

(продолжение)

Строковые алгоритмы и структуры данных



БЕЛОРУССКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

Бор

Суффиксный бор

Суффиксный массив

Задача поиска заданной строки в множестве строке

Предположим, что есть множество строк

$$S = \{S_0, S_1, \dots, S_{n-1}\}$$

из букв латинского алфавита Σ и необходимо быстро проверить, есть ли среди них некоторая заданная строка

$$A = (a_0, a_2, \dots, a_{l-1}).$$

Если поместить все строки из множества $S = \{S_0, S_1, \dots, S_{n-1}\}$ в **массив** и осуществлять поиск строки $A = (a_0, a_2, \dots, a_{l-1})$ последовательно просматривая элементы массива, то **время поиска строки A** —

$$O(|A| \cdot n),$$

где n — количество строк в S .

При этом требуемая **память** — $O(|S_0| + |S_1| + \dots + |S_{n-1}|)$.

Если для хранения строк из множества $S = \{S_0, S_1, \dots, S_{n-1}\}$ использовать сбалансированное бинарное поисковое дерево, то **время поиска строки A** —

$$O(|A| \cdot \log n),$$

где n — количество строк в S .

Существуют структуры данных, которые позволяют выполнять поиск строки более эффективно, например, за время

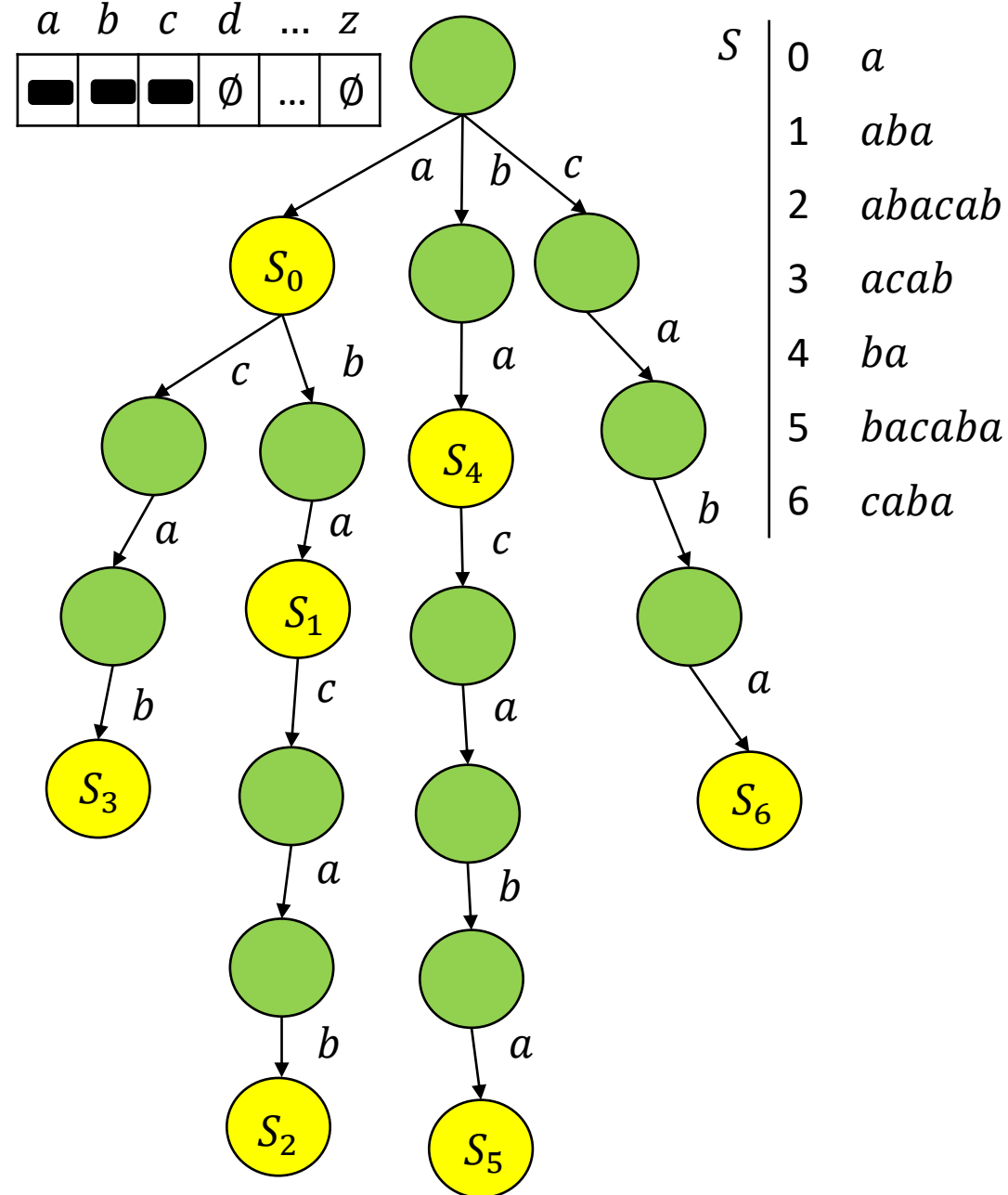
$$O(|A|)$$

(время поиска не зависит от количества строк в S).

Одной из таких структур данных является **бор**.

Бор

(англ. trie, луч, нагруженное дерево)



$A = aba$

Разобьем слова из множества $S = \{S_0, S_1, \dots, S_{n-1}\}$ на группы в соответствии с первыми символами слов.

Пусть S' - группа, у слов которой первый символ совпадает с первым символом строки A . Сокращаем область поиска до множества S' , исключив из рассмотрения те строки, которые туда не попадают.

Заведем массив, в качестве индексов которого будут выступать символы алфавита, а в самом массиве будут храниться сами строки.

$$S \begin{array}{l|l} 0 & a \\ 1 & aba \\ 2 & abacab \\ 3 & acab \\ 4 & ba \\ 5 & bacaba \\ 6 & caba \end{array}$$

$A = (\underline{a}, b, a)$

S' →

	a	b	c	d
	a aba $abacaba$ $acab$	ba $bacaba$	$caba$	

Удаляем из всех слов множества S' и слова A первый символ и повторяем процедуру.

$$S' \begin{array}{l|l} 1 & ba \\ 2 & bacaba \\ 3 & cab \end{array}$$

$A' = (a, \underline{b}, a)$

	a	b	c	d
		ba $bacaba$	cab	

$$S' \begin{array}{l|l} 1 & a \\ 2 & acaba \end{array}$$

$A' = (a, b, \underline{a})$

	a	b	c	d
	a $acaba$			

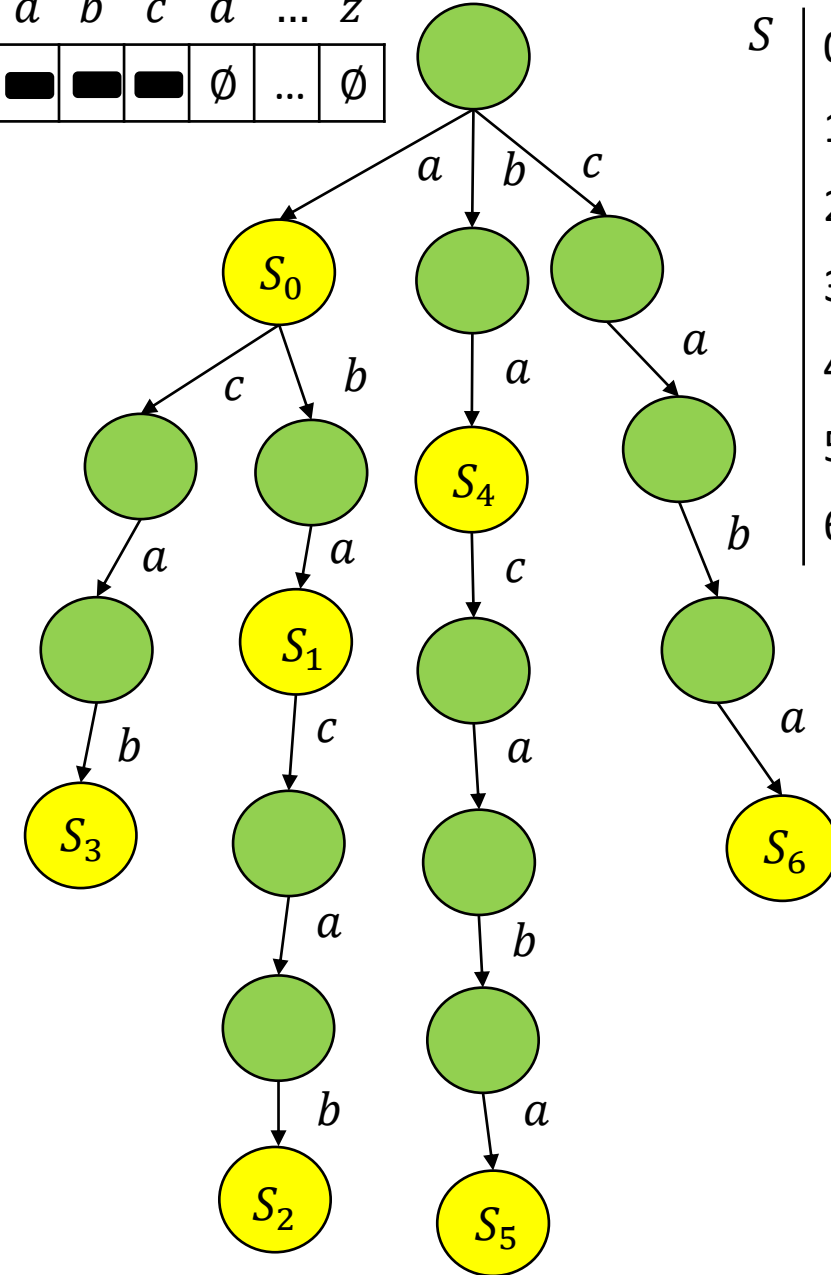
Если искомая строка A' состоит из одного символа, то достаточно проверить наличие в множестве S' строки длины 1.

Приведенное иерархическое разбиение строк на множества можно изобразить в виде древовидной структуры.

Бор

специализированная древовидная структура данных, предназначенная для хранения слов некоторого алфавита Σ .

a	b	c	d	\dots	z
■	■	■	\emptyset	\dots	\emptyset



S		
0	a	
1	aba	
2	$abacab$	
3	$acab$	
4	ba	
5	$bacaba$	
6	$caba$	

➤ вершина дерева содержит

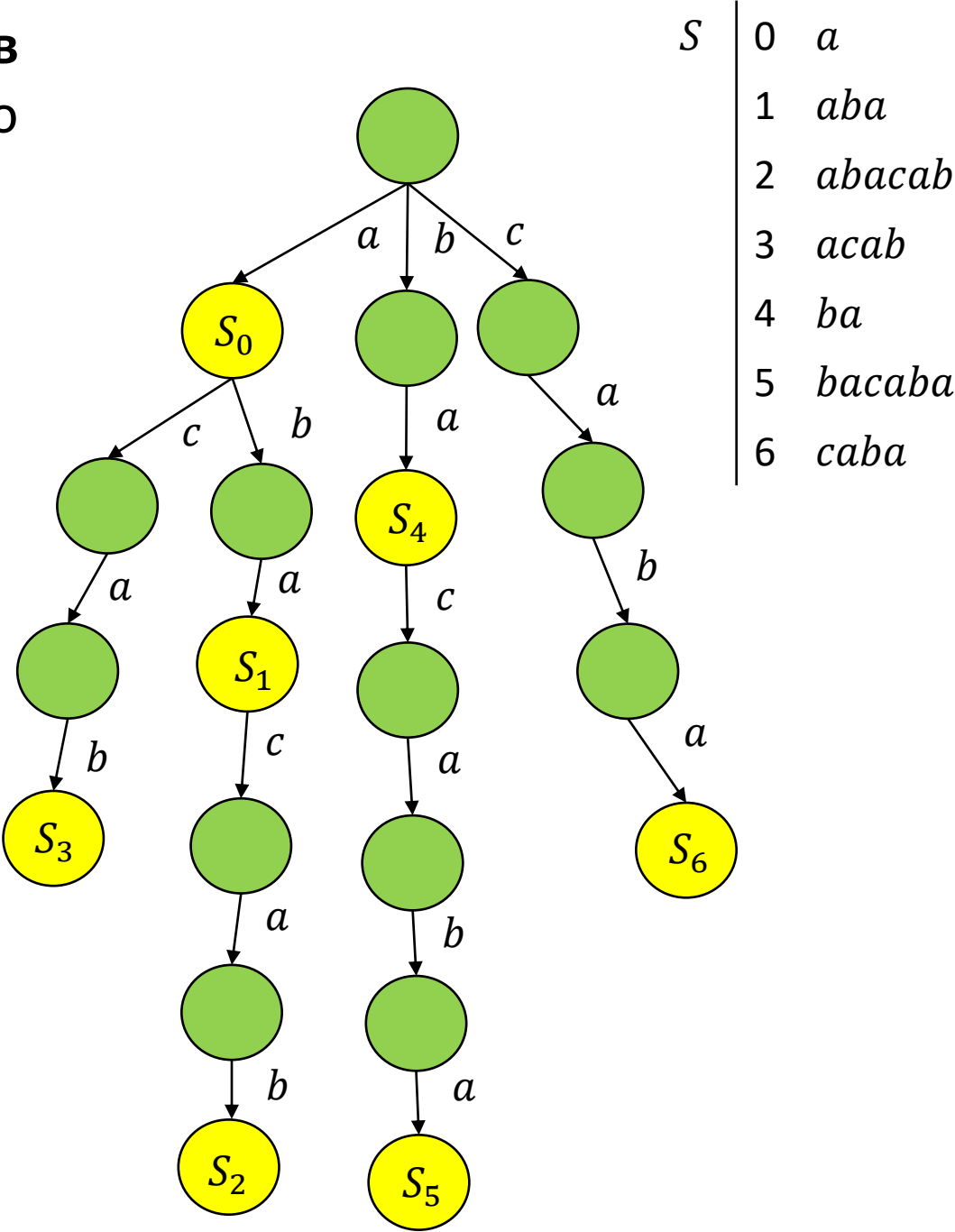
- информацию о вершинах, в которые можно перейти по каждому допустимому символу (если переход по дуге с некоторым символом не возможен, то результат перехода будем обозначать \emptyset);
- пометку: ● терминальная или ● нет (вершина v – терминальная, если строка, которая определяется последовательностью символов, встречающихся на дугах в пути от корня к v , есть в множестве S);
- терминальная вершина может хранить индекс слова в S ;

➤ дуге ставится в соответствие символ строки;

Если **выписать все символы на пути из корня в терминальную вершину**, то получим некоторую строку (слово) из множества S .

Листья дерева соответствуют строкам из S .

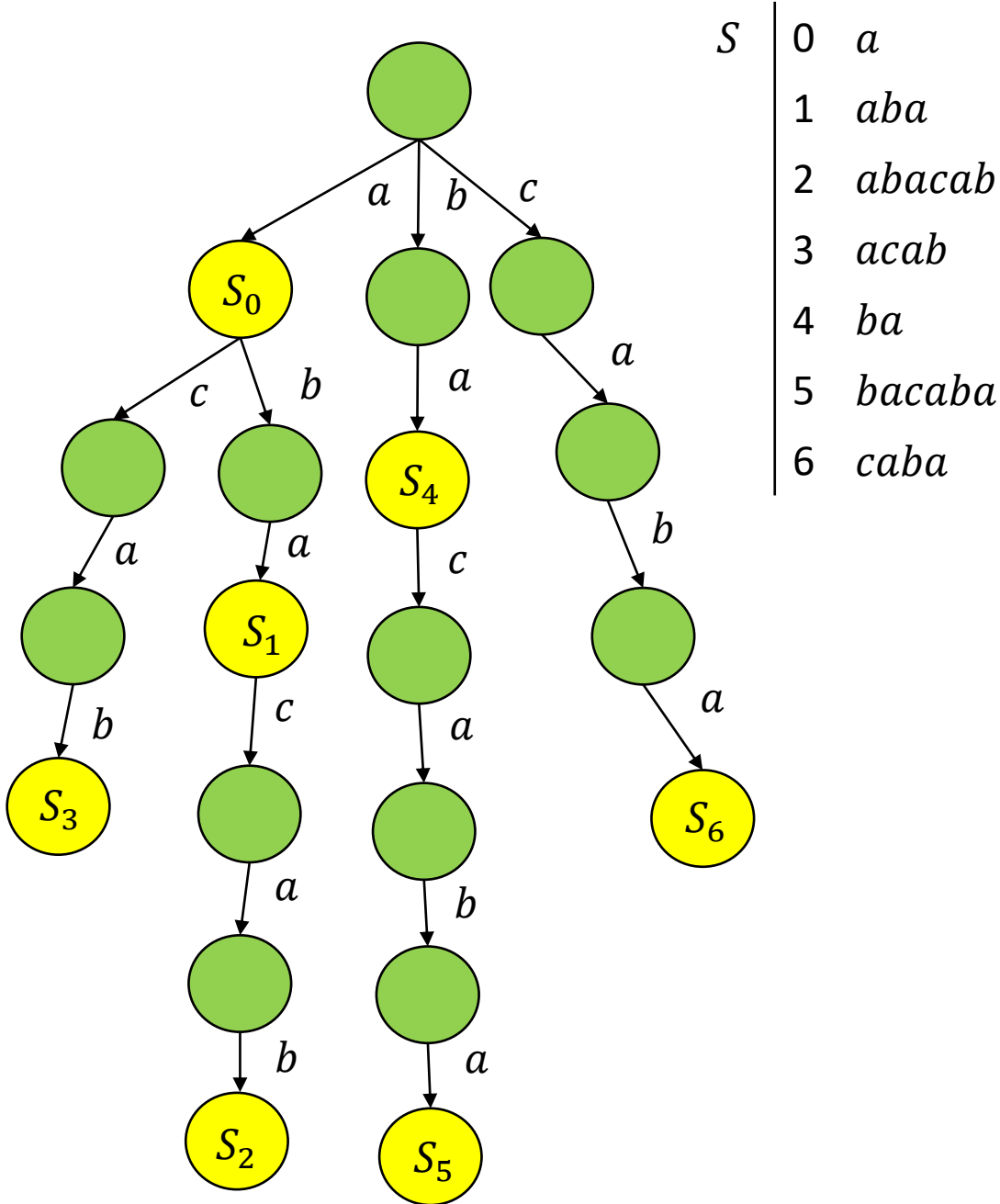
Если некоторая строка из S является префиксом другой строки этого множества, то ей будет соответствовать внутренняя вершина дерева ($S_1 = \mathbf{aba}$, $S_2 = \mathbf{abacab}$).



Для поиска строки A в боре будем последовательно спускаться из корня дерева по дугам, соответствующим символам строки A , пока строка A не закончится и в боре существует дуга, соответствующая текущему состоянию строки A .

Если в результате спуска:

- (1) попадём в терминальную вершину и дойдём до конца строки A , то **искомое слово найдено**;
- (2) если остановимся в нетерминальной вершине либо во время спуска не найдём дуги в дереве, соответствующей текущему символу строки A , то делаем вывод о том, что **строка A в множестве S отсутствует**.

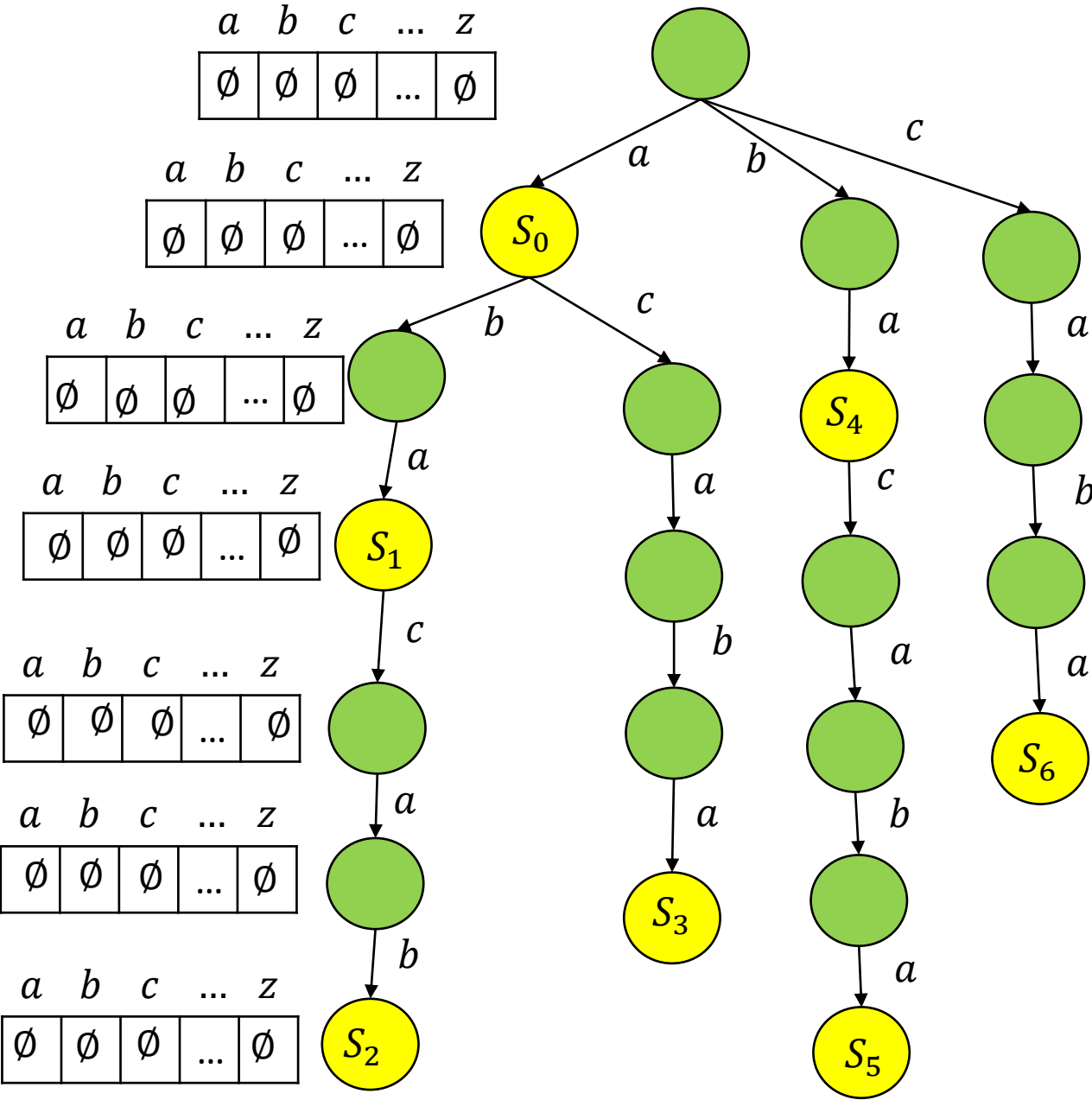


Добавление строки

Как и во время поиска, спускаемся по дереву, начиная с корня. Если во время спуска мы попадаем в ситуацию, что надо перейти по несуществующей дуге, то добавляем к дереву новую вершину и ведущую к ней дугу с соответствующей пометкой.

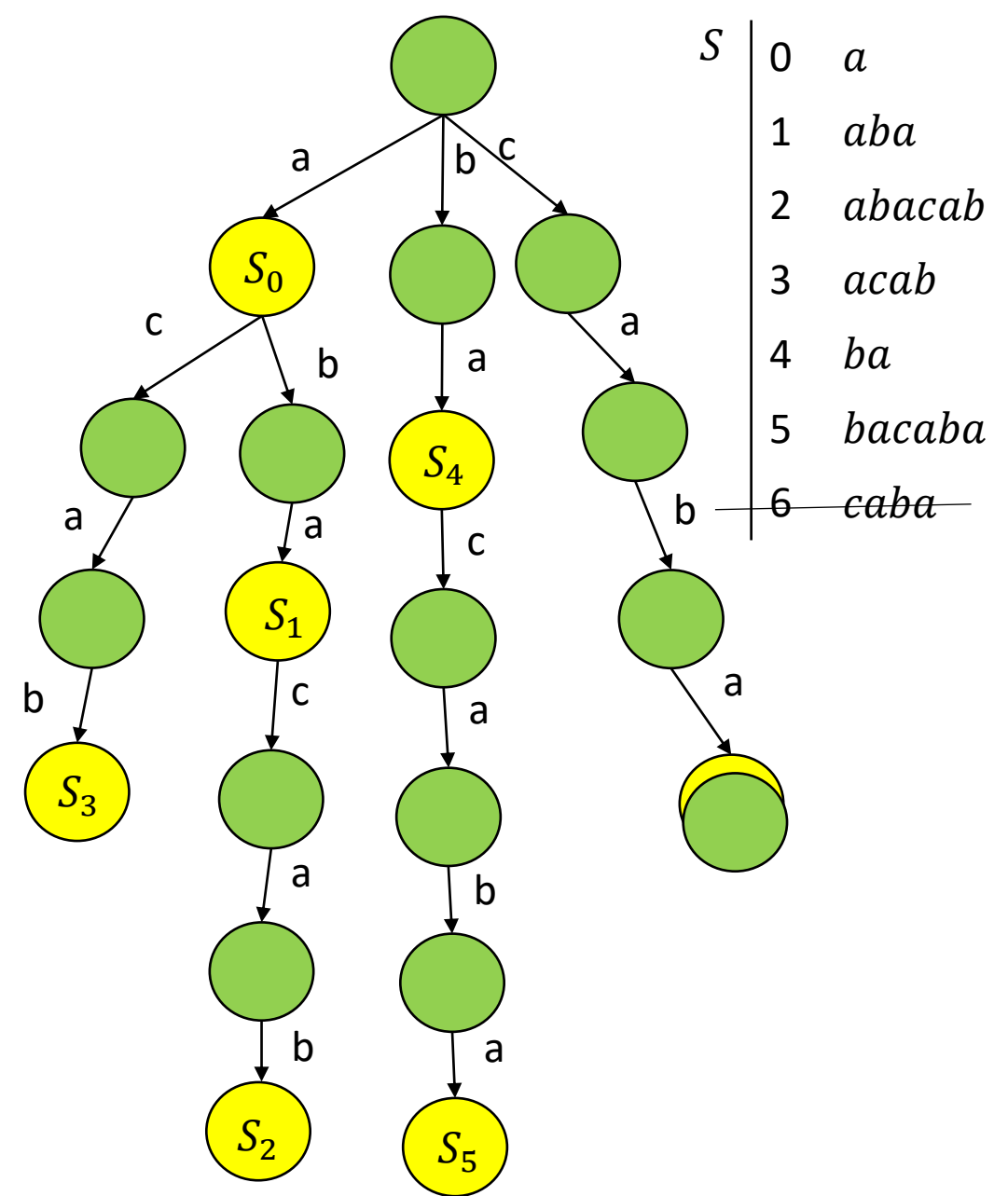
Движение осуществляем, пока не пройдем все символы добавляемой строки.

S	0	a
	1	aba
	2	abacaba
	3	acaba
	4	ba
	5	bacaba
	6	caba

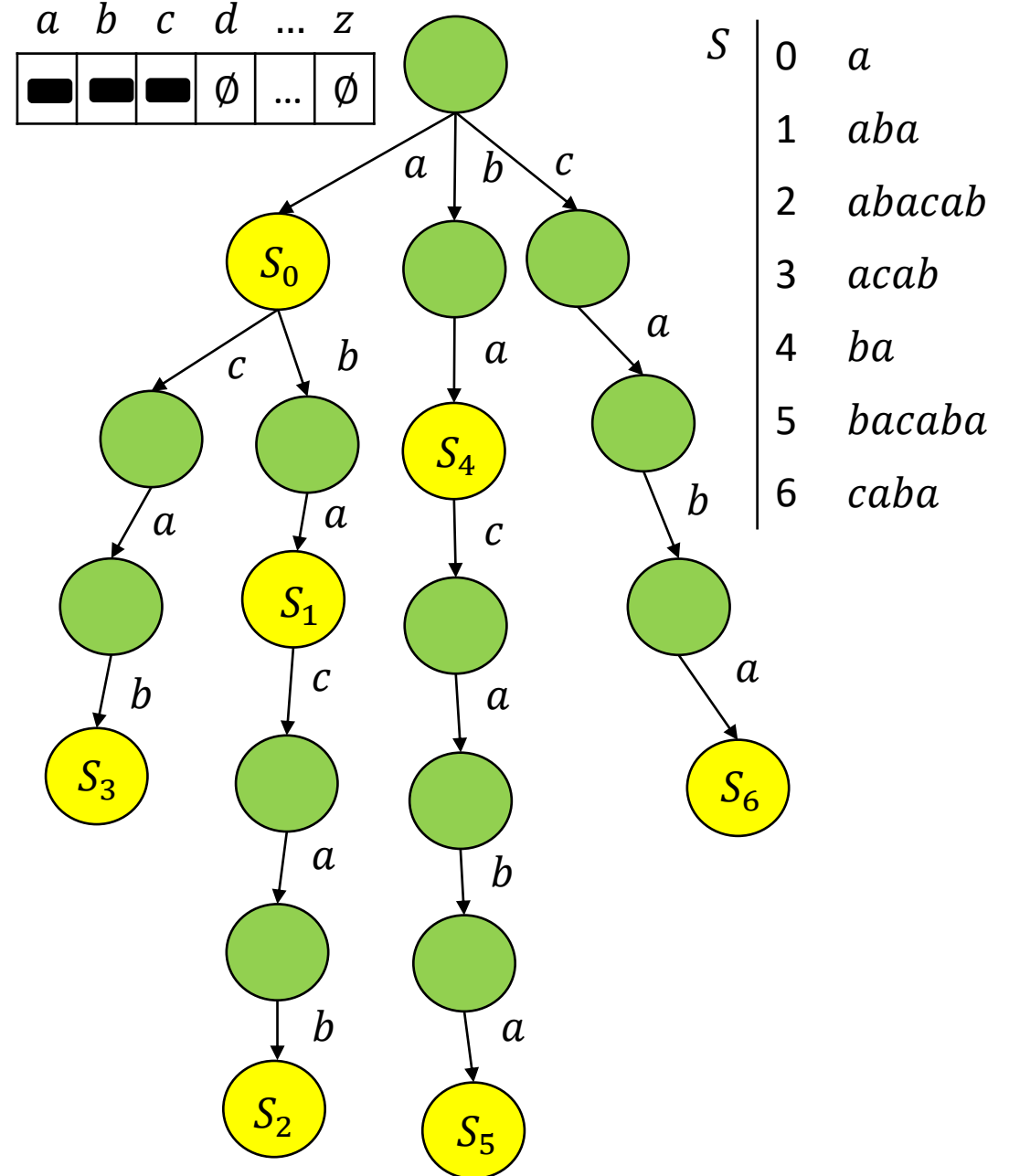


Удалить строку

из бора можно «по ленивому» – снять метку с соответствующей терминальной вершины.



ОЦЕНКИ



Время **построения** структуры данных бор для множества строк из $S = \{S_0, S_1, \dots, S_{n-1}\}$ есть

$$O\left(\sum_{i=0}^{n-1} |S_i|\right),$$

требуемая **память**:

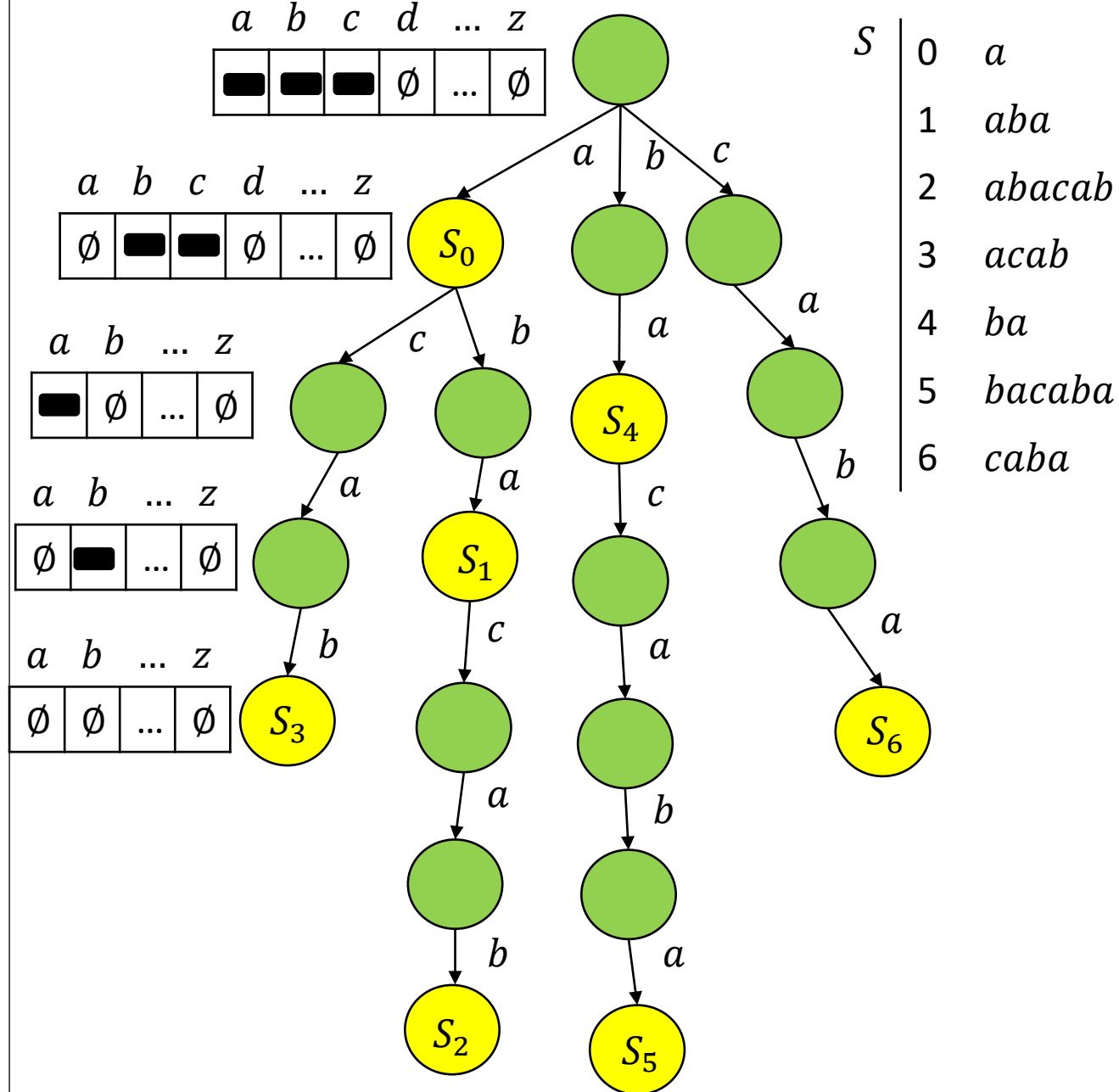
$$O\left(|\Sigma| \cdot \sum_{i=0}^{n-1} |S_i|\right),$$

где n – число строк в множестве S ,
 $|S_i|$ – длина i -ой строки,
 $|\Sigma|$ – размер алфавита.

Время **добавления/поиска** строки

$A = \{a_0, a_1, \dots, a_{l-1}\}$ зависит от длины строки A и не зависит от количества строк в боре:

$$O(l).$$



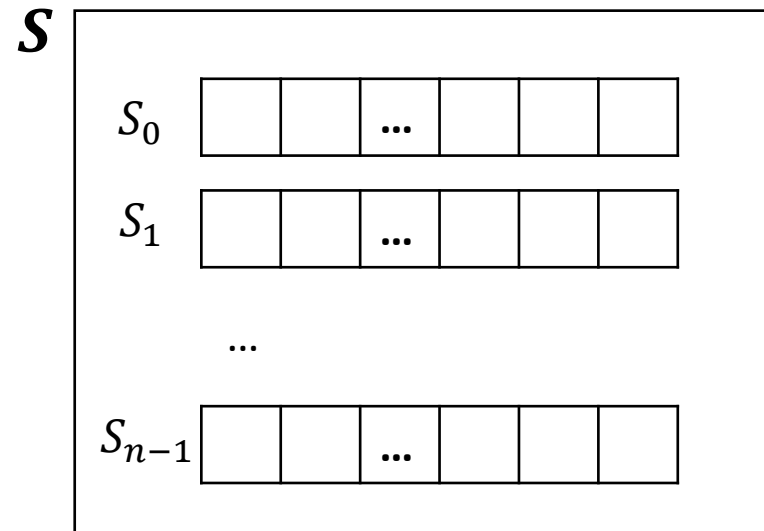
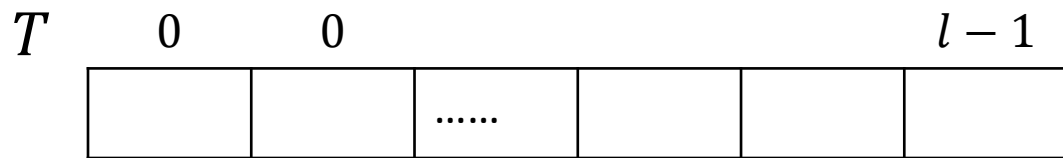
Множество строк $S = \{S_0, S_1, \dots, S_{n-1}\}$	Строка $A = (a_0, a_1, \dots, a_{l-1})$	Определить, есть ли строка A в S?
--	---	--

Структура данных	Время построения	Память	Время поиска
Массив	$O\left(\sum_{i=0}^{n-1} S_i \right)$	$O\left(\sum_{i=0}^{n-1} S_i \right)$	$O(A \cdot n)$
Сбалансированное бинарное поисковое дерево	$O(n \cdot \log n \cdot l_{\max}),$ $l_{\max} = \max_{i=0, \dots, n-1} S_i $	$O\left(\sum_{i=0}^{n-1} S_i \right)$	$O(A \cdot \log n)$
Бор	$O\left(\sum_{i=0}^{n-1} S_i \right)$	$O\left(\Sigma \cdot \sum_{i=0}^{n-1} S_i \right)$	$O(A)$

Задача поиска множества образцов в строке в режиме on-line

- Пусть у нас есть текст $T = (t_0, t_1, \dots, t_{l-1})$, и несколько образцов, которые поступают в режиме *on-line* из множества $S = \{S_0, S_1, \dots, S_{n-1}\}$.

Найти все подстроки текста T , которые совпадали бы с одним из образцов (т.е. необходимо определить, встречается ли образец из S в качестве подстроки в тексте T).



Задача поиска множества образцов в строке в режиме реального времени

Решить задачу можно эффективно,
используя такие структуры данных, как

- ❑ **суффиксный бор**

- ❑ **суффиксный массив**

Задача поиска множества образцов в строке в режиме реального времени

**Суффиксный
бор**

Суффиксный бор - структура данных бор, построенная для всех суффиксов текста $T = (t_0, t_1, \dots, t_{l-1})$.

$T = abacaba$

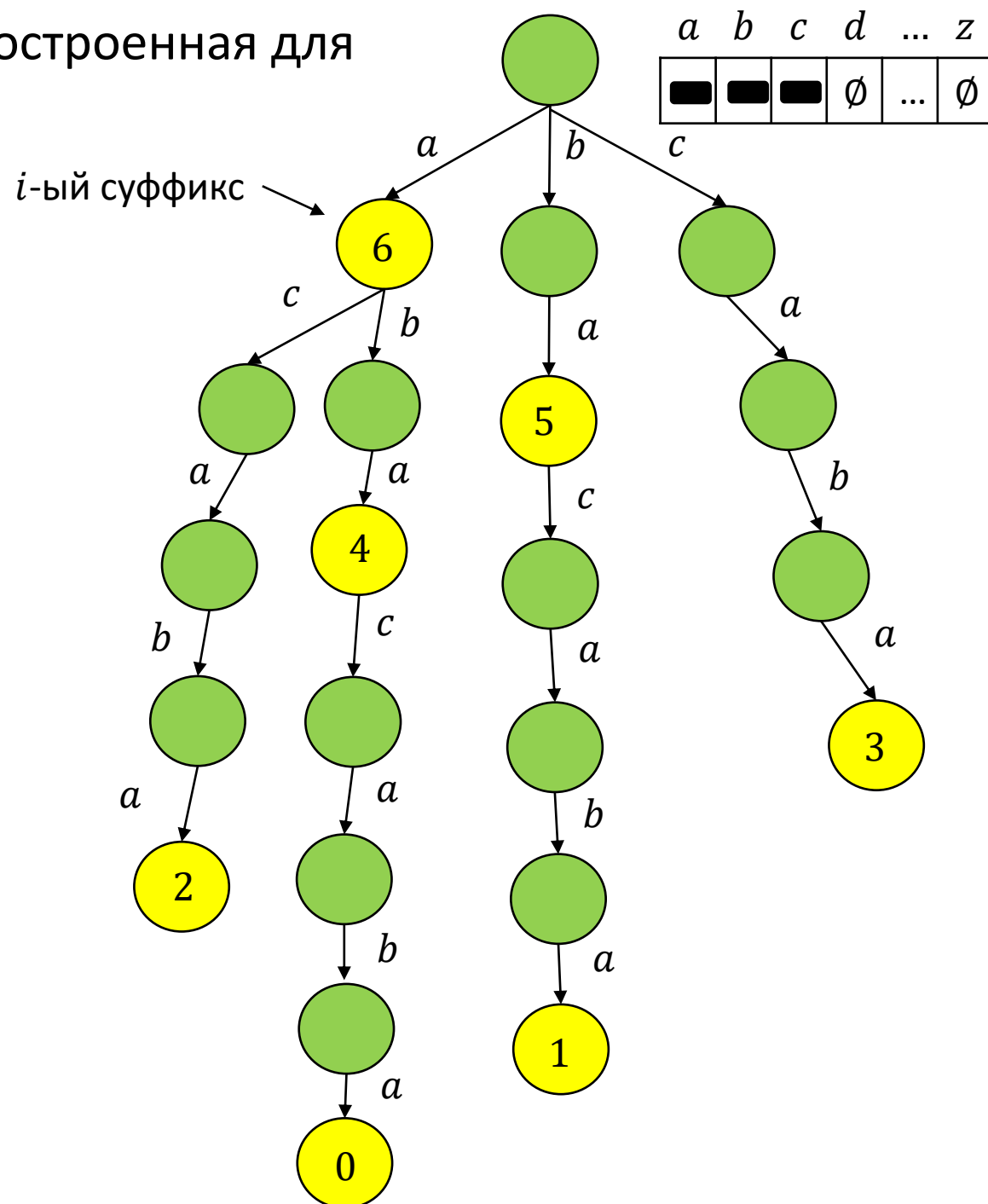
0	1	2	3	4	5	6
a	b	a	c	a	b	a

0-й	<i>abacaba</i>
1-й	<i>bacaba</i>
2-й	<i>acaba</i>
3-й	<i>caba</i>
4-й	<i>aba</i>
5-й	<i>ba</i>
6-й	<i>a</i>

Время построения : $O(l^2)$,

требуемая память: $O(|\Sigma| \cdot l^2)$,

где $l = |T|, |\Sigma|$ - размер алфавита.



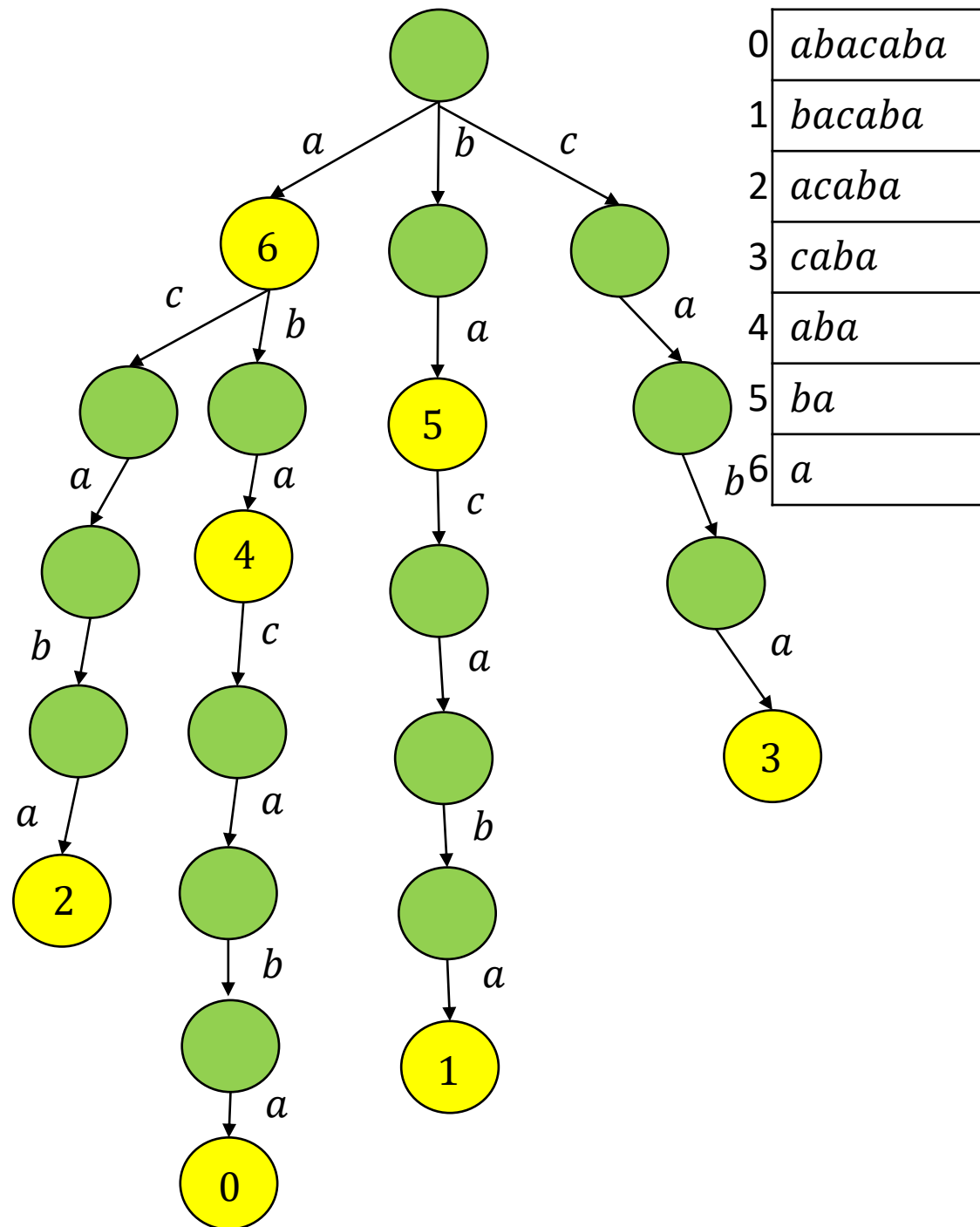
Если в **суффиксном боре** находится некоторый суффикс, то все префиксы этого суффикса также находятся в боре.

Если некоторая строка S_i встречается в качестве подстроки в строке T , то она является префиксом какого-то суффикса строки T . Поэтому суффиксный бор можно использовать для решения задачи поиска множества образцов в строке в режиме on-line.

Время поиска множества образцов в строке:

$$O\left(\sum_{i=0}^{n-1} |S_i|\right) = O(n \cdot l_{max}),$$

где $l_{max} = \max_{i=0, \dots, n-1} |S_i|$.



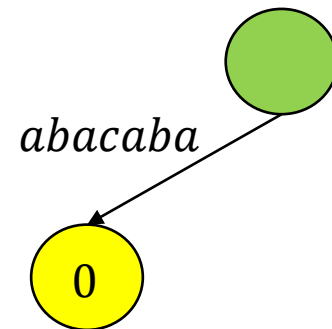
Оптимизация бора

Сжатый бор

Сжатый бор— это корневое дерево, на каждом ребре которого написана непустая строка, обладающее следующими свойствами:

- Ни из какой вершины не выходит два ребра, строки для которых начинаются на одну букву.
- Если вершина не является корнем или листом дерева, из нее выходит не менее двух ребер.

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.



Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

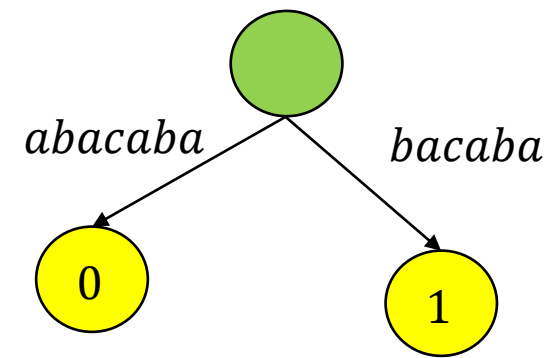
Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.

0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.

Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.

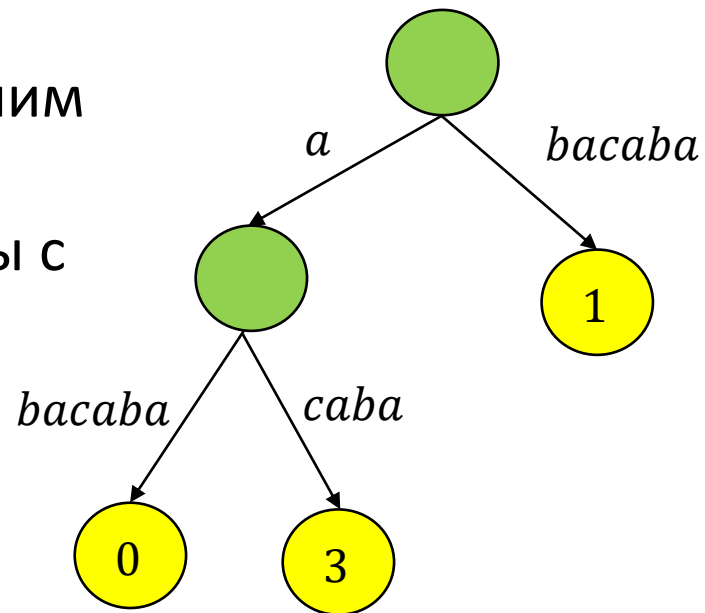


0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.

Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.

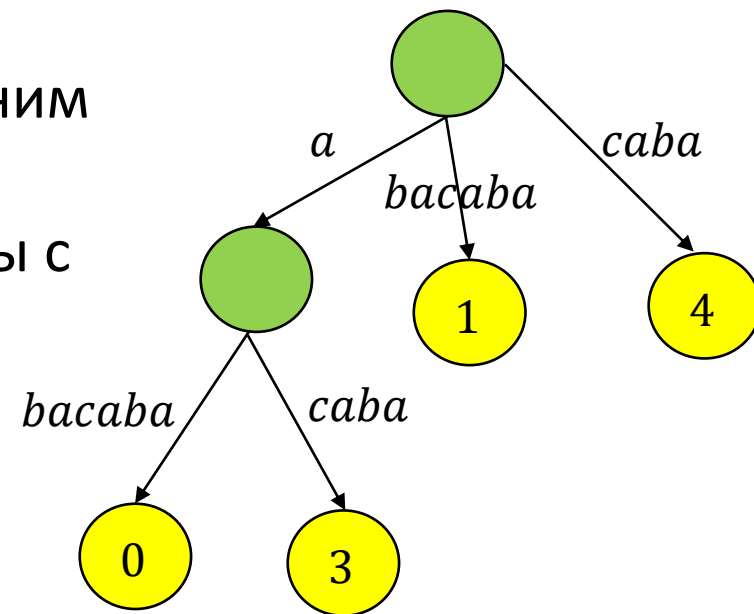


0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.

Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.

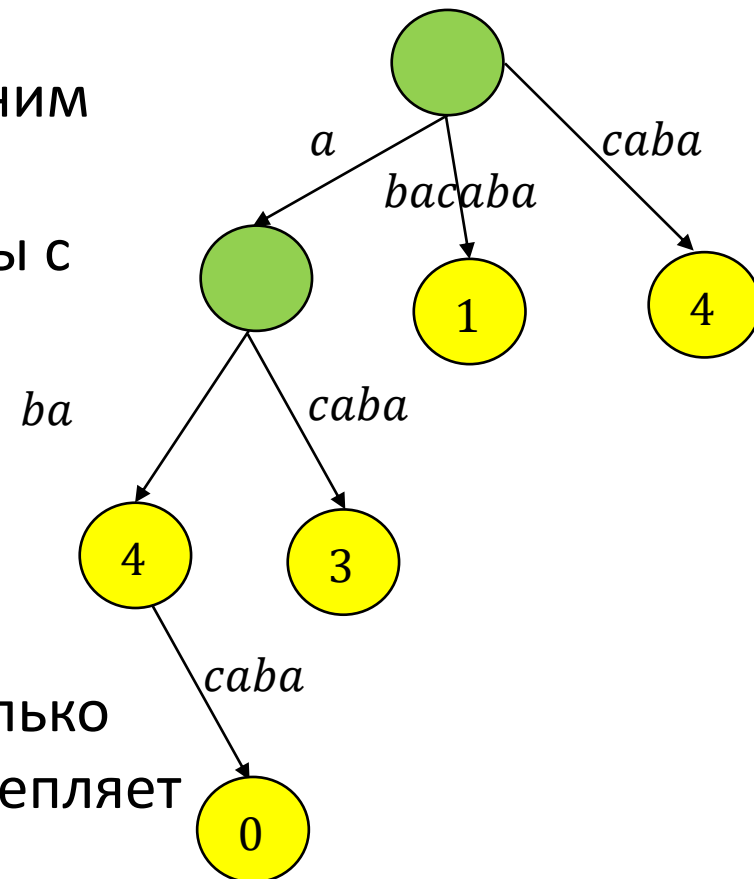


0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.

Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.

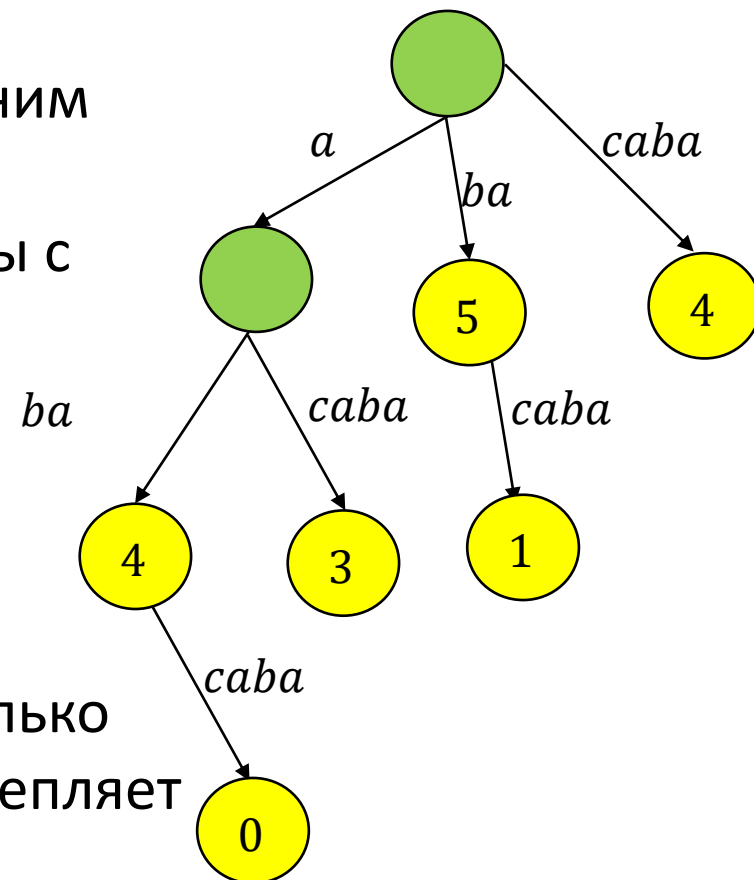


0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.

Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.

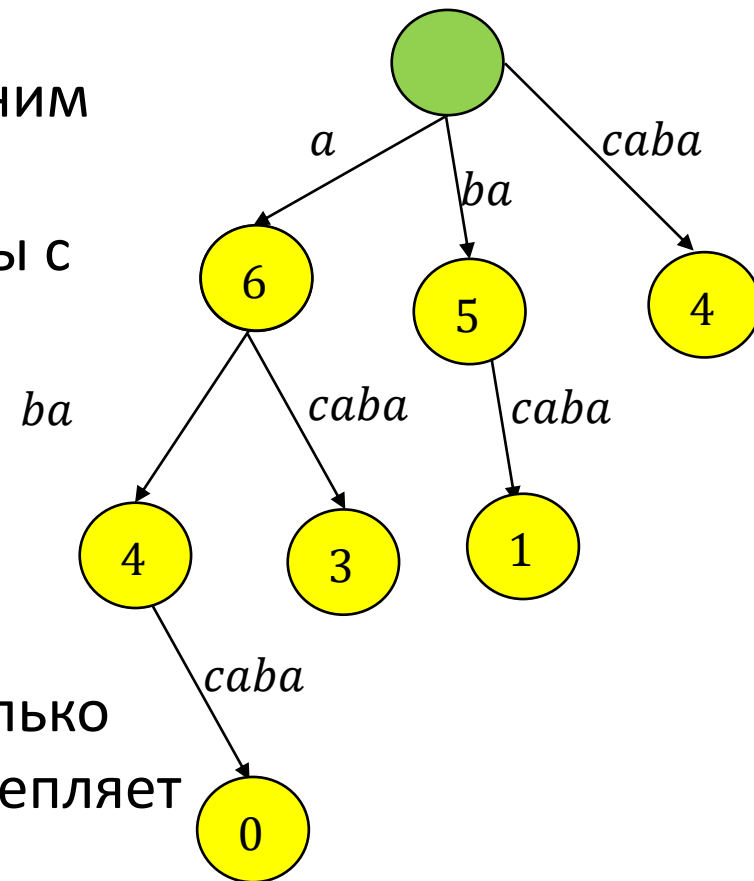


0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Сжатый бор – обычный бор, но на ребрах мы храним не одну букву, а множество символов, при этом сохраняется главное свойство, то, что все переходы с одной вершины начинаются с разных букв.

Эта идея чисто о том, чтобы избежать длинных цепочек вершин.

Если добавляется строка, у которой только есть только префикс какого-то перехода, то этот переход расщепляет строку перехода.



0	<i>abacaba</i>
1	<i>bacaba</i>
2	<i>acaba</i>
3	<i>caba</i>
4	<i>aba</i>
5	<i>ba</i>
6	<i>a</i>

Анализ

Время поиска множества образцов в строке:

$$O\left(\sum_{i=0}^{n-1} |S_i|\right) = O(n \cdot l_{max}),$$

где $l_{max} = \max_{i=0, \dots, n-1} |S_i|$.

Также как и в обычном боре.

Затраченная память:

$$O(|\Sigma| * n)$$

Действительно, количество листьев в сжатом боре (равно как и в обычном боре) не превосходит k , но теперь в дереве почти нет вершин исходящей степени 1, поэтому суммарное количество вершин не превосходит $2k = O(k)$.

Также, мы можем добиться такой памяти если каждое ребро бора будет содержать три числа, индекс строки, индекс начала в данной строке и индекс конца, т.е. таким образом мы не храним строки на ребрах явно, поэтому сам бор занимает столько памяти.

Сжатый бор занимает $O(|\Sigma| * n)$ памяти, однако, для операций с ним необходимо явно хранить все строки S_i , поэтому по памяти мы не выиграли. Зачем же он тогда нужен?

А что если построить сжатый суффиксный бор – получаем суффиксное дерево!

Задача поиска образца в строке в режиме реального времени

Задан фиксированный текст $T = (t_0, t_1, \dots, t_{l-1})$	On-line поступает образец S - строка	Определить, встречается ли образец S в качестве подстроки в тексте T	
Структура данных	Время построения	Память	Время поиска образца S
Суффиксный массив	$O(T \cdot \log^2 T)$ или $O(T \cdot \log T)$ или $O(T)$	$O(T)$	$O(\log T \cdot S)$
Суффиксный бор	$O(T ^2)$	$O(T ^2 \cdot \Sigma)$	$O(S)$
Суффиксный сжатый бор (суффиксное дерево)	$O(T)$	$O(T)$	$O(S)$
Префикс функция ака КМП	$O(T)$	$O(T)$	$O(S)$

Задача поиска множества образцов в строке в режиме реального времени

Задан фиксированный текст $T = (t_0, t_1, \dots, t_{l-1})$	On-line поступает образцы из множества $S = \{S_0, S_1, \dots, S_{n-1}\}$	Определить, встречается ли образец S_i в качестве подстроки в тексте T
---	---	--

Структура данных	Время построения	Память	Время поиска образца S
Суффиксный массив	$O(T \cdot \log^2 T)$ или $O(T \cdot \log T)$ или $O(T)$	$O(T)$	$O\left(\log T \cdot \sum_{i=0}^{n-1} S_i \right)$
Суффиксный бор	$O(T ^2)$	$O(T ^2 \cdot \Sigma)$	$O\left(\sum_{i=0}^{n-1} S_i \right)$
Суффиксный сжатый бор (суффиксное дерево)	$O(T)$	$O(T)$	$O\left(\sum_{i=0}^{n-1} S_i \right)$
Префикс функция ака КМП	$O(T)$	$O(T)$	$O\left(n * S + \sum_{i=0}^{n-1} S_i \right)$

Структура данных **суффиксный массив** была разработана в **1989** году .

Юджин Уимберли «Джин»

Майерс-младший

Eugene Wimberly «Gene» Myers, Jr.



Дата рождения	31 декабря 1953
Место рождения	• Бойсе , США
Страна	• США
Научная сфера	информатика

Уди Манбер

Udi Manber



Место рождения:	Кирьят-Хаим , Израиль
Страна	• США
Научная сфера	информатика
Должность	вице-президент Google по разработкам

Суффиксный массив строки $T = (t_0, t_1, \dots, t_{l-1})$
это последовательность **лексикографически отсортиро-
ванных суффиксов** строки T (очевидно, что достаточно
хранить только индексы суффиксов).

i	0	1	2	3	4	5	6	
T	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	$l = 7$

все суффиксы
строки T

0-й	<i>abacaba</i>	лексикографическая сортировка суффиксов	<i>a</i>
1-й	<i>bacaba</i>		<i>aba</i>
2-й	<i>acaba</i>		<i>abacaba</i>
3-й	<i>caba</i>		<i>acaba</i>
4-й	<i>aba</i>		<i>ba</i>
5-й	<i>ba</i>		<i>bacaba</i>
6-й	<i>a</i>		<i>caba</i>

6-й	<i>a</i>
4-й	<i>aba</i>
0-й	<i>abacaba</i>
2-й	<i>acaba</i>
5-й	<i>ba</i>
1-й	<i>bacaba</i>
3-й	<i>caba</i>

перестановка индексов суффиксов,
которая задаёт порядок суффиксов
в порядке лексикографической
сортировки

суффиксный
массив
 $P[0..l-1]$

0	6
1	4
2	0
3	2
4	5
5	1
6	3

Пусть есть фиксированный текст $T = (t_0, t_1, \dots, t_{l-1})$:

	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>
<i>T</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>

На вход в режиме реального времени поступают образцы $S_i = (s_0, s_1, \dots, s_{k-1})$ и нам необходимо эффективно определять, встречается ли образец S_i в качестве подстроки в тексте T (*on-line* версия задачи).

S_i	
a	b

Как, используя суффиксный массив, эффективно решить эту задачу?

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

	0	1
S_i	a	b

1) Построили по тексту T суффиксный массив P

0	6	a
1	4	aba
2	0	abacaba
3	2	acaba
4	5	ba
5	1	bacaba
6	3	caba

2) Если строка S_i является подстрокой T , то она является префиксом какого-то суффикса, поэтому дихотомией ($LowerBound, UpperBound$) по суффиксному массиву P (осуществляем поиск диапазона суффиксов, у которых префикс совпадает с образцом S_i).

S_i	
a	

0	6	a
1	4	aba
2	0	abacaba
3	2	acaba

$L = 0, \quad R = n$
 $L = LowerBound(L, R, S_i[0])$
 $R = UpperBound(L, R, S_i[0])$

$L = 0, R = 4$

если в диапазоне у суффиксов, у которых первый символ совпадает, удалить этот символ, то по следующему символу эти префиксы в этом диапазоне также упорядочены лексикографически

S_i	a	b
-------	--------------	----------

0	6	a
1	4	a ba
2	0	a bacaba
3	2	a caba

1	4	a ba
2	0	a bacaba

$L = LowerBound(L, R, S_i[1])$
 $R = UpperBound(L, R, S_i[1])$
 $L = 1, R = 3$

Время поиска вхождений образца

S_i в $T = (t_0, t_1, \dots, t_{l-1})$

$$\Theta(|S_i| \cdot \log |T|) = \Theta(|S_i| \cdot \log l).$$

Задача построения суффиксного массива

Построение суфф. массива

Идея построения	Время построения	Память
Лексикографическая сортировка	$\Theta\left(\sum_{i=0}^{l-1} t^i + l_{max} \cdot \Sigma \right)$	$O(T)$
Быстрая сортировка	$O(T ^2 \cdot \log T)$	$O(T)$
Классы эквивалентности + Быстрая сортировка	$O(T \cdot \log^2 T)$	$O(T)$
Классы эквивалентности + Сортировка подсчётом	$O(T \cdot \log T)$	$O(T)$
Алгоритм Карккайнена-Сандерса	$O(T)$	$O(T)$

Как построить суффиксный массив?

Алгоритм построения суффиксного массива для $T = (t_0, t_1, \dots, t_{l-1})$ за время

$$\Theta(|T|^2 + |T| \cdot |\Sigma|)$$

заключается, например, в непосредственном упорядочивании всех суффиксов строки T алгоритмом лексикографической сортировки.

Время работы
алгоритма лексикографической
сортировки

$$\Theta\left(\sum_{i=0}^{l-1} |t^j| + l_{max} \cdot |\Sigma|\right), \text{ где}$$

l — число кортежей,

l_{max} — длина самого длинного кортежа,

$|\Sigma|$ — число различных символов в кортежах.

Как построить суффиксный массив эффективнее?

Если для сортировки суффиксов строки $T = (t_0, t_1, \dots, t_{l-1})$ использовать любую сортировку за $O(N \cdot \log|N|)$, то время работы алгоритма построения суффиксного массива будет зависеть от того, каким образом будут сравниваться суффиксы.

Сравнение двух суффиксов в лоб работает за $O(|T|)$, это приведет к тому, что время работы алгоритма построения суффиксного массива:

$$O(|T| \cdot (|T| \cdot \log|T|)) = O(|T|^2 \cdot \log|T|).$$

Как научиться сравнивать строки быстрее, например за константное время?

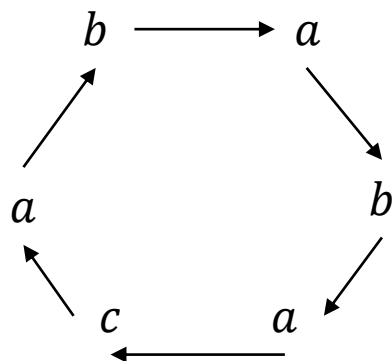
Алгоритм
построения суффиксного массива
строки $T = (t_0, t_1, \dots, t_{l-1})$ за время

$$O(|T| \cdot \log^2 |T|) = \mathbf{O}(l \cdot \log^2 l)$$

Циклический сдвиг строки *T*

это строка, которая получается из исходной строки *T* путем перемещением её первых символов в конец строки.

T = *abacaba*



циклический сдвиг длины 1	<i>bacabaa</i>
циклический сдвиг длины 2	<i>acabaab</i>
циклический сдвиг длины 3	<i>cabaaba</i>
циклический сдвиг длины 4	<i>abaabac</i>
циклический сдвиг длины 5	<i>baabaca</i>
циклический сдвиг длины 6	<i>aabacab</i>
циклический сдвиг длины 6	<i>abacaba</i>

лексикографически
минимальный
циклический сдвиг

Если к строке T добавить некоторый символ, код которого меньше всех кодов символов строки, например \$, выполнить лексикографическую сортировку всех циклических сдвигов $T + \$$, затем удалить из каждого циклического сдвига строки суффикс, который начинается с \$, то получим лексикографическую сортировку всех суффиксов строки T .

$T = abacaba$

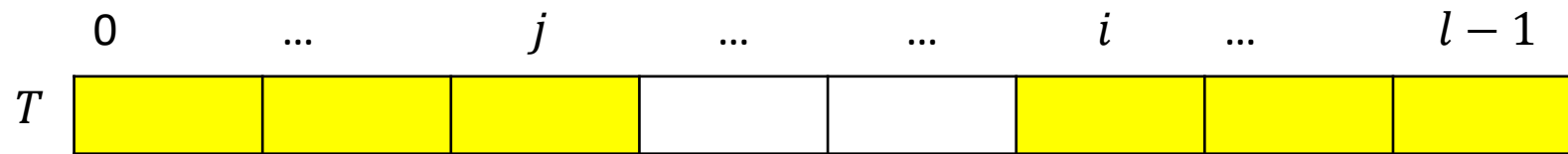
$T + \$ = abacaba + \$$

Циклические сдвиги $T + \$$	Лексикографическая сортировка	Лексикогр. упорядоченные суффиксы строки T
<i>bacaba\$a</i>	<i>\$abacaba</i>	
<i>acaba\$ab</i>	<i>a\$abacab</i>	<i>a</i>
<i>caba\$aba</i>	<i>aba\$abac</i>	<i>aba</i>
<i>aba\$abac</i>	<i>abacaba\$</i>	<i>abacaba</i>
<i>ba\$abaca</i>	<i>acaba\$ab</i>	<i>acaba</i>
<i>a\$abacab</i>	<i>ba\$abaca</i>	<i>ba</i>
<i>\$abacaba</i>	<i>bacaba\$a</i>	<i>bacaba</i>
<i>abacaba\$</i>	<i>caba\$aba</i>	<i>caba</i>

Таким образом мы свели задачу к сортировке циклических сдвигов $T + \$$.

Циклическая подстрока $t[i..j]$ определяется, как

$$\begin{cases} t[i..l-1] + t[0..j], & \text{если } i > j \\ t[i..j], & \text{если } i \leq j \end{cases}$$



Таким образом, нам необходимо отсортировать циклические подстроки длины L .

Ключевая идея

Выполним **k** фаз сортировки циклических подстрок.

На i -ой фазе сортируются циклические подстроки длины $2^i, 0 \leq i \leq k$.

Чему равно **k** ?

Пусть $l = |T|$, $2^{k-1} < l \leq 2^k$.

Тогда $2^{k-1} < l$
 $k < \log_2 l + 1$
 $k = \lceil \log_2 l \rceil$

Например, пусть

$T + \$ = \textit{abacaba} + \$$

Отсортируем циклические строки длины 1

Номер суффикса	Циклический сдвиг	Циклическая строка длины 1
0	<i>abacaba\$</i>	<i>a</i>
1	<i>bacaba\$a</i>	<i>b</i>
2	<i>acaba\$ab</i>	<i>a</i>
3	<i>caba\$aba</i>	<i>c</i>
4	<i>aba\$abac</i>	<i>a</i>
5	<i>ba\$abaca</i>	<i>b</i>
6	<i>a\$abacab</i>	<i>a</i>
7	<i>\$abacaba</i>	<i>\$</i>

лексикографическая
сортировка
циклических строк
длины 1



Номер суффикса	Циклический сдвиг	Циклическая строка длины 1
7	<i>\$abacaba</i>	<i>\$</i>
0	<i>abacaba\$</i>	<i>a</i>
2	<i>acaba\$ab</i>	<i>a</i>
4	<i>aba\$abac</i>	<i>a</i>
6	<i>a\$abacab</i>	<i>a</i>
1	<i>bacaba\$a</i>	<i>b</i>
5	<i>ba\$abaca</i>	<i>b</i>
3	<i>caba\$aba</i>	<i>c</i>

$T + \$ = abacaba + \$$

Отсортируем циклические строки длины 2

Номер суффикса	Циклический сдвиг	Циклическая строка длины 2
0	<i>abacaba\$</i>	<i>ab</i>
1	<i>bacaba\$a</i>	<i>ba</i>
2	<i>acaba\$ab</i>	<i>ac</i>
3	<i>caba\$aba</i>	<i>ca</i>
4	<i>aba\$abac</i>	<i>ab</i>
5	<i>ba\$abaca</i>	<i>ba</i>
6	<i>a\$abacab</i>	<i>a\$</i>
7	<i>\$abacaba</i>	<i>\$a</i>

лексикографическая
сортировка
циклических строк
длины 2



Номер суффикса	Циклический сдвиг	Циклическая строка длины 2
7	<i>\$abacaba</i>	<i>\$a</i>
6	<i>a\$abacab</i>	<i>a\$</i>
0	<i>abacaba\$</i>	<i>ab</i>
4	<i>aba\$abac</i>	<i>ab</i>
2	<i>acaba\$ab</i>	<i>ac</i>
1	<i>bacaba\$a</i>	<i>ba</i>
5	<i>ba\$abaca</i>	<i>ba</i>
3	<i>caba\$aba</i>	<i>ca</i>

$T + \$ = abacaba + \$$

Отсортируем циклические строки длины 4

Номер суффикса	Циклический сдвиг	Циклическая строка длины 4
0	<i>abacaba\$</i>	<i>abac</i>
1	<i>bacaba\$a</i>	<i>baca</i>
2	<i>acaba\$ab</i>	<i>acab</i>
3	<i>caba\$aba</i>	<i>caba</i>
4	<i>aba\$abac</i>	<i>aba\$</i>
5	<i>ba\$abaca</i>	<i>ba\$a</i>
6	<i>a\$abacab</i>	<i>a\$ab</i>
7	<i>\$abacaba</i>	<i>\$aba</i>

лексикографическая
сортировка
циклических строк
длины 4



Номер суффикса	Циклический сдвиг	Циклическая строка длины 4
7	<i>\$abacaba</i>	<i>\$aba</i>
6	<i>a\$abacab</i>	<i>a\$ab</i>
4	<i>aba\$abac</i>	<i>aba\$</i>
0	<i>abacaba\$</i>	<i>abac</i>
2	<i>acaba\$ab</i>	<i>acab</i>
5	<i>ba\$abaca</i>	<i>ba\$a</i>
1	<i>bacaba\$a</i>	<i>baca</i>
3	<i>caba\$aba</i>	<i>caba</i>

$T + \$ = abacaba + \$$

Отсортируем циклические строки длины 8

Номер суффикса	Циклический сдвиг	Циклическая строка длины 8
0	<i>abacaba\$</i>	<i>abacaba\$</i>
1	<i>bacaba\$a</i>	<i>bacaba\$a</i>
2	<i>acaba\$ab</i>	<i>acaba\$ab</i>
3	<i>caba\$aba</i>	<i>caba\$aba</i>
4	<i>aba\$abac</i>	<i>aba\$abac</i>
5	<i>ba\$abaca</i>	<i>ba\$abaca</i>
6	<i>a\$abacab</i>	<i>a\$abacab</i>
7	<i>\$abacaba</i>	<i>\$abacaba</i>

лексикографическая
сортировка
циклических строк
длины 8



Номер суффикса	Циклический сдвиг	Циклическая строка длины 8
7	<i>\$abacaba</i>	<i>\$abacaba</i>
6	<i>a\$abacab</i>	<i>a\$abacab</i>
4	<i>aba\$abac</i>	<i>aba\$abac</i>
0	<i>abacaba\$</i>	<i>abacaba\$</i>
2	<i>acaba\$ab</i>	<i>acaba\$ab</i>
5	<i>ba\$abaca</i>	<i>ba\$abaca</i>
1	<i>bacaba\$a</i>	<i>bacaba\$a</i>
3	<i>caba\$aba</i>	<i>caba\$aba</i>

Получили ответ, но что делать, если $|T + \$|$ - не степень двойки?

Заметим, что $j + 1$ фазу стоит делать тогда и только тогда, когда на j -ой фазе есть хотя бы две одинаковые строки.

Если $|T + \$|$ не степень двойки. Например, если $T = \mathbf{abaca} + \$$, какие строки надо сортировать на 3ей фазе, когда длина циклической строки = 8?

Представим, что мы зациклим циклическую строку. Получим следующую таблицу.

Номер суффикса	Циклический сдвиг	Циклическая строка длины 8
0	<i>abaca\$</i>	<i>abaca\$ab</i>
1	<i>baca\$a</i>	<i>baca\$aba</i>
2	<i>aca\$ab</i>	<i>aca\$abac</i>
3	<i>ca\$aba</i>	<i>ca\$abaca</i>
4	<i>a\$abac</i>	<i>a\$abaca\$</i>
5	<i>\$abaca</i>	<i>\$abaca\$a</i>

Заметим, что эти циклические строки всегда однозначно сортируются по первым $|T + \$|$ символам, т.к. на каждой из первых $|T + \$|$ позиций только в одном месте находится \$, значит как минимум в этих позициях строки различаются → символы на следующих позициях не влияют на ответ.

Таким образом мы показали, что достаточно сделать $k = \lceil \log_2 l \rceil$ фаз сортировок, т.к. на k — ой фазе $l \leq 2^k$. Теперь осталось научиться выполнять сортировки на этих фазах!

На j — ой фазе поддерживаются следующие массивы:

P^j — перестановка индексов, которая получится, если отсортировать циклические строки длины 2^j (левый столбец в предыдущих примерах).

C^j — массив классов эквивалентности (элементы массива - целые числа ≥ 0), где

- ✓ $c^j[i]$ — номер класса эквивалентности для циклической подстроки длины 2^j , начинающейся в позиции i ;
- ✓ Если циклическая строка, начинающаяся в позиции i_1 длины 2^j равна циклической строке, начинающейся в позиции i_2 длины 2^j то $c^j[i_1] == c^j[i_2]$
- ✓ Если циклическая строка, начинающаяся в позиции i_1 длины 2^j лексикографически меньше циклической строки, начинающейся в позиции i_2 длины 2^j то $c^j[i_1] < c^j[i_2]$

То есть, грубо говоря, равные циклические строки получают равные классы эквивалентности.

А для различных циклических строк, их отношение лексикографического порядка сохраняется и на массиве классов эквивалентностей.

Таким образом, теперь мы сравниваем не циклические строки, а только лишь их классы эквивалентности.

Рассмотрим нулевую фазу, когда длинна циклических строк = 1.

Нам необходимо отсортировать циклические строки длины 1, это легко сделать используя сортировку подсчетом, например:

i	0	1	2	3	4	5	6	7
T	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	\$
P^0	7	0	2	4	6	1	5	3
C^0	1	2	1	3	1	2	1	0

Сформировать указанные массивы можно устойчивым алгоритмом **сортировки подсчётом** за время $O(l + |\Sigma|)$

Тут, в массиве C^0 следующие классы эквивалентности.

\$ - класс эквивалентности 0

a - класс эквивалентности 1

b - класс эквивалентности 2

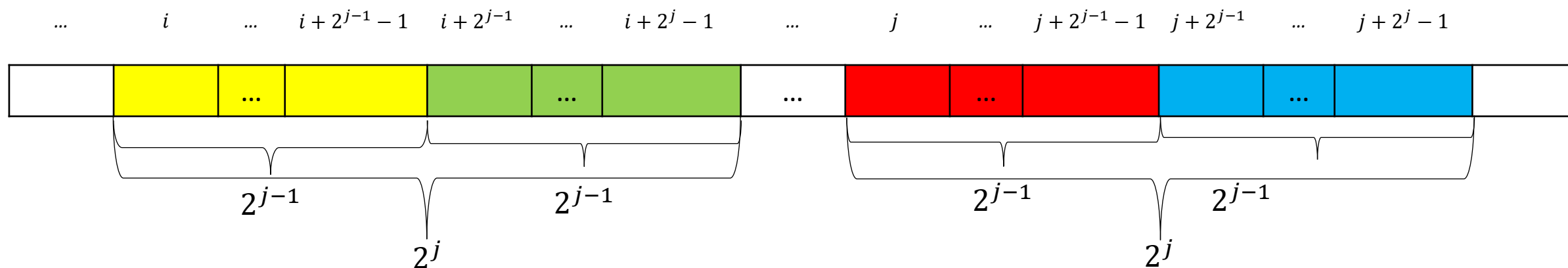
c - класс эквивалентности 3

Соответственно, в массиве P^0 сначала идут индексы с классом эквивалентности 0, т.к. они лексикографически минимальны, потом класс эквивалентности 1, 2 и т.д.

На фазах, начиная с 1-й, необходимо лексикографически упорядочивать подстроки длины которых больше 1.

Делать это можно, например, сортировкой слиянием, но надо научиться, используя информацию, полученную на предыдущих фазах, сравнивать две циклические подстроки одинаковой длины за константное время.

Сравнение двух циклических подстрок на j -ой фазе можно **выполнить за константное время**, так как мы знаем для каждой циклической подстроки длины 2^{j-1} к какому классу эквивалентности она отнесена, а любая циклическая подстрока длины 2^j состоит из двух циклических подстрок длины 2^{j-1} .



если $c^{j-1}[i] < c^{j-1}[j]$, то $t[i..i + 2^j - 1] < t[j..j + 2^j - 1]$

если $c^{j-1}[i] > c^{j-1}[j]$, то $t[i..i + 2^j - 1] > t[j..j + 2^j - 1]$

если $c^{j-1}[i] = c^{j-1}[j]$, то

если $c^{j-1}[i + 2^{j-1}] < c^{j-1}[j + 2^{j-1}]$, то $t[i..i + 2^j - 1] < t[j..j + 2^j - 1]$

иначе, если $c^{j-1}[i + 2^{j-1}] > c^{j-1}[j + 2^{j-1}]$, то $t[i..i + 2^j - 1] > t[j..j + 2^j - 1]$

иначе $t[i..i + 2^j - 1] = t[j..j + 2^j - 1]$

$l = 8$

i

0

1

2

3

4

5

6

7

T

P^0

C^0

Цикл
стр

a	b	a	c	a	b	a	\$
7	0	2	4	6	1	5	3
1	2	1	3	1	2	1	0
a	b	a	c	a	b	a	\$

- классы эквивалентности на 0-ой фазе

$c[p[0]] = 0$
 для $1 \leq i \leq l - 1$
 если $t[p[i]..p[i] + 2^j - 1] = t[p[i - 1]..p[i - 1] + 2^j - 1]$,
 то $c[p[i]] = c[p[i - 1]]$, иначе $c[p[i]] = c[p[i - 1]] + 1$).

2^1

i

0

1

2

3

4

5

6

7

T

P^1

C^1

Цикл.
Стр

a	b	a	c	a	b	a	\$
7	6	0	4	2	1	5	3
2	4	3	5	2	4	1	0
$a + b$ 1кл + 2кл	$b + a$ 2кл + 1кл	$a + c$ 1кл + 3кл	$c + a$ 3кл + 1кл	$a + b$ 1кл + 2кл	$b + a$ 2кл + 1кл	$a + \$$ 1кл + 0кл	$\$ + a$ 0кл + 1кл

Отсортировав эти пары чисел, мы можем построить массив классов эквивалентностей

2¹

i

0

1

2

3

4

5

6

7

T

*p*¹

*C*¹

Цикл.
Стр

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	\$
7	6	0	4	2	1	5	3
2	4	3	5	2	4	1	0
<i>ab</i>	<i>ba</i>	<i>ac</i>	<i>ca</i>	<i>ab</i>	<i>ba</i>	<i>a</i> \$	\$ <i>a</i>

$c[p[0]] = 0$
 для $1 \leq i \leq l - 1$
 если $s[p[i]..p[i] + 2^j - 1] = s[p[i - 1]..p[i - 1] + 2^j - 1]$,
 то $c[p[i]] = c[p[i - 1]]$, иначе $c[p[i]] = c[p[i - 1]] + 1$).

2²

i

0

1

2

3

4

5

6

7

T

*p*²

*C*²

Цикл.
Стр

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	\$
7	6	4	0	2	5	1	3
3	6	4	7	2	5	1	0
<i>ab + ac</i> 2кл + 3кл	<i>ba + ca</i> 4кл + 5кл	<i>ac + ab</i> 3кл + 2кл	<i>ca + ba</i> 5кл + 4кл	<i>ab + a</i> \$ 2кл + 1кл	<i>ba + \$a</i> 4кл + 0кл	<i>a</i> \$ + <i>ab</i> 1кл + 2кл	\$ <i>a</i> + <i>ba</i> 0кл + 4кл

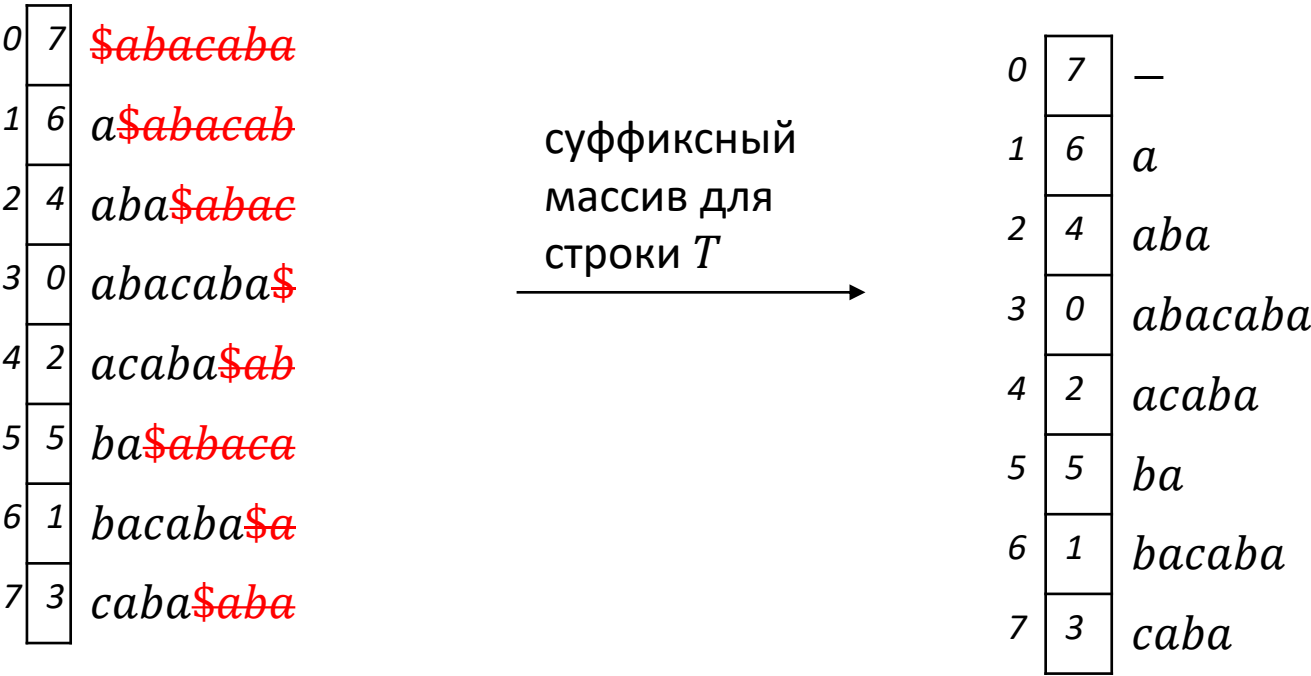
Массив C^j формируется на j —ой фазе за время $\Theta(l)$ после формирования массива P^j , проходом по массиву P^j слева направо и сравнением (за константу) двух циклических подстрок длины 2^j на равенство:

- номер класса для $t[p^j[0]..p^j[0] + 2^j - 1]$ полагаем равным 0, т.е. $c[p^j[0]] := 0$;
- для $1 \leq i \leq l$
если $t[p^j[i]..p^j[i] + 2^j - 1] = t[p^j[i-1]..p^j[i-1] + 2^j - 1]$, то номер класса $c^j[p^j[i]]$ для $t[p^j[i]..p^j[i] + 2^k - 1]$ остается таким же, как и для $t[p^j[i-1]..p^j[i-1] + 2^k - 1]$ (т. е. $c^j[p^j[i]] = c^j[p^j[i-1]]$), иначе он увеличивается на 1 (т. е. $c^j[p^j[i]] := c^j[p^j[i-1]] + 1$).

Так как к строке T добавляется \$, то это гарантирует, что каждый суффикс будет иметь свой класс эквивалентности.

После того, как выполнена сортировка циклических сдвигов строки $T + \$$, удалим из каждого циклического сдвига строки её суффикс, который начинается с $\$$, получим суффиксный массив для строки T :

i	0	1	2	3	4	5	6	7	$l = 8$
T	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	\$	
P^3	7	6	4	0	2	5	1	3	
C^3	3	6	4	7	2	5	1	0	



Если на j -й фазе выполнять лексикографическую сортировку l циклических подстрок длины 2^j ($j < \log_2 |T| + 1$) **сортировкой слиянием**, то так как сравнение двух циклических подстрок мы выполняем за $O(1)$, а массив C формируется за $O(|T|)$, то время выполнения одной фазы

$$O(|T| \cdot \log |T|) + O(|T|) = O(|T| \cdot \log |T|).$$

Учитывая, что число фаз $k = \lceil \log |T| \rceil < \log |T| + 1$, получаем, что **время работы алгоритма лексикографической сортировки циклических сдвигов строки** (построения суффиксного массива строки T)

$$O(|T| \cdot \log^2 |T|).$$

Требуемая память — $O(|T|)$.

Чтобы немного ускорить построение, воспользуемся идеей цифровой сортировки – сначала отсортируем массив пар по второму значению, потом по первому – но обязательно устойчивой сортировкой.

Для обеих ситуаций подойдет сортировка подсчетом за $O(|T|)$.

Таким образом, время выполнения одной фазы

$$O(|T|) + O(|T|) = O(|T|).$$

Учитывая, что число фаз $k = \lceil \log |T| \rceil$, получаем, что **время работы алгоритма лексикографической сортировки циклических сдвигов строки** (построения суффиксного массива строки T)

$$O(|T| \cdot \log |T|).$$

Требуемая **память** – $O(|T|)$.

Как построить **суфмас** за
время $O(|T|)$?

(а главное, зачем?)

Алгоритм Карккайнена-Сандерса

Схема идеи:

- Будем строить рекурсивно.
- Функция **`vector<int> BuildSuffArr(vector<int> C)`** возьмёт строку и вернёт построенный суффиксный массив этой строки.
- Принимать строку она будет в виде массива классов эквивалентности, то есть на первом этапе, к примеру, заменим каждый символ строки его номером в алфавите.
- Внутри функции произойдёт один рекурсивный вызов себя на данных меньшего размера, а также некоторые манипуляции, подробно про которые далее.

a	b	o	b	a
0	1	2	0	1

- Разделим суффиксы строки на три категории: **0**-суффиксы, **1**-суффиксы и **2**-суффиксы по остатку от деления на 3 (индексация с нуля)
- В примере, **boba** – **1**-суффикс, **ba** – **0**-суффикс
- Обозначим упорядоченную тройку символов (T_i, T_{i+1}, T_{i+2}) как tr_i
- Определим, в каком порядке **1**-суффиксы и **2**-суффиксы будут выстроены в итоговом суфмассе. Зная этот порядок, позже мы сможем определить и порядок **0**-суффиксов.

Новый алфавит

- Для этого рассмотрим строку $S^{(3)} = T_1 \dots T_n T_2 \dots T_{n+1}$ (за n примем $3 * \left\lceil \frac{|T|}{3} \right\rceil$, при необходимости допихнув в конец T символы **#**), в которой подряд выписаны сначала все **1**-тройки, затем все **2**-тройки. Перекодируем S в новый алфавит на множестве всех tr_i . А чтобы отсортировать тройки, вспомним *сортировку подсчётом* из предыдущего алгоритма.

index	T_i	T_i	tr_i	В порядке сортировки	нумеруем	В новом алфавите
0	A	0	(A, B, O)	(A, #, #)	0	1
1	B	1	(B, O, B)	(A, B, O)	1	3
2	O	2	(O, B, O)	(B, A, #)	2	5
3	B	1	(B, O, B)	(B, O, B)	3	3
4	O	2	(O, B, A)	(B, O, B)	3	4
5	B	1	(B, A, #)	(O, B, A)	4	2
6	A	0	(A, #, #)	(O, B, O)	5	0

i	0			1			2			3		
$\mathcal{S}^{(3)}$	BOB			OBA			OBO			BA#		
S_i	3			4			5			2		
j	1	2	3	4	5	6	2	3	4	5	6	-
T_j	B	O	B	O	B	A	O	B	O	B	A	#



index	T_i	tr_i	tr_i
0	0	(A, B, O)	1
1	1	(B, O, B)	3
2	2	(O, B, O)	5
3	1	(B, O, B)	3
4	2	(O, B, A)	4
5	1	(B, A, #)	2
6	0	(A, #, #)	0

Зачем строка S ?

- Построим рекурсивно её суффиксный массив.
- Теперь можно восстановить суффиксный массив **1**-суффиксов и **2**-суффиксов T .
- Жёлтые (из первой половины S) индексы соответствуют **1**-суффиксам, зелёные (из второй половины) – **2**-.

S	3	4	5	2
sufarr	3	0	1	2
Суффиксы T	BA	BOBOBA	OBA	OBOBA
Индекс суффикса	5	1	4	2

Часть с 0-суффиксами

T_i	A	B	O	B	O	B	A
i	0	1	2	3	4	5	6
0-суффикс				B	O	B	A
1-суффикс					O	B	A

- Теперь определим порядок **0**-суффиксов T . После предыдущего шага мы умеем сравнивать лексикографически **1**-суффиксы и **2**-суффиксы за $O(1)$, так как известны их позиции в суфмасе.
- Но что нужно, чтобы сравнить пару **0**-суффиксов?
- Заметим, что **0**-суффикс представляет собой **1**-суффикс, к которому спереди дописали символ. Выходит, что отсортировать **0**-суффиксы — то же самое, что отсортировать упорядоченные пары (символ, **1**-суффикс). Здесь снова поможет *сортировка подсчётом*.

Сливаем

- Теперь получились по отдельности суффиксные массивы для **0**-суффиксов и для **1**-,**2**-суффиксов.
- И всё это за линейное время!
- Осталось их слить.
- Используем обычный алгоритм слияния отсортированных массивов (как в **MergeSort**). Но есть один нюанс.
- Чтобы написать **Merge**, придётся научиться сравнивать **0**-суффикс и **1**-суффикс, а также **0**-суффикс и **2**-суффикс. Используем тот же трюк, что в **0-0** сравнении.

0-1 сравнение

- **1**-суффикс расписываем как пару (символ, **2**-суффикс)
- **0**-суффикс расписываем как пару (символ, **1**-суффикс)
- Символы мы сравнивать умеем, **1**-суффикс с **2**-суффиксом тоже

0-2 сравнение

- **2**-суффикс расписываем как тройку (символ, символ, **1**-суффикс)
- **0**-суффикс расписываем как тройку (символ, символ, **2**-суффикс)
- Аналогично, получаем сравнение упорядоченных троек

Успешный успех.

- Таким образом, был построен суффиксный массив строки T .
Подсчитаем сложность:
 - пара сортировок подсчётом за линию
 - рекурсивный вызов на $\frac{2}{3}$ длины
 - некоторая линейная возня
- Итого, $T(N) = T(\frac{2}{3}N) + O(N)$. Если решить, это $O(N)$. Честно.

Псевдокод

```
vector<int> BuildSuffArr(vector<int> C):  
    if C.length < слишком_мало:  
        std::sort  
        return  
    RecodeTriplesAlphabet(C) // строим новый алфавит  
    S := ToNewAlphabet(C[1:] + C[2:]) // строим строку S, используя новый алфавит  
    s12 := BuildSuffArr(S) // вызываем функцию, получаем суфмас для 1- и 2-  
    s0 := ... // формируем суфмас для 0- сортировкой подсчётом, используя s12  
    return merge(s0, s12) // сливаем s0 и s12 за линейное время
```

Для не понявших, но стремящихся

- Тут подробнее:



Массив наибольших общих префиксов. Построение и применение

LCP: Longest Common Prefix

LCP

Массив LCP — это массив длин наибольших общих префиксов для всех соседних суффиксов строки, отсортированных в лексикографическом порядке.

Пример

Пусть дана строка $S = \text{"abacabadabacaba"}$

1	a
3	aba
7	abacaba
3	abacabadabacaba
1	abadabacaba
5	acaba
1	acabadabacaba
0	adabacaba
2	ba
6	bacaba
2	bacabadabacaba
0	badabacaba
4	caba
0	cabadabacaba
	dabacaba

Как построить массив lcp?

Алгоритм Касай, Аримур, Ариквы, Ли, Парка

Пусть дана строка $S = \text{“abaabbbaa”}$

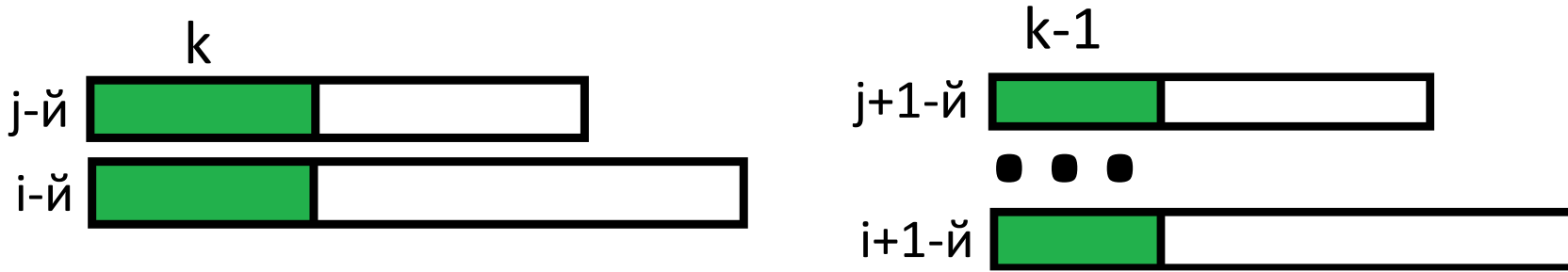
1	a	7
2	aa	6
	#	
1	aabbbaa	2
	#	
2	abaabbbaa	0
	#	
0	abbaa	3
	#	
3	baa	5
1	baabbbaa	1
	#	
	bbaa	4

Последовательно рассмотрим суффиксы строки S начиная с 0-го. Будем находить l_{sr} очередного суффикса и предыдущего для него в суфмассе.

Посчитаем l_{sr} 0-го суффикса и предыдущего для него в суфмассе.

- Пусть
1. Уже просмотрели i суффиксов.
 2. Предыдущий для i -го суффикса является j -й.
 3. l_{sr} i -го и j -го суффиксов равно k .

Тогда l_{sr} $i+1$ -го и $j+1$ -го суффикса равно $k-1$.
Но $i+1$ -й и $j+1$ -й суффиксы могут находиться далеко друг от друга.
Т.к. суффиксы в суфмассе отсортированы, то l_{sr} $i+1$ -го и предыдущего для него суффикса как минимум $k-1$.



Асимптотика

$O(|S|)$

Как доказывать?

Аналогично Z-функции

Допустим мы построили массив lcp , где lcp_i равен наибольшему общему префиксу суффиксов p_i и p_{i+1} .
Как же его использовать?

Задача: По заданной строке s после предподсчёта научиться отвечать на запросы вида «наибольший общий префикс для двух произвольных суффиксов i и j »

Задача: По заданной строке s после предподсчёта научиться отвечать на запросы вида «наибольший общий префикс для двух произвольных суффиксов i и j »

Пусть поступил запрос с некоторыми индексами суффиксов i и j .


Найдём эти индексы в суффиксном массиве, т.е. пусть K_1 и K_2 — их позиции в суффиксном массиве (упорядочим их, т.е. пусть $K_1 < K_2$).

Тогда ответом на данный запрос будет минимум в массиве lcp , взятый на отрезке $[K_1..K_2 - 1]$. В самом деле, переход от суффикса i к суффиксу j можно заменить целой цепочкой переходов, начинающейся с суффикса i и заканчивающейся в суффиксе j , но включающей в себя все промежуточные суффиксы, находящиеся в порядке сортировки между ними.

Таким образом, если мы имеем такой массив lcp , то ответ на любой запрос наибольшего общего префикса сводится к запросу минимума на отрезке массива lcp . Эта классическая задача минимума на отрезке (RMQ), которую можно решить, используя такие структуры данных, как дерево отрезков, разреженные таблицы, корневая декомпозиция и другое.

Пусть дана строка $S = abacabadabacaba$ и поступает запрос найти наибольший общий префикс у суффиксов aba и $acabadabacaba$.

Красный столбик – значения lcp , зелёными полосками помечено для наглядности. Чтобы ответить на запрос, находим минимум на жёлтом отрезке. Он жирный.



1

aba

3

abacaba

7

abacabadabacaba

3

abadabacaba

1

acaba

5

acabadabacaba

1

adabacaba

0

ba

2

bacaba

6

bacabadabacaba

2

badabacaba

0

caba

4

cabadabacaba

0

dabacaba

индекс	lcp
0	1
1	3
2	7
3	3
4	1
5	5
6	1
7	0
8	2
9	6
10	2
11	0
12	4
13	0

Классическая задача на $I_{\text{ср}}$

Задача: Задана строка S . Требуется посчитать количество различных подстрок строки S .

Как решать?

1. Построим суфмас и посчитаем $|sr$.
2. Последовательно пройдемся по суффиксному массиву. Сколько новых уникальных подстрок добавит очередной суффикс?
 - Если это первый суффикс в суфмассе, то он добавит размер этого суффикса уникальных подстрок.
 - Иначе, размер этого суффикса минус $|sr$ текущего суффикса с предыдущим.

Пример

1	a
3	aba
7	abacaba
3	abacabadabacaba
1	abadabacaba
5	acaba
1	acabadabacaba
0	adabacaba
2	ba
6	bacaba
2	bacabadabacaba
0	badabacaba
4	caba
0	cabadabacaba
	dabacaba

1. Суффикс 'а' добавляет одну уникальную подстроку.
 2. Суффикс 'aba' добавляет 2 уникальных подстроки('ab', 'aba'), т.к. подстрока 'а' уже находится в множестве уникальных подстрок.
 3. Суффикс 'abacaba' добавит подстроки 'abac' 'abaca' 'abacab', 'abacaba'.
- и т.д.

Почему это работает?

1. Рассматриваем суффиксы в отсортированном порядке.
2. Рассмотрели все позиции, с которых могут начинаться подстроки.
3. Добавили только те, которые не встречались до этого.



БЕЛОРУССКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

Спасибо за внимание!