Российская академия наук
Национальный комитет по автоматическому управлению
Научный совет по теории управляемых процессов и автоматизации
ОЭММПУ РАН
Институт проблем управления им. В.А. Трапезникова РАН
Министерство образования и науки Удмуртской Республики
Удмуртский государственный университет
Удмуртский НОЦ ПУ ( на базе УдГУ)
Волгоградский НОЦ ПУ ( на базе ВолГУ)
Воронежский НОЦ ПУ ( на базе ВГАСУ)
Инновационный НОЦ ПУ ( на базе МАИ)
Казанский НОЦ ПУ ( на базе КГТУ-КАИ)
Курский НОЦ ПУ (на базе КГТУ)
Липецкий НОЦ ПУ ( на базе ЛГТУ)
НОЦ «Системный анализ в управлении» (на базе МИФИ)
Пермский НОЦ ПУ ( на базе ПГТУ)
Самарский НОЦ ПУ ( на базе СГАУ)
Тверской НОЦ ПУ ( на базе ТГТУ)
Старооскольский НОЦ ПУ ( на базе СТИ)
Удмуртский Совет ИТ-директоров
Институт логики, когнитологии и развития личности РАН

# VI Всероссийская школа-семинар молодых ученых

# Управление большими системами

*посвящается памяти А.А. Маркова*

## Том 1

31 августа – 5 сентября 2009
г. Ижевск

УДК 007

VI Всероссийская школа-семинар молодых ученых «Управление большими системами»: Сборник трудов. – Т1.- Ижевск: ООО Информационно-издательский центр «Бон Анца», 2009. – 400 с.

В первый том сборника трудов включены научные статьи молодых ученых по фундаментальным основам теории управления, вопросам управления организационными и социально-экономическими системами, управлению качеством.

**ISBN 978-5-903140-57-2**

Научное издание осуществлено при поддержке РФФИ
грант № 09-07-06039г

© Авторы, постатейно, 2009
©ООО ИИЦ «Бон Анца»
(оформление обложки, верстка)

УДК 004.042

# Блочное построение суффиксного массива

*Айткулов П.Г.*
*Удмуртский государственный университет, г. Ижевск*

**Аннотация**

В статье строится алгоритм блочного построения суффиксного массива с асимптотикой $O(n \log^2(n))$. К существующему суффиксному массиву добавляется минимальная строка, не входящая в текущий суффиксный массив. Как побочное действие, на каждом шаге поддерживается *lcp*.

**Ключевые слова**: сопоставление строк, суффиксный массив, динамическое построение суффиксного массива, строковые алгоритмы.

**Abstract**

An algorithm for incremental constructing suffix array in $O(n \log^2(n))$ in worst case is described here. This algorithm adds a string (a minimal string which doesn't contained in suffix array) to the existing suffix array and updates supplementary data structures. As side effects on every iteration we have lcp-array (longest common prefix).

**Keywords:** string matching, suffix array, dynamic construction suffix array, string algorithm.

## Введение

Суффиксный массив строки определяется как перестановка, выражающая лексикографический порядок всех суффиксов строки. Суффиксный массив представляет собой компактное представление суффиксного дерева. Суффиксные массивы имеют массу приложений в задачах поиска образца в строке, биоинформатике [10], сжатии данных [4]. Имеются алгоритмы построения суффиксного массива за $O(n)$ [3], распределенные алгоритмы [6], [7].

Для существующих алгоритмов построения суффиксных массивов требуется наличие всей входной строки. Это ограничивает использование суффиксных массивов в задачах с потоковыми данными.

В статье построен алгоритм блочного построения суффиксного массива с асимптотикой $O(n \log^2(n))$ для наихудшего случая, приспособленный для применения к частично заданной строке.

## 1. Концепция приложения

Пусть дана строка $s$. Суффиксный массив строки $s$ обозначим как $sa$. К строке $s$ мы добавляем новый блок $ns$. Необходимо получить суффиксный массив для объединенной строки $s$ ++ $ns$. Здесь и далее символ ++ означает конкатенацию строк, а символ + будет использоваться только

в арифметическом смысле. Введем обозначение $n = |s|$, $k = |ns|$, где $|s|$ – длина строки $s$. Запись $s[i..j]$ обозначает подстроку $s$, начинающуюся с символа с индексом $i$ по символ с индексом $j$. Запись $s[i..]$ является сокращением для суффикса $s[i..|s|]$.

## 2. Построение алгоритма

В качестве $ns$ выберем блок такого размера, чтобы строка $ns$ была минимальной строкой, еще не входящей в текущий суффиксный массив. Такой выбор блока $ns$ важен, объяснение выбора будет изложено ниже.

Рассмотрим, что происходит при добавлении строки $ns$ к существующему суффиксному массиву $sa$ строки $s$. Ко всем суффиксам приписываем строку $ns$. Некоторые суффиксы могут изменить свое положение относительно существующих. Необходимо также добавить в суффиксный массив $sa$ все суффиксы строки $ns$.

Шаг 1. Ко всем суффиксам приписать строку ns.
Поддерживать отсортированный порядок в суффиксном массиве.
Шаг 2. В суффиксный массив добавить все суффиксы строки ns.

При построении алгоритма нам потребуется такая дополнительная структура данных, как массив наибольшего общего префикса (lcp, longest common prefix). Значением $i$-го элемента массива $lcp$ определяется как длина общего префикса между суффиксами $sa_i$ и $sa_{i+1}$.

Наша задача состоит в реализации алгоритма со сложностью меньше квадратичной. Очевидная реализация первой части алгоритма (прямое приписывание строки $ns$, попарное сравнение суффиксов, сравнение двух суффиксов за линейное время, перемещение одного суффикса за линейное время) не удовлетворяет требованиям сложности. Покажем, как можно эффективно реализовать эту часть алгоритма.

**Лемма 1.**

*Пусть дан суффиксный массив sa. При приписывании новой строки ns ко всем суффиксам суффиксного массива sa свое местоположение могут поменять только суффиксы, для которых выполнено* $lcp_i = |sa_i|$.

**Доказательство.**

Утверждение $lcp_i < |sa_i|$ означает, что $lcp_i < |sa_{i+1}|$; поэтому добавление строки $ns$ не изменит лексикографический порядок суффиксов $sa_i$ и $sa_{i+1}$.

Утверждение $lcp_i = |sa_i|$ также означает, что $lcp_i < |sa_{i+1}|$, и весь суффикс $sa_i$ является префиксом для суффикса $sa_{i+1}$. В зависимости от добавляемой строки $ns$ можно получить: $sa_i + +ns > sa_{i+1} + +ns$, $sa_i + +ns < sa_{i+1} + +ns$ и $sa_i + +ns = sa_{i+1}[1..|sa_i| + |ns|]$ (в зависимости от отношения $ns$ и $sa_{i+1}[|sa_i| + 1..]$). ■

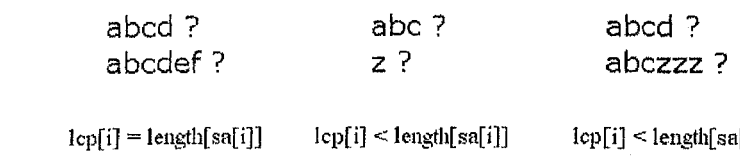


**Рис. 1. Добавление новой строки изменяет положение суффиксов, только если $lcp_i = |sa_i|$**

Из леммы следует, что для работы первой части алгоритма достаточно обработать суффиксы, для которых верно $lcp_i = |sa_i|$. Будем хранить эту информацию в массиве *bo* (от *bool* (булевский) и *boundary* (граница)).

Покажем, как определить отношение порядка для суффикса относительно следующего, для которого верно, что $lcp_i = |sa_i|$, после того как приписать к обоим суффиксам строку *ns*.

Вместо сравнения за линейное время предложим следующее. Сравнение можно разбить на следующие две части: сравнение между $ns[1..|sa_{i+1}| - |sa_i|]$ и $sa_{i+1}\,[|sa_i| + 1..]$, а также $ns[|sa_{i+1}| - |sa_i| + 1..]$ и *ns*.

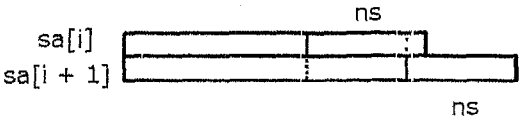


**Рис. 2. Сравнение суффиксов разбивается на два сравнения**

Первое сравнение $ns[1..|sa_{i+1}| - |sa_i|]$ и $sa_{i+1}\,[|sa_i| + 1..]$ (обозначим как $\delta_{i,i+1}$) есть лексикографическое сравнение префикса *ns* и $\delta_{i,i+1}$. Так как во второй части алгоритма потребуется вставлять все суффиксы строки *ns* в суффиксный массив (и саму строку *ns* тоже), то вставим в суффиксный массив строку *ns* (так же поддерживаем значение *lcp*-массива). Тогда указанное сравнение сводится к поиску местоположения суффикса $\delta_{i,i+1}$ и сравнению местоположения *ns*.

Если $\delta_{i,i+1}$ совпадает с префиксом *ns*, то необходимо выполнить оставшуюся проверку $ns[|sa_{i+1}| - |sa_i| + 1..]$ и *ns*, что равносильно задаче о сравнении строки с собственным суффиксом. Такое сравнение потребуется для различных суффиксов строки *ns*, поэтому построим для строки *ns* суффиксный массив за линейное время [3]. Тогда задача сводится к сравнению местоположений суффикса строки *ns* и самой строки *ns* в суффиксном массиве для строки *ns*.

То есть мы можем сравнивать два суффикса без полного сравнения строк. Точные оценки алгоритмической сложности приводятся в последующих параграфах.

Если $sa_i + +ns > sa_{i+1} + +ns$, то необходимо переместить суффикс $sa_i + +ns$ на новое место. Для определения нового местоположения суффикса $sa_i + +ns$ воспользуемся следующей леммой.

**Лемма 2.**

*Пусть дан суффиксный массив sa, массив lcp для суффиксного массива sa. К суффиксу $sa_i$ приписали новую строку ns, так что $sa_i + +ns > sa_{i+1} + +ns$ и $lcp(sa_i + +ns, sa_{i+1} + +ns) = l$. Тогда для поддержания корректности суффиксного массива sa и структуры lcp необходимо произвести следующие действия.*

- при удалении суффикса $sa_i + +ns$ обновить значение $lcp(sa_{i-1}) \leftarrow \leftarrow min(lcp(sa_{i-1}))$,
- при вставке $sa_i + +ns$ на новое место позиция определяется как $np = min\{k | k > i \wedge lcp(sa_i + +ns, k) < l\}$.
- модифицировать $lcp[np] \leftarrow lcp[np - 1]$, $lcp[np - 1] \leftarrow l$.

**Доказательство.**

Согласно свойствам *lcp* ([8], [5]) имеет место $lcp_{i,j} = min_{k \in [i \ldots j-1]} lcp_k$.

Откуда следует изменение значение элемента *lcp* при удалении суффикса.

Исходя из определения *lcp* определяется место для вставки суффикса (ближайшая позиция *k*, для которой $lcp_k < l$. На отрезке $[i + 1, k]$ вставить суффикс $sa_i + +ns$ нельзя, так как на этом отрезке имеется общий префикс). ■

| lcp | suffixes | lcp | suffixes |
|---|---|---|---|
| | 1 a | | 1 a |
| | 3 ana | | 3 ananaz |
| | 0 anana | | 0 anaz |
| | 0 banana | | 0 banana |
| | 2 na | | 2 na |
| | - nana | | - nana |

**Рис. 3. Ко второму и третьему суффиксу добавили символ «z»**

Для того чтобы воспользоваться Леммой 2, необходимо получить значение $lcp(sa_i + +ns)$. Это значение получим на этапе сравнения суффиксов. При первой части сравнения (префикс *ns* и некоторый суффикс *sa*) значение *lcp* получается при поиске в текущем суффиксном массиве *sa*. Если потребуется вторая часть сравнения (некоторый суффикс строки *ns* и сама строка *ns*), то значение *lcp* определяется из суффиксного массива для строки *ns* (*lcp* строится за линейное время от длины строки [5]).

Перейдем ко второй части алгоритма, вставке в суффиксный массив всех суффиксов добавляемой строки *ns*. Очевидная реализация (для каждого суффикса строки *ns* найти местоположение в текущий суффиксный массив за $O(k + log(n))$ [8]) данной части алгоритма приводит к квадратичной сложности.

Строка *ns* уже была добавлена в суффиксный массив на предыдущем шаге. Осталось добавить все собственные суффиксы строки *ns*. Напомним, что мы выбрали *ns* так, чтобы строка была минимальной, еще не входящей в суффиксный массив *sa*. Это означает, что строка $ns[1..|ns|-1]$ присутствует в суффиксном массиве. Так же это означает, что для всех суффиксов строки *ns* после вставки в суффиксный массив будет выполнено $lcp(ns_{suffix}) = |ns_{suffix}| \vee lcp(ns_{suffix}) = |ns_{suffix}| - 1$ (так как за исключением последнего символа подсуффикс уже содержится в суффиксном массиве).

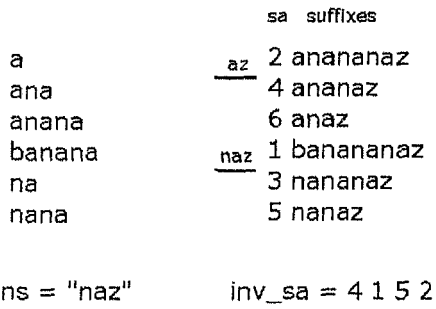


**Рис. 4. К суффиксному массиву для слова «*banana*» добавляем строку «*naz*»**

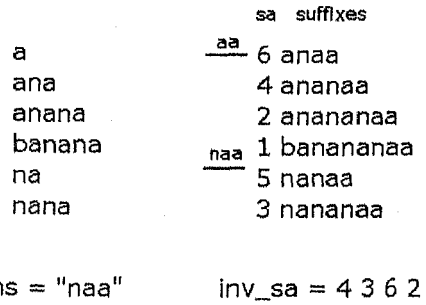


**Рис. 5. К суффиксному массиву для слова «*banana*» добавляем строку «*naa*»**

Найдем положение суффикса, префикс которого совпадает с $ns[2..|ns|-1]$. Если таких суффиксов несколько, то выберем лексикографически наименьший. Обозначим местоположение такого суффикса как $pos_{ns.2}$. Тогда вставлять $ns[2..]$ надо перед позицией $pos_{ns.2}$ (в зависимости от последнего символа $ns$, возможно $ns[2..]$ надо будет переместить по аналогии с перемещением суффикса в первой части алгоритма). Для перехода к $ns[3..]$ можно не искать местоположение как на предыдущей итерации. Достаточно перейти к суффиксу, чья длина на 1 меньше суффикса $sa_{pos_{ns.2}}$, и произвести аналогичные операции.

Для перехода к суффиксам нужной длины будем поддерживать массив (обозначим как $inv_{sa}$), являющийся обратной перестановкой к суффиксному массиву $sa$ (при любой операции вставки, удаления из $sa$ производим аналогичную над $inv_{sa}$).

Теперь осталось только добавить в суффиксный массив еще один суффикс, а именно $ns[|ns|]$.

## 2.1. Описание алгоритма

```
1   procedure Preprocessing
2   Создание lcp.
3   Создание bo.
4   Создание RMQ.
5   Создание inv_sa.
    end procedure
    procedure AddString(string ns)
6   Построение суффиксного массива для ns (sa_ns)
7   Вставка в суффиксный массив sa строку ns.
8   Обновление lcp.
9   Для всех суффиксов sa_i, помеченных bo_i
10      Сравнение sa_i + +ns и sa_{i+1} + + ns.
11      Вычисление lcp(sa_i + +ns, sa_{i+1} + +ns).
12      Если sa_i + +ns > sa_{i+1} + +ns, то
13          перемещение sa_i + +ns, обновление значения lcp
14      иначе
15          обновление значений lcp.
16  Очистка массива bo
17  Поиск местоположения суффикса, префикс которого
        совпадает с ns[2..|ns| - 1]
18  For i in |ns| - 1 downTo 2 do
19      Вставка суффикса ns[|ns| - i + 1..]
```

```
20          Обновление lcp
21          Обновление bo
22      Вставка ns[|ns|]
23      Обновление lcp
24      Обновление bo
        end procedure
```

## 2.2. Структуры данных

Заметим, что в алгоритме используются операции вставки и удаления элементов из линейной структуры. Операции вставки и удаления в линейном массиве или списке требуют $O(n)$ операций. Будем использовать структуру данных (основанную, например, на красно-черном дереве или AVL-дереве [11]) с доступом к элементу и операцией вставки, удаления за время $O(log(n))$.

Для вычисления $lcp(i, j)$ найдем минимальное значение на отрезке $lcp_k$, $k \in [i, j-1]$, [8]. На массиве постоянного размера за время препроцессинга $O(n\ log(n))$ и время запроса $O(log(n))$ можно решить задачу при помощи дерева отрезков [12]. Использование дерева отрезков также позволяет решать расширенную задачу с возможностью модификации значений элементов массива (время препроцессинга и запроса прежнее, время на модификацию значения равно $O(log(n))$).

Так как нам необходимо производить операции вставки и перемещения элемента, то дерево отрезков использовать нельзя. В качестве решения воспользуемся [1]. В работе используется структура данных, основанная на сбалансированном дереве, позволяющая вставлять, перемещать, модифицировать элемент и искать минимум на отрезке за $O(log(n))$ операций.

## 2.3. Структуры данных

Оценим алгоритмическую сложность построенного алгоритма.

На строке 6 (построение суффиксного массива для строки *ns*) потребуется $O(k)$ операций [5].

На строке 7 (вставка в суффиксный массив *sa* строки *ns)* потребуется $O(k + log(n))$ на вставку [8]. Строка 8 затратит $O(n)$ операций (линейный подсчет *lcp* для вставленного и предыдущего элемента. Можно быстрее, но ускорение на данном шаге не улучшит общую асимптотику алгоритма).

Сравнение и вычисление *lcp* (10 11 строки) разбивается на два подсравнения. Оба сравнения вычисление *lcp* происходит за $O(log(n))$ (сравнение индексов в соответствующих суффиксных массивах, вычисление *lcp* как запрос к структуре *RMQ*) [1, 8].

В случае перемещения суффикса $sa_i$ (строка 13) для поиска нового положения необходимо вычислить $min\{k|k > i \wedge lcp\ (sa_i + +ns,\ k) < lcp\}$. Применим бинарный поиск для поиска минимума на отрезке $[i + 1, x]$ по аргументу $x$. Тогда поиск нового положения суффикса займет $log^2(n)$ (бинарный поиск и вычисление *lcp)*.

Обновление значений *lcp* на строках 13, 15 занимает $O(1)$ (Лемма 2).

Так как число суффиксов, помеченных *bo*, может достигать $O(n)$, то блок 9–15 строки выполняется за $O(n\ log^2(n))$.

Так как *ns* является строкой, не входящей в качестве подстроки в *s*, то после приписывания ко всем суффиксам новой строки *ns* не будет суффикса, для которого верно $|\ sa_i\ | = lcp_i$. На строке 16 очистим массив *bo* за $O(1)$.

Вставка всех суффиксов строки *ns* занимает $O(k\ log^2(n))$. Так как поиск первоначального места (строка 17) требует $O(k + log(n))$ ([8]), тело цикла (строки 19-21) выполняется за $O(log^2(n))$, а количество итераций цикла составляет $O(k)$. Вставка последнего символа строки *ns* занимает $O(log(n))$ операций.

Сложность всего алгоритма составляет $O(n\ log^2(n))$.

## 2.4. Препроцессинг

Алгоритм можно использовать для модификации суффиксных массивов, уже построенных другими алгоритмами. Для этого на шаге препроцессинга (строки 1 5) достаточно создать все необходимые дополнительные структуры данных.

Оценим алгоритмическую сложность шага препроцессинга.

Массив *lcp* строится за $O(n)$ [5].

Массив *bo* также строится за $O(n)$, линейный проход по всем суффиксам $bo_i \leftarrow (lep_i == length(sa_i))$. Заметим, что число помеченных суффиксов может составлять $O(n)$.

Структура *RMQ* нам требуется при основной работе алгоритма. Построение занимает $O(n\ log(n))$ операций [1]. При каждом изменении суффиксного массива *sa* (вставка или удаление элемента) или массива *lcp* (изменение значения) необходимо модифицировать структуру *RMQ*. Это требует $O(log(n))$ операций [1].

Обратная перестановка для суффиксного массива $inv_{sa}$ строится за линейное время. При каждом изменении суффиксного массива (вставка или удаление элемента) производим аналогичное действие над $inv_{sa}$.

Общая асимптотика препроцессинга составляет $O(n \log(n))$ операций.

### 2.5. Оценка использование памяти

В процессе работы алгоритма используются только структуры данных с объемом памяти $O(n)$ (*lcp*, *bo*, $inv_{sa}$, *RMQ*, суффиксный массив, [1, 8, 11]).

## 3. Недостатки алгоритма

Для добавления одного символа может потребоваться $O(n \log^2(n))$ операций (в случае, если число суффиксов, помеченных *bo*, составляет $O(n)$). То есть использование линейного алгоритма построения суффиксного массива (например [3]) заново для всей строки асимптотически окажется лучше, чем модификация существующего суффиксного массива алгоритмом блочного построения суффиксного массива.

Этот недостаток не устраним, так как, например, добавление символа *«b»* к строке *«aa..a»* «переворачивает» суффиксный массив, что потребует не меньше чем $O(n)$ операций.

## 4. Преимущества алгоритма

Для работы алгоритма необязательно иметь исходную строку целиком. Это позволяет использовать предложенный алгоритм в задачах с потоковыми данными (например преобразование Барроуза Уилера [2] в базах данных).

В отличие от [9], мы добавляем к суффиксному массиву не один символ, а строку. Это позволяет уменьшить асимптотику от квадратичной до $O(n \log^2(n))$ для наихудшего случая.

Для суффиксного массива (созданный любым другим алгоритмом) можно за время $O(n \log(n))$ построить дополнительные структуры данных *lcp*, *bool*, *RMQ* [5], [1] и продолжить его построение предложенным методом.

На каждой итерации алгоритма поддерживается массив *lcp*. Это можно использовать, например, для поиска наибольшей подстроки. Достаточно хранить значения массива *lcp* в структуре данных для поддержки максимального значения (например двоичная куча, сбалансированное дерево [11])

## Заключение

В статье построен алгоритм блочного построения суффисного массива с временем работы $O(n \log^2(n))$ и линейным размером памяти. Алгоритм можно использовать для потоковых данных.

## Список литературы

1. *Benson Gary, Roderic D.M. Page.* Algorithms in Bioinformatics. Third International Workshop, WABI 2003, Budapest. (T. Shibuya, I. Kurochkin. Match chaining algorithm for cDNA Mapping).

2. *Burrows M., Wheeler D.* A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994.

3. *Karkkainen Juha and Sanders Peter.* Simple Linear Work Suffix Array Construction, Proceedings of the 30th International Colloquium on Automata, Languages and Programming, LNCS 2719, Springer. P. 943 955, 2003.

4. *Karkkainen Juha.* Fast BWT in Small Space by Blockwise Suffix Sorting. Theoretical Computer Science, 2006.

5. *Kasai T., Lee G., Arimura H., Arikawa S., Park K.* Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. CPM 2001: 181 192.

6. *Kitajima Joao Paulo, Gonzalo Navarro, Berthier Ribeiro and Nivio Ziviani.* Distributed Generation of Suffix Arrays: a Quicksort-Based Approach, Proceedings of the 4th South American Workshop on String Processing, Valparaiso, Chile. P. 53 59, 1997.

7. *Navarro Gonzalo, Kitajima Joao Paulo, Ribeiro Berthier and Ziviani Nivio.* Distributed Generation of Suffix Arrays, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, LNCS 1264. P. 102 115, 1997.

8. *Manber U., Mayers G.* Suffix arrays: a new method for on-line string searches // SIAM Journal on Computing 22, 953 948, 1993.

9. *Айткулов П.* Последовательное построение суффиксного массива // Удмуртский вестник (неопубликовано).

10. *Гасфилд Д.* Строки, деревья, последовательности в алгоритмах. СПб.: Невский Диалект; БХВ Петербург, 2003.

11. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999.

12. *Препарата Ф., Шеймос М.* Вычислительная геометрия: введение. М.: Мир, 1989.