1]$

пока $(k > 0)$ **и** $(t[k] \neq t[i])$

$k := p[k-1]$

если $(t[k] = t[i])$, **то** $p[i] := k+1$, **иначе** $p[i] := k$

Утверждение 2.

При построения префиксной функции P для всех позиций строки $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1})$ суммарное число уменьшений значения префиксной функции P не превосходит суммарного числа её увеличений и равно $\mathrm{O}(n)$.

Обозначим через $t_i \geq 0$ число, на которое уменьшилось значение префиксной функции для позиции i по сравнению с её значением для позиции $i-1$.

На основании утверждения 1 значение префиксной функции для позиции i по сравнению с её значением для позиции $i-1$ могло увеличиться максимум на 1. Поэтому справедливы следующие неравенства.

$$p[0] = 0$$
$$0 \leq p[1] \leq 0 + 1 - t_1$$
$$0 \leq p[2] \leq (1 - t_1) + 1 - t_2 = 2 - (t_1 + t_2)$$
$$\ldots$$
$$0 \leq p[n-1] \leq (n-1) - (t_1 + t_2 + \cdots + t_{n-1})$$

$$\sum_{i=1}^{n-1} t_i \leq n - 1$$

Утверждение 3.

Время работы алгоритма построения префиксной функции P для всех позиций строки $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1})$ есть $\mathrm{O}(n)$.

Алгоритм



Кнута-Морриса-Пратта

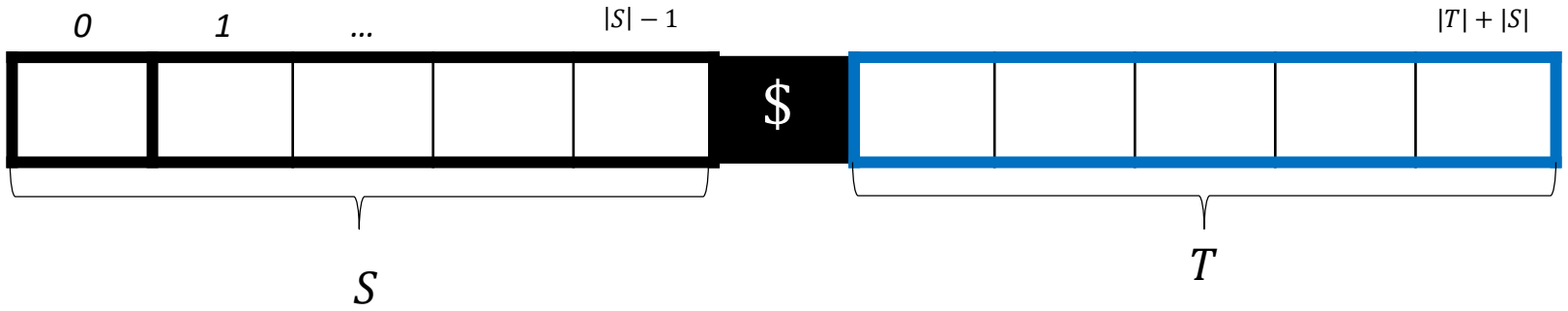
(*сокр*. **КМП)**

поиска подстроки в строке

Задан текст T и образец S.
Требуется найти все вхождения образца S в текст T.

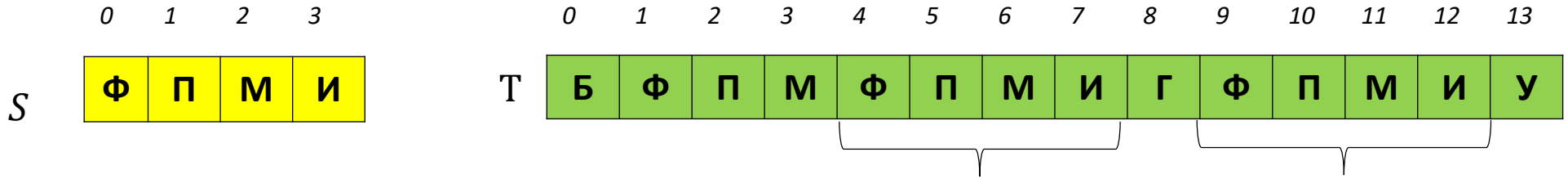
Эффективный алгоритм **КМП** основан на построении префикс-функции P и работает за время $\mathbf{O}(|\boldsymbol{S}| + |\boldsymbol{T}|)$.

1) Выполняем конкатенацию образца $\boldsymbol{S}$ и текста $\boldsymbol{T}$, добавляя между ними символ, которого нет в обеих строках (в нашем примере это $).



2) Вычисляем префиксную функцию $\boldsymbol{P}$ для всех позиций строки $\mathbf{S} + \$ + \mathbf{T}$.

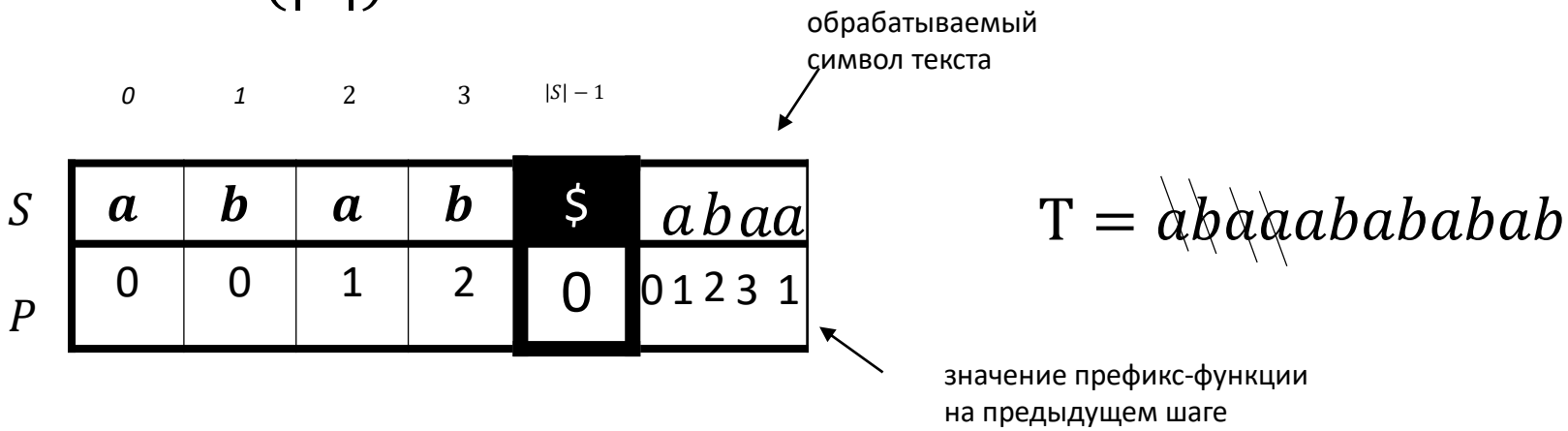
3) Образец встречается в тексте столько раз, сколько раз префиксная функция принимала значение, равное длине образца.



Если $\mathrm{p}[i] = |S|$, то

образец S встречается в тексте T, начиная с индекса $j = i - 2 \cdot |S|$.

Память - $O(|S|)$.



$$T = abaaababababab$$

Так как значение префикс-функции не превосходит $|S|$, то достаточно хранить только строку $S + \$$, значение префиксной функции для строки $S + \$$, последнее вычисленное значение префиксной функции и, считывая по одному очередной символ текста T, и вычислять префикс-функцию для этого символа.

Примеры задач, для которых можно разработать эффективный алгоритм, используя алгоритм построения префиксной функции.

1. Задана строка$\boldsymbol{T}$, сколько различных подстрок в этой строке. Время работы алгоритма $\mathbf{O}(|\boldsymbol{T}|^{\mathbf{2}})$.
2. Задана строка $\boldsymbol{T}$. Найти такую строку $\boldsymbol{S}$ наименьшей длины, что $\boldsymbol{T}$ можно представить в виде конкатенации строк $\boldsymbol{S}$ $(\text{например, для T} = {}'\text{aaaaaaaa}'\ \text{строка S} = {}'\text{a}')$ (т.е. хотим найти самое короткое сжатое представление строки). Время работы алгоритма $\mathbf{O}(|\boldsymbol{T}|)$.
3. Циклическим сдвигом строки $\boldsymbol{T}$ на k символов будем называть строку, которая получается из исходной строки переносом первых k символов в конец строки. Необходимо определить, можно ли получить из одной строки $(\boldsymbol{T_1})$ другую $(\boldsymbol{T_2})$ при помощи некоторого циклического сдвига. Время работы алгоритма $\mathbf{O}(|\boldsymbol{T_1}| +$

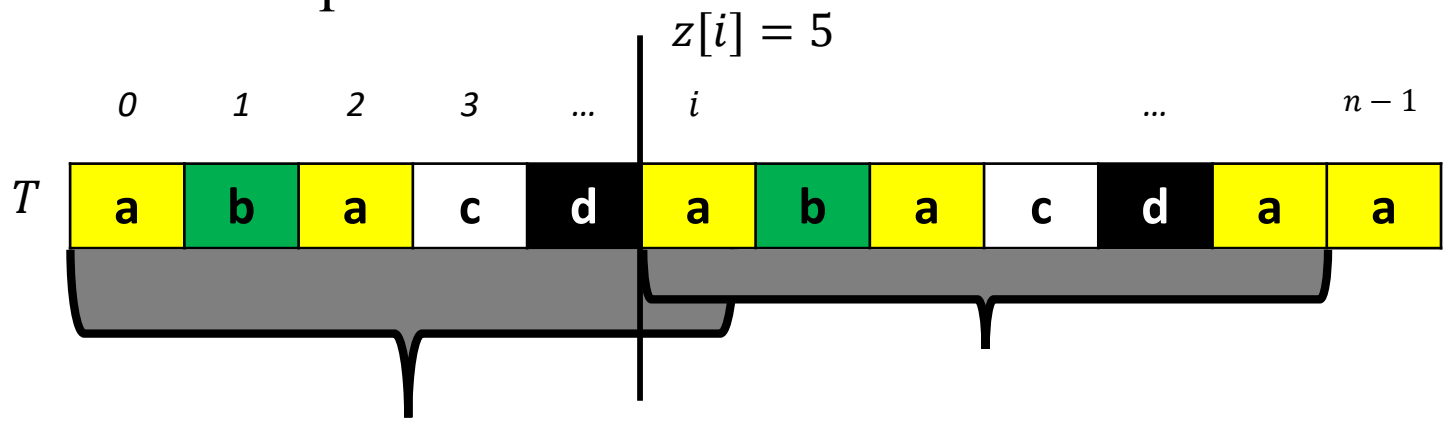
Z-функция строки T

(англ. *Z-function*)

z-функция (“зет-функция”) от строки $\mathrm{T} = (t_0, t_1, \ldots, t_{n-1})$, также, как и префиксная функция, используется в алгоритмах обработки строк

Z-функция
от строки
$T = (t_0, t_1, \dots, t_{n-1})$

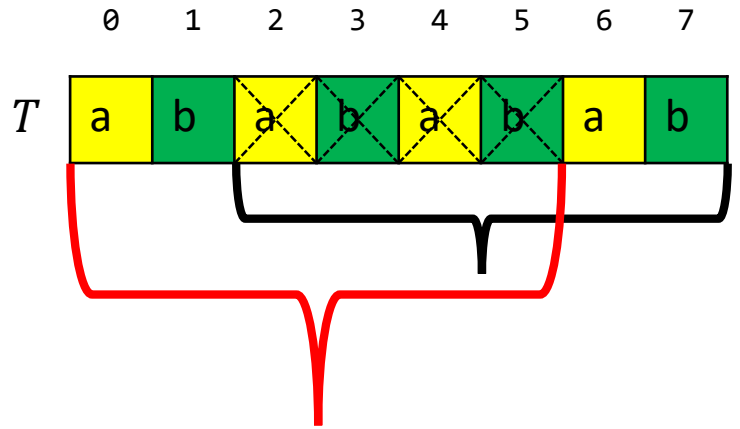
это массив целых чисел $\boldsymbol{Z[0..n-1]}$,
где $z[i]$ – длина наибольшего общего префикса i-ого суффикса строки $\boldsymbol{T}$ и самой строки $\boldsymbol{T}$



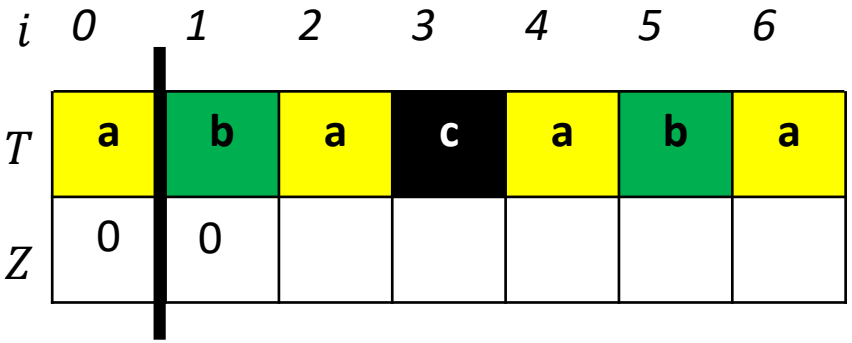
$$\begin{cases} z[0] = 0 \\ z[i] = \max\{k \mid T[i..i+k-1] = T[0..k-1]\}, \mathrm{i} = 1,..,n-1 \end{cases}$$

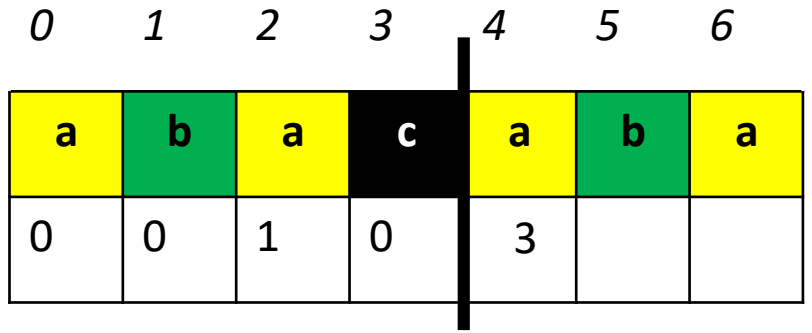
другими словами, $z[i]$ – наибольшее число k, такое, что первые k символов подстроки $T[i..n-1]$ совпадают с первыми k символами строки T;

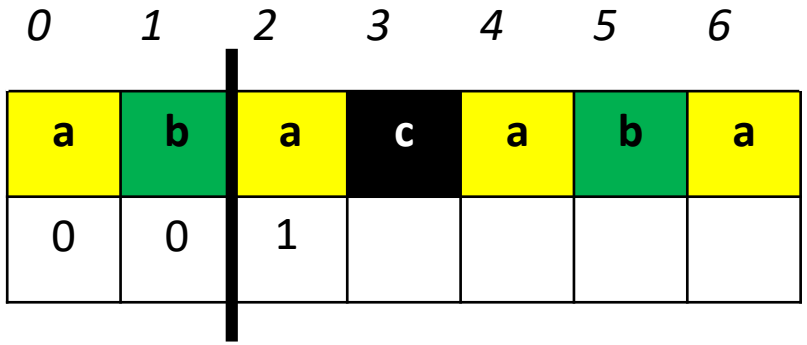
наибольший общий префикс строки T и подстроки $T[i..n-1]$ могут пересекаться

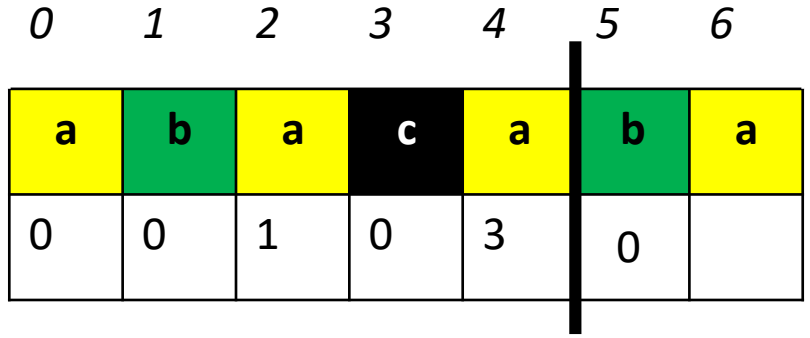


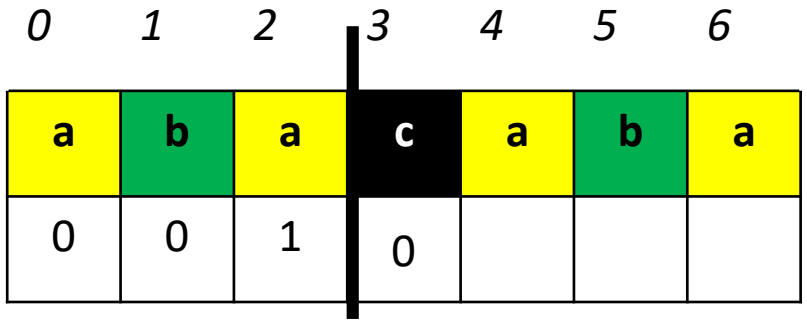
$z[2] = 6$

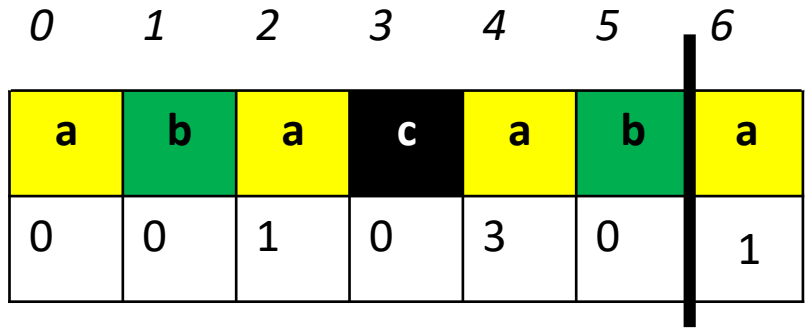








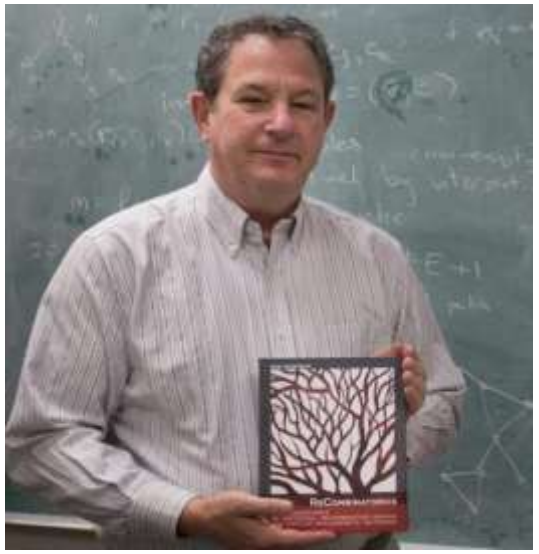




Наивный алгоритм (в основе которого лежит сравнение подстрок) работает за время $\Theta(|T|^2) = \Theta(n^2)$.

Эффективный алгоритм построения z-функции ($O(|T|)= O(n)$)
был изложен в **1997** году **Дэном Гасфилдом** в его книге

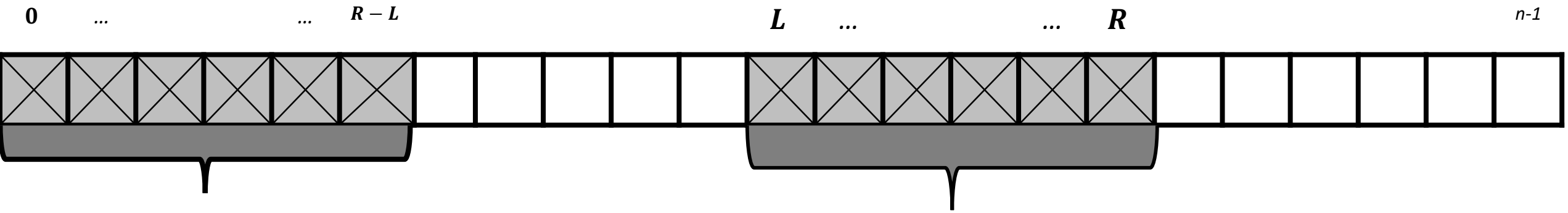
«Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология»



Professor Daniel Gusfield
Computer Science
University of California

Пусть на некоторой итерации алгоритма L и R – индексы левой и правой границы самого «правого» отрезка (блока) совпадения (среди двух блоков с одинаковой правой границей будем брать больший по числу элементов).

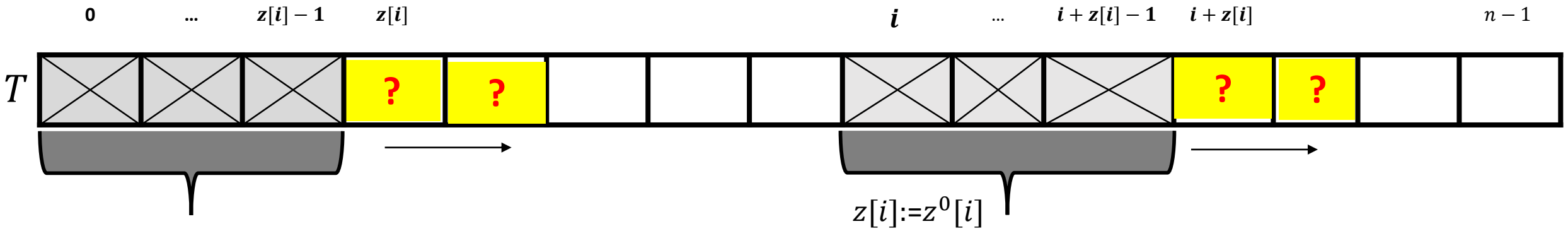
Первоначально полагаем L и R равными **0**.



i	0	1	2	3	4	5	6	7
T	**a**	**b**	**a**	**b**	**a**	**b**	**c**	**a**
Z	0	0	4	0	2	0	0	1
L, R	**0,0**	**0,0**	**2,5**	**2,5**	**2,5**	**2,5**	**2,5**	**7, 7**

i	0	1	2	3
T	**a**	**a**	**a**	**a**
Z	0	3	2	1
L, R	**0,0**	**1,3**	**1,3**	**1,3**

Будем формировать массив $\boldsymbol{Z}$ последовательно от $\boldsymbol{0}$ до $\boldsymbol{n-1}$.
Предположим, что вычислены элементы $\boldsymbol{z[j], j < i}$.
Вычислим $\boldsymbol{z[i]}$.
Сначала попытаемся проинициализировать $\boldsymbol{z[i]}$ некоторым ненулевым значением, используя ранее вычисленные значения: $\boldsymbol{z[i] := z^0[i]}$.



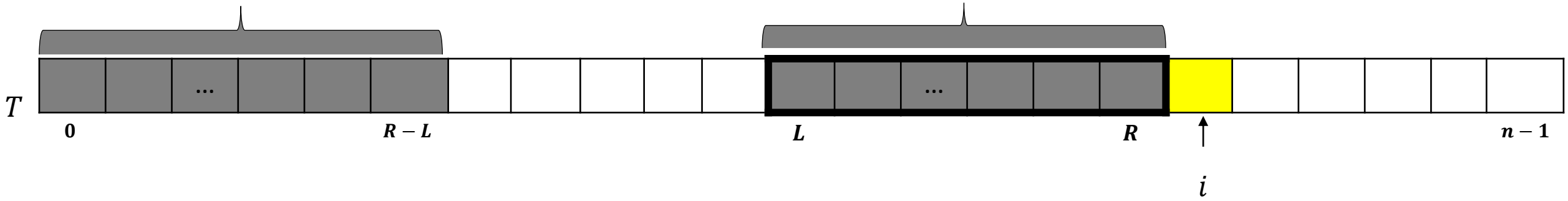
Затем, наивным алгоритмом пробуем увеличить значение $\boldsymbol{z[i]}$, пока идёт совпадение:

$t[z[i]] \quad ?= \quad t[i + z[i]]$

$t\,[z[i] + 1]\,?= \quad t[i + z[i] + 1]$

...

Случай 1: $i > R$ (текущий индекс i вышел за пределы блока).



1) Проинициализируем $z[i] = z^0[i] = 0$.

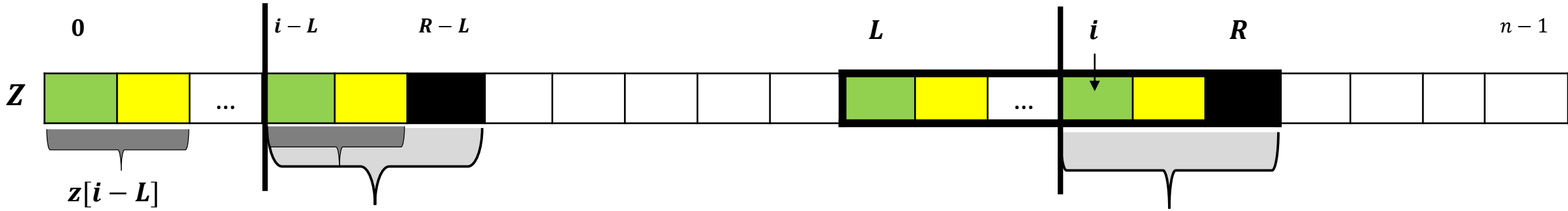
2) Наивным алгоритмом, пока совпадение, сравниваем $t[0] ? = t[i]$, $t[1] ? = t[i+1]$ ... ,
увеличивая $z[i]$ на 1 при каждом совпадении (как только символы не совпадают, останавливаемся).

3) Если $z[i] > 0$, то корректируем границы самого правого блока совпадения:

$$L = i,\ R = i + z[i] - 1.$$

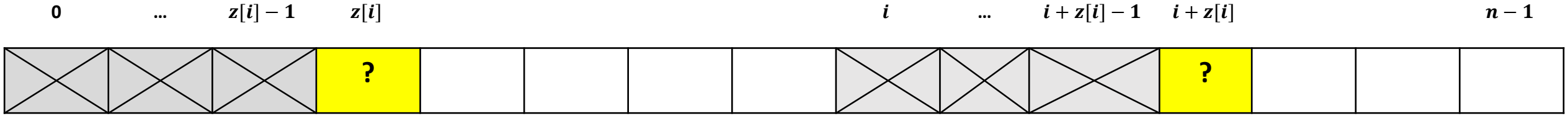
✓ отметим, что **при каждом совпадении символов**, величина R (правая границы самого правого блока совпадения) **будет увеличиваться на 1**

Случай 2: $i \le R$ (текущий индекс i находится в блоке).



1) Проинициализируем $z[i] = z^0[i] = \min\{z[i-L], R-i+1\}$.

2) Наивным алгоритмом, **пока совпадение**, сравниваем : $t[z[i]]? = t[i+z[i]], t[z[i]+1]? = t[i+z[i]+1]$ … увеличивая каждый раз $z[i]$ на 1 при совпадении.



✓ если при инициализации $z^0[i] < R-i+1$, то это начальное значение будет итоговым для $z[i]$;

✓ если $z^0[i] = R-i+1$, то правая граница при каждом последующем успешном сравнении соответствующих символов будет увеличиваться на 1 и число увеличений равно количеству выполненных операций сравнения символов;

для всех $i = 1, \ldots, n-1$

$z[i] := 0$

$L := 0, R := 0$

для всех $i = 1, .., n-1$

если $i \leq R$

$z[i] = \min(R - i + 1, z[i - L])$

пока $(i + z[i] < n)$ и $(t[z[i]] = t[i + z[i]])$

$z[i] := z[i] + 1$

если $(i + z[i] - 1 > R)$

$L := i$

$R := i + z[i] - 1$

При рассмотрении случаев 1 и 2 было показано, что каждый раз, когда происходит **сравнение символов и они оказываются равны** (внутренний цикл пока), значение правой границы $\boldsymbol{R}$ **увеличивается на 1**.
Значение $\boldsymbol{R}$ в ходе работы алгоритма не может уменьшиться.

Наибольшее значение, которое может достигнуть $\boldsymbol{R}$, равно длине строки $\boldsymbol{T}$. Следовательно, суммарное число итераций внутреннего цикла $\mathbf{O}(|\boldsymbol{T}|)$.

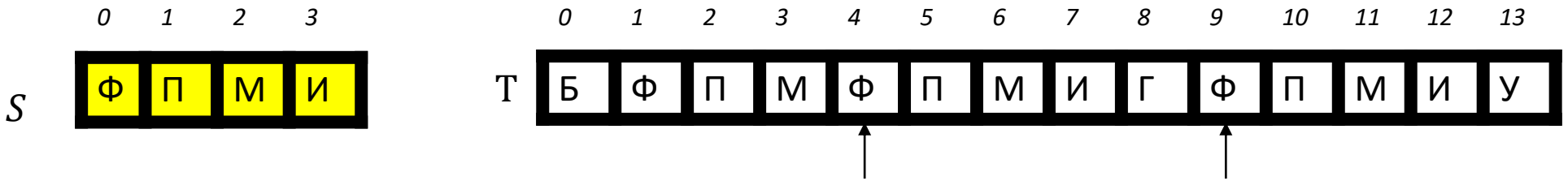
Все остальные операции алгоритма выполняются суммарно за время $\boldsymbol{O}(|\boldsymbol{T}|)$.

для всех $i = 1, \dots, n-1$
  $z[i] := 0$
$L := 0, R := 0$
для всех $i = 1, .., n-1$
  если $i \le R$
    $z[i] = \min(R - i + 1, z[i - L])$
  пока $(i + z[i] < n)$ и $(t[z[i]] = t[i + z[i]])$
    $z[i] := z[i] + 1$
  если $(i + z[i] - 1 > R)$
    $L := i$
    $R := i + z[i] - 1$

Следовательно время работы алгоритма построения z-функции

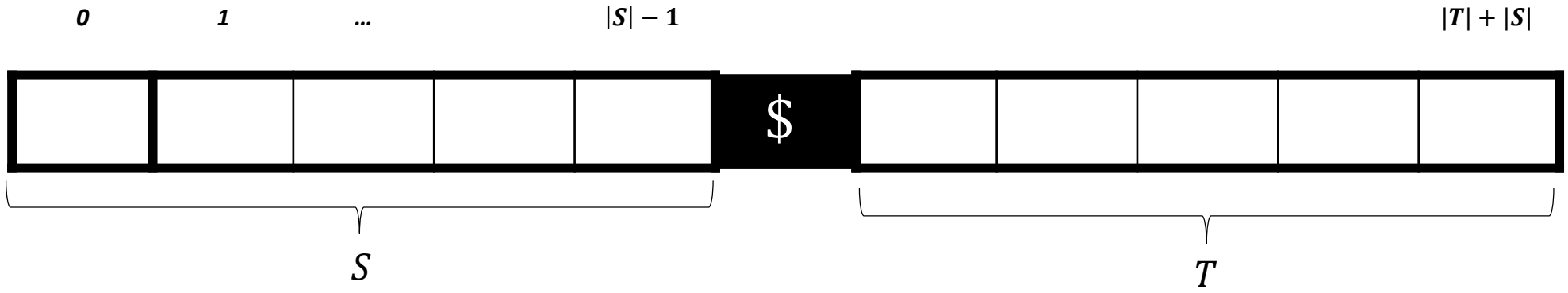
$$\mathbf{O}(|\boldsymbol{T}|) = \mathbf{O}(\boldsymbol{n}).$$

Задан текст $\boldsymbol{T}$ и образец $\boldsymbol{S}$. Найти все вхождения образца $\boldsymbol{S}$ в текст $\boldsymbol{T}$.



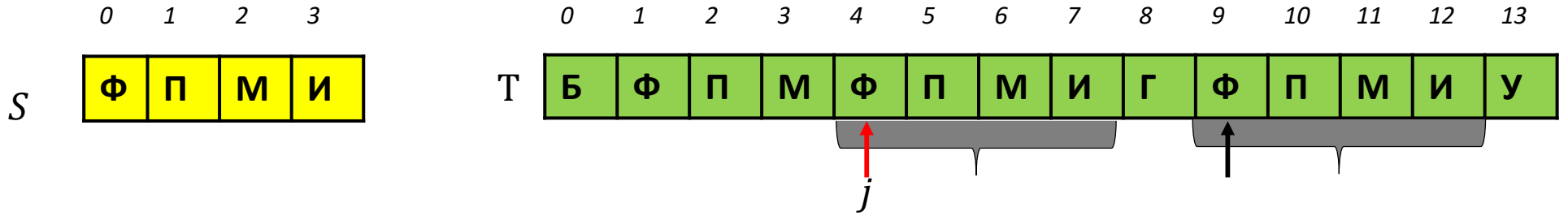
Эффективный алгоритм, основанный на построении z-функции, работает за время $O(|S| + |T|)$.

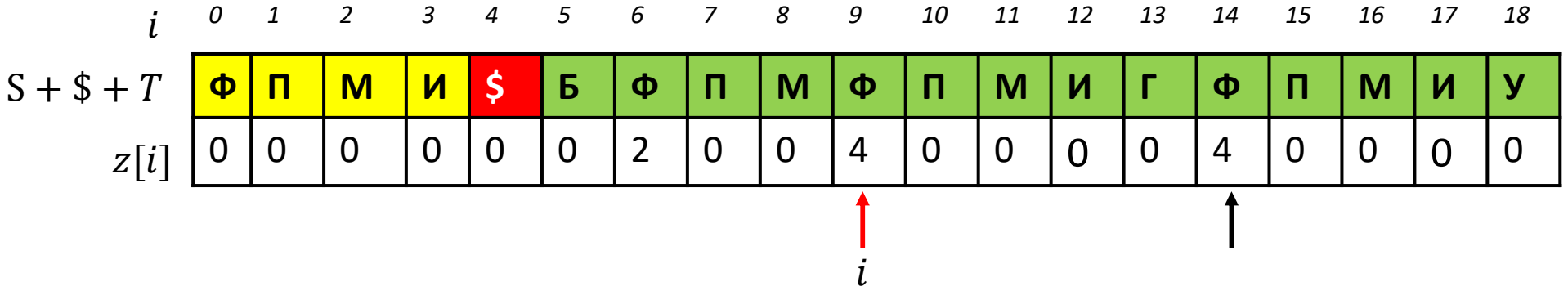
1) Выполняем конкатенацию образца $\boldsymbol{S}$ и текста $\boldsymbol{T}$, добавляя между ними символ, которого нет в обеих строках (в нашем примере это \$).



2) Вычисляем **z**-функцию для $\boldsymbol{S} + \$ + \boldsymbol{T}$.

3) Образец $\boldsymbol{S}$ встречается в тексте $\boldsymbol{T}$ столько раз, сколько раз z-функция принимала значение, равное длине образца.
4) Если $\boldsymbol{z[i] = |S|}$, то в тексте $\boldsymbol{T}$, начиная с позиции $\boldsymbol{j = i - |S| - 1}$, найдено вхождение образца $\boldsymbol{S}$.





если $\boldsymbol{z[i] = |S|}$, то

образец $\boldsymbol{S}$ встречается в тексте $\boldsymbol{T}$, начиная с индекса $\boldsymbol{j = i - |S| - 1}$

Задача поиска всех подстрок, которые являются палиндромами

Техника поддержки самого правого блока совпадения, которая использовалась для построения эффективного алгоритма построения z–функции, также используется в задаче **поиска всех подстрок заданной строки T, которые являются палиндромами.**

Эффективный алгоритм решения этой задачи, работающий за время $\mathbf{O}(|T|)$, был предложен **Гленном Манакером** (Glenn Manacher) в **1975** г.

Glenn Manacher

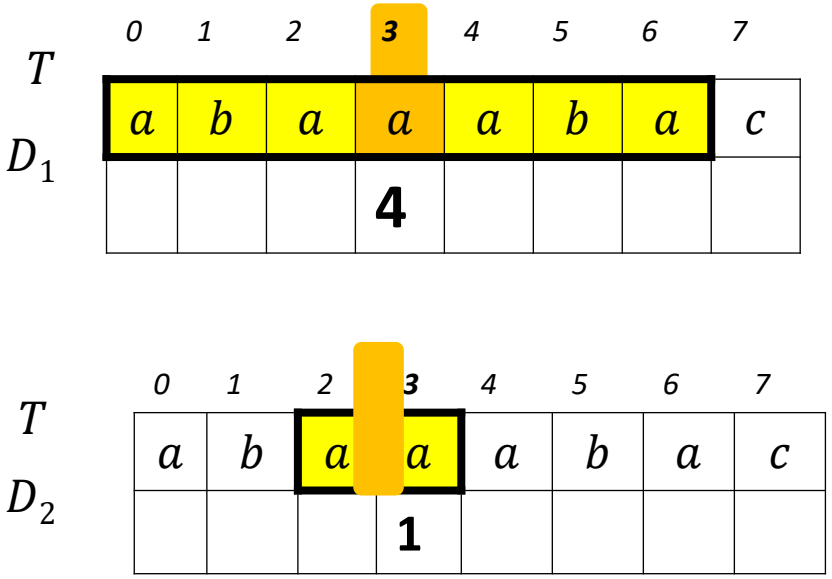
Assoc. Professor Emeritus
Mathematics, Statistics, and Computer Science
Department of Information Engineering,
University of Illinois at Chicago

Простейший же тривиальный алгоритм, который проверяет все возможные подстроки работает за время $\mathbf{O}(|T|^2)$.

Для решения задачи нужно сформировать два массива D_1 и D_2, где

$\boldsymbol{d_1[i]}$ – число палиндромов нечётной длины с центром в i;

$\boldsymbol{d_2[i]}$ – число палиндромов чётной длины с центром в i (мнимый центр в i, а подстрока, начиная с индекса i, сдвинута на 1 позицию вправо);



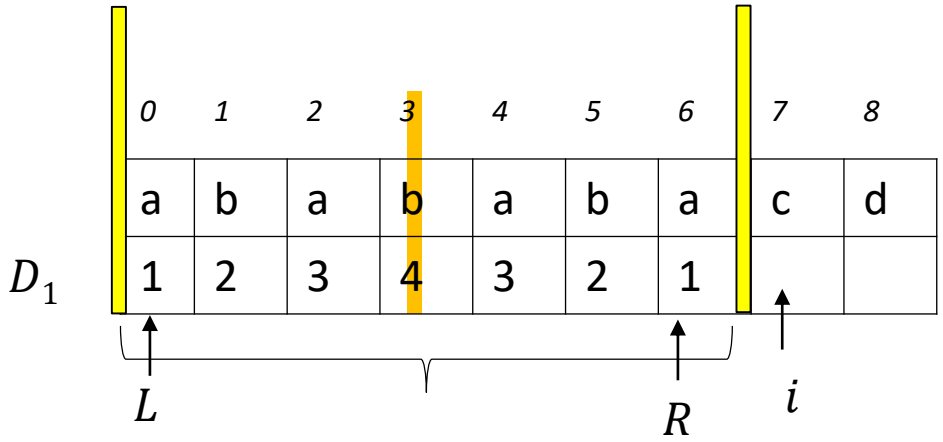
$aba\boldsymbol{a}aba$
$ba\boldsymbol{a}ab$
$a\boldsymbol{a}a$
$\boldsymbol{a}$

Если для каждой позиции i подсчитать **длину наибольшего палиндрома** чётной и нечётной длины, то легко вычислить значения соответствующих массивов $d_1[i]$ и $d_2[i]$, так как, если строка T является палиндромом, то отбрасывание первого и последнего символов этой строки T также приводит к палиндрому с тем же центром.

Будем поддерживать **левую L и правую R границу самого правого из найденных палиндромов.**

Предположим, что вычислены элементы массива $d_1[j], j < i$ и нужно вычислить $d_1[i]$, используя ранее вычисленные значения (первоначально $\boldsymbol{d_1[j] = 1}$ **для всех** j) :

Случай 1 (текущий индекс i вышел за пределы блока) $\boldsymbol{i > R}$



Пока не вышли за пределы строки и есть совпадения, применяем наивный алгоритм, сравнивая

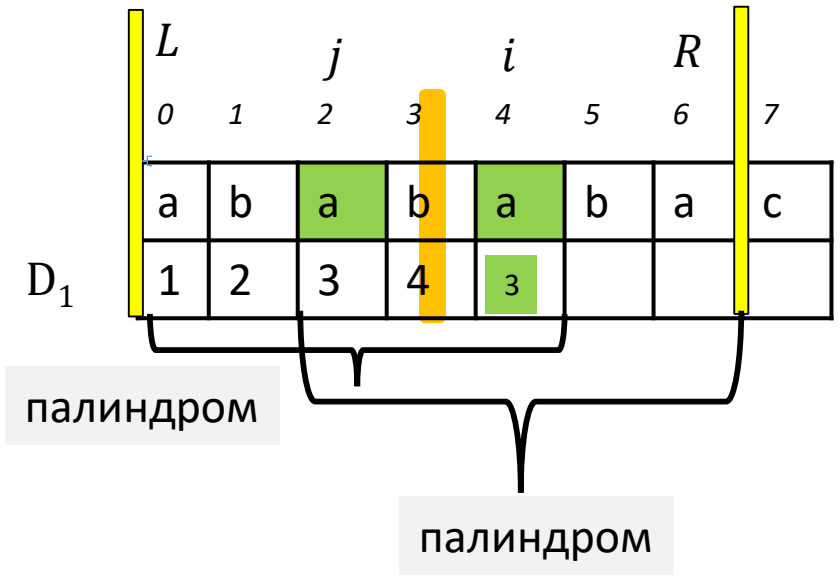
$$t[i + d_1[i]\,]? = t[i - d_1[i]\,],$$

увеличиваем значение $d_1[i]$ в случае совпадения.

Корректируем границы самого правого блока.

Случай 2 (текущий индекс i находится в блоке)

$\boldsymbol{i \le R}$, в этом случае можно проинициализировать $d_1[i]$ через ранее вычисленные значения



при формировании массива d_1
для i симметричный индекс
$j = (R - i) + L$

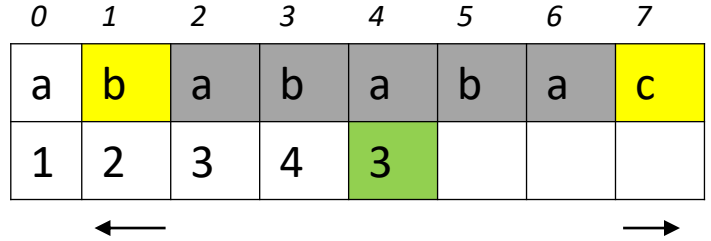
при формировании массива d_2
для i симметричный индекс
$j = (R - i) + L + 1)$

Начальная инициализация $d_1[i] := \min\{d_1[j], R - i + 1\}$, где $j = (R - i) + L$.

Далее, пока не вышли за пределы строки и есть совпадения, применяем наивный алгоритм, сравнивая

$$t[i + d_1[i]\,] \,?= t[i - d_1[i]\,]$$

и увеличивая $d_1[i]$ в случае совпадения.



При необходимости корректируем границы самого правого блока.

D_1

```
L := 0, R := −1
для всех i = 0, .., n − 1
    если i > R
        k := 1
    иначе l := min(d1[l + r − i], r − i + 1)
    пока (i + k < n) и (i − k ≥ 0) и (t[i + k] = t[i − k])
        k := k + 1
    d1[i] := k
    если (i + k − 1 > R)
        L := i − k + 1
        R := i + k − 1
```

D_2

```
L := 0, R := −1
для всех i = 0, .., n − 1
    если i > R
        k := 1
    иначе l := min(d2[l + r − i + 1], r − i + 1)
    пока (i + k < n) и (i − k − 1 ≥ 0) и (t[i + k] = t[i − k − 1])
        k := k + 1
    d2[i] := k
    если (i + k − 1 > R)
        L := i − k
        R := i + k − 1
```

При рассмотрении случаев 1 и 2 было показано, что каждый раз, когда происходит сравнение символов и они оказываются равны, значение правой границы R увеличивается на 1.

Значение R в ходе работы алгоритма не может уменьшиться.

Наибольшее значение, которое может достигнуть R, равно длине строки T.

Следовательно, суммарное число итераций внутреннего цикла **пока** – $O(|T|)$.

Все остальные операции алгоритма выполняться суммарно за время $O(|T|)$.

Следовательно, время работы алгоритма поиска всех подстрок заданной строки T, которые являются палиндромами, есть $O(|T|)$.

Спасибо за внимание!