УДК 004.042
ББК 32.973.26-018.2

ОБРАБОТКА СИМВОЛЬНЫХ МАССИВОВ

Айткулов П. Г. [1]
(*Удмуртский государственный университет, Ижевск*)

Суффиксный массив для строки представляет собой структуру данных, которая позволяет искать все вхождения образца за линейное время от длины образца. Построены алгоритмы модификации суффиксного массива при добавлении одного символа, при добавлении блока к исходной строке и удалении блока из строки. Найдено применение построенных алгоритмов к индексации текстовых записей в базах данных и имен файлов в файловой системе. Построен алгоритм поиска наибольшей общей подстроки для k-строк для динамического случая.

Ключевые слова: алгоритмы на строках, суффиксный массив, наибольшая общая подстрока.

Введение

Работа в текстовом редакторе, поисковые запросы в базе данных, задачи в биоинформатике, лексический анализ программ требуют эффективных алгоритмов работы со строками.

Задачи поиска образца в тексте используются в криптографии, различных разделах физики, сжатии данных, распознавании речи.

В конце 1970-х годов на стыке генетики и информатики появилась биоинформатика (или вычислительная биология). Длина генотипа человека составляет 3,2 миллиарда символов (нуклеотидов). Для обработки таких больших данных требуются эффективные алгоритмы вычислений на строках.

[1] *Павел Григорьевич Айткулов, аспирант* (*ajtkulov@gmail.com*).

Существует два подхода в алгоритмах поиска образца: преобразование образца и суффиксные структуры данных.

В первом подходе образец является статичным, а исходный текст динамичен. Для каждого поискового запроса требуется прочитать исходный текст заново.

Если исходный текст является статичным, то стоит воспользоваться суффиксными структурами данных. Поисковый запрос к таким структурам требует линейных от длины образца ресурсов (количество операций и память).

К недостаткам существующих алгоритмов построения суффиксных структур данных относится то, что для построения структуры требуется вся строка целиком. Это ограничивает использование суффиксных структур данных с потоковыми данными. К настоящему времени нет алгоритмов модификации суффиксных структур данных при изменении исходной строки.

В работе построены алгоритмы модификации такой суффиксной структуры данных, как суффиксный массив. Построено практическое применение для индексации строковых полей в базе данных и имен файлов в файловой системе. Построенные алгоритмы позволяют решать задачу о наибольшей общей подстроке для динамического случая для одной, двух и k-строк.

1. Обзор

БАЗОВЫЕ ОПРЕДЕЛЕНИЯ

Алфавит Σ — конечное множество символов.

Строка s — это упорядоченный список символов. Рассматриваются конечные строки.

Подстрока $s[i..j]$ — строка, начинающаяся в позиции i и заканчивающаяся в позиции j строки s.

Префиксом строки s называется подстрока $s[1..i]$ для некоторого i.

Суффиксом строки s называется подстрока $s[i..|s|]$ для некоторого i.

1.1. АЛГОРИТМЫ РАБОТЫ СО СТРОКАМИ

В работе рассматриваются в основном задачи точного поиска подстроки. В противоположность таким задачам есть задачи нечеткого поиска, где допускается некоторое число ошибок при сравнении строк. Также существуют задачи нахождения минимального редакционного расстояния, где требуется найти минимальное число изменений (замена, вставка и удаление символа) необходимых для преобразования из одной строки в другую.

В этой главе число n обозначает длину исходного текста s, m — длина образца t.

1.1.1. BRUTE-FORCE

Самый простой и очевидный алгоритм. Пробуем сопоставить образец с позиции 1. Если сопоставление прошло неудачно, то пытаемся сопоставить образец с позиции на единицу больше предыдущего. Асимптотика наихудшего случая – $O(nm)$ (пример, $s =' aa..aa'$, $t =' aa..ab'$). Затраты по памяти – $O(1)$.

1.1.2. ПОСТРОЕНИЕ КОНЕЧНОГО АВТОМАТА

Строится детерминированный конечный автомат (распознающий образец t) за $O(m)$ операций. Далее за $O(n)$ проверяем, достигает ли автомат заключительного состояния. Итоговая (для наихудшего и наилучшего случая) асимптотика – $O(n+m)$. Замечание: если учитывать размерность алфавита Σ, то на построение автомата потребуется $O(m|\Sigma|)$ операций, на хранение – $O(m|\Sigma|$ памяти. Общая асимптотика составит $O(n + m|\Sigma|)$.

1.1.3. АЛГОРИТМ КНУТА-МОРРИСА-ПРАТТА

В 1977 году была опубликована работа [24][2], где устранялись недостатки поиска при помощи конечного автомата (зависимости от мощности алфавита Σ).

[2] *Алгоритм был открыт Кнутом, Праттом и, независимо от них, Моррисом.*

Вместо построения детерминированного конечного автомата строится префикс-функция π для образца t. Значение префикс-функции $\pi(q)$ определяет длину наибольшего префикса t, являющегося собственным суффиксом $t[1..q]$. Префикс-функция π по образцу t строится на линейное время $O(m)$, и с точки зрения алгоритма поиска с использованием конечного автомата, новое состояние автомата определяется при помощи префикс функции за амортизированную $O(1)$.

Общее время работы алгоритма составляет $O(n + m)$ при линейных затратах памяти $O(m)$ (где, в отличие от алгоритма поиска с конечным автоматом, сложность алгоритма и затраты на память точно не зависят от размерности алфавита).

1.1.4. АЛГОРИТМ РАБИНА-КАРПА

Идея этого алгоритма состоит в вычислении хеш-функции от образца и вычислений хеш-функции от части сопоставляемой исходной строки. Если значения хеш-функций не равны, то сопоставления явно будет неудачным, при равенстве значений необходимо линейное сопоставление. Предлагается использовать элементы динамического программирования. Например, в качестве хеш-функции можно выбрать сумму символов. Тогда для перехода к следующему символу из текущего значения хеш-функции необходимо вычесть первый символ и добавить новый. Для такой хеш-функции строки состоящие из одних символов получат одинаковый результат ($f('ab') = f('ba')$).

Рабин и Карп предложили в качестве хеш-функции использовать

$$a_1 \cdot 2^{m-1} + a_2 \cdot 2^{m-1} + \ldots + a_m$$

в кольце вычетов. Тогда переход к следующему символу состоит в

$$f_{new} = 2(f_{old} - a_1 \cdot 2^{m-1}) + a_{m+1} (mod\ p)$$

(вычитании из предыдущего значения хеш-функции первого элемента, сдвиг вправо, добавление нового элемента). Переход к следующему символу осуществляется за $O(1)$). Такая функция имеет

бо́льшую область значений. Асимптотика для среднего случая – $O(n+m)$.

Данный метод также используется для поиска образцов в двумерной карте.

1.1.5. АЛГОРИТМ БОЙЕРА-МУРА

В 1977 году Бойер и Мур в совместной работе [13] опубликовали наиболее эффективный в настоящее время алгоритм поиска строки.

В алгоритме используется две эвристики (стоп-символ, безопасный суффикс), каждая из которых, примененная в отдельности, не дает нужной асимптотики. При сопоставлении строки сравниваются с конца.

Асимптотика наилучшего случая является сублинейной ($O(n/m)$), для наихудщего случая – линейная $O(n+m)$.

Доказательство для асимптотики для наихудшего случая имеет запутанную историю ([3], препроцессинг для сильного правила изложен в [24] некорректно и исправлен в работе [29] без пояснений). В работе Баасе [12], основанной на [29], приведен код без доступного объяснения. Печатных источников, где бы пытались объяснить метод, к настоящему времени не существует ([3]).

1.2. АХО-КОРАСИКА

Производится поиск вхождения k строк. Вместо поиска вхождения для каждой строки $O(kf(n,m))$ (где $f(n,m)$ — сложность поиска образца размером m в тексте размером n) строится конечный детерминированный автомат за $O(km)$ ([11]). Поиск осуществляется за $O(n)$.

Стоит различать задачу поиска первого вхождения, поиска всех вхождений, поиска всех вхождений без пересечений. Различие между второй и третьей задачей можно показать на примере. Для строки $'abababababa'$ строка $'aba'$ входит в 1, 3, 5, 7 позициях, но в задаче поиска всех вхождений без пересечений правильным ответом считается вхождение в 1, 5 позициях (так как вхождение в 1 и 3 позициях пересекается в позиции 3).

Все описанные алгоритмы объединяет то, что мы преобразуем образец поиска. При этом исходный текст остается без преобразований. Это позволяет заранее произвести вычисления для образца и искать его в любом тексте.

Во многих задачах текст является статичным (например, биоинформатика), а образец от запроса к запросу изменяется (поиск различных белков в ДНК).

1.3. ОБЗОР СУФФИКСНЫХ СТРУКТУР ДАННЫХ

1.3.1. БОР

Бор (trie, луч, нагруженное дерево [25]) представляет собой дерево, хранящее все суффиксы строки (каждый путь в дереве представляет собой некоторый суффикс, длина пути в точности равна длине суффикса, каждая вершина хранит символ).

Хотя время поиска подстроки является линейным, для построения бора требуется $O(n^2)$ операций, а для хранения – $O(n^2)$ памяти, что сильно ограничивает его практическое использование.

1.3.2. СУФФИКСНЫЙ АВТОМАТ

Суффиксный автомат строки s строится за линейное время [14] и допускает все суффиксы s и только их. Эта структура данных не очень распространена, так как практическое применение весьма ограничено.

1.3.3. СУФФИКСНОЕ ДЕРЕВО

Еще в [25] было замечены недостатки бора и возможность хранения с использованием $O(n)$ памяти и поиска за линейное время на производной от бора структуре данных.

В 1973 и 1976 годах Вайнером и МакКрейгом годах были созданы два алгоритма построения суффиксного дерева за линейное время [27, 33]. Эти алгоритms довольно трудны для понимания и в литературе рассматриваются лишь вскользь.

В 1995 году Эско Укконен ([32]) изобретает *online*-алгоритм построения суффиксного дерева. Это означает, что добавление одного символа требует амортизированную $O(1)$ для изменения текущего суффиксного дерева.

1.3.4. СУФФИКСНЫЙ МАССИВ

Суффиксный массив строки определяется как перестановка, выражающая лексикографический порядок всех суффиксов строки. Элемент перестановки с индексом i выражает индекс суффикса, который будет i-ым в лексикографическом порядке среди всех суффиксов. Например, первый элемент суффиксного массива показывает индекс лексикографически наименьшего суффикса.

Суффиксный массив представляет собой компактное представление суффиксного дерева. Суффиксные массивы имеют массу приложений в задачах поиска образца в строке, биоинформатике [3], сжатии данных [20] Имеются алгоритмы построения суффиксного массива за $O(n)$ [19], распределенные алгоритмы, [28].

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

1.4. ОБЗОР АЛГОРИТМОВ ПОСТРОЕНИЯ СУФФИКСНЫХ МАССИВОВ

До начала 1990-х годов суффиксный массив строился из суффиксного дерева посредством обхода в глубину. Это не имело практического смысла, так как затраты памяти на построение суффиксного дерева значительно больше затрат для хранения суффиксных массивов, и область задач решаемых при помощи суффиксных деревьев больше чем у суффиксных массивов. То есть преобразование из суффиксного дерева в суффиксный массив необходимо только для экономии памяти для более узкого класса задач.

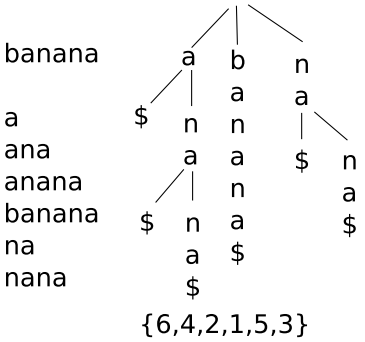

Рис. 1. Создание суффиксного массива из суффиксного дерева при помощи обхода в глубину

В 1993 году Манбер и Майерс в работе [26] построили алгоритм построения суффиксного массива за $O(n\,log(n))$ операций. Предложенный метод (техника удвоения) используется в нескольких модификациях алгоритма Манбера и Майераса.

1.4.1. ТЕХНИКА УДВОЕНИЯ

Пусть все суффиксы отсортированы по первым k символам (для начала работы алгоритма достаточно отсортировать по первому символу при помощи цифровой сортировки). Суффиксы, которые совпадают по первым k символам, принадлежат одному фактор-классу[3]. Нас интересуют фактор-классы, состоящие из более чем одного элемента (если таковых не имеется, значит построение суффиксного массива закончено).

Возмем два суффикса s_1 и s_2 из одного фактор-класса. Первые k символов этих суффиксов совпадают. Рассмотрим следующие k символов в этих суффиксах. Это тоже некоторые суффиксы исходной строки $s_1^{'}$ и $s_2^{'}$. Определяем, принадлежат ли эти суффиксы $s_1^{'}$ и $s_2^{'}$ одному фактор-классу. Если ответ отрицателен, то суффиксы s_1 и s_2 текущего фактор-класса разбиваются на более

[3] *В работе [26] используется термин «корзина».*

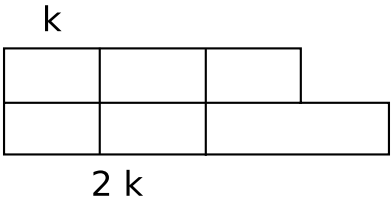

Рис. 2. Техника удвоения при создании суффиксного массива

мелкие классы. Иначе s_1 и s_2 обладают общим префиксом из $2k$ символов. Рассмотрев все фактор-классы из более чем одного элемента, получим суффиксный массив, элементы которого отсортированы по первым $2k$ символам.

1.4.2. ДРУГИЕ МЕТОДЫ

Имеются работы по распределенному и параллельному построению суффиксных массивов [28].

Также есть возможность построить суффиксное дерево по суффиксному массиву [15, 16, 17].

1.5. АНАЛОГИЯ С ОТСОРТИРОВАННОЙ КОЛЛЕКЦИЕЙ

Предположим, вы пришли в бухгалтерию за некоторой справкой. Справка уже напечатана и лежит в общей куче. Необходимо найти нужную справку.

Если стопка справок не отсортирована по фамилии, то чтобы найти нужную, необходимо просмотреть всю стопку или затратить $O(n)$ операций. Если же справки отсортированы по фамилии, то поиск можно осуществить за $O(log(n))$ операций при помощи бинарного поиска[4]. Также можно легко подсчитать количество справок с фамилией с первой буквой «Н» за $O(log(n))$ операций. Для суффиксных массивов это означает поиск количества и всех вхождений заданной подстроки в тексте за $O(log(n))$.

[4] *Алгоритмы хеширования не рассматриваются*

Проблемы начинаются при добавлении новых объектов. Найти место для вставки можно за $O(log(n))$ операций, но сама операция вставки для линейного массива займет $O(n)$ (для небольшой стопки бумаг вставка требует константного времени. В памяти раздвинуть элементы просто так нельзя. Приведем аналогию с сортировкой тяжелых металлических пластин: чтение является простой задачей, а перемещение объекта – тяжелой в прямом и переносном смысле).

Для чисел (и других не очень сложных объектов) имеются способы хранения в сбалансированных деревьях, где операции поиска, изменения, вставки и удаления объекта занимают $O(log(n))$ операций. Сложность для строк заключается в том, что определение отношения порядка занимает линейное от длины строки время, а для чисел отношение порядка определяется за $O(1)$. Заметим, что для lcp-естественных строк определение лексикографического порядка в среднем занимает $O(max_{lcp})$ (что можно считать как $O(1)$, так как max_{lcp} является константой; с практической точки зрения такой переход не является чистым). Имеются методы построения суффиксного массива, основанные на быстрой сортировке ([22]) с некоторыми ограничениями на структуры строк.

1.6. ЗАМЕЧАНИЕ ПО ИСПОЛЬЗУЕМЫМ СТРУКТУРАМ ДАННЫМ

Далее в работе будут строиться алгоритмы динамического построения суффиксных массивов. В строку будут вноситься изменения: добавление символа, добавление строки, удаление подстроки. Необходимо поддерживать суффиксный массив.

Необходим доступ по индексу (как к обыкновенному линейному массиву), так и эффективная реализация операций вставки и удаления элемента. В [5] рассматривается структура данных (*динамические порядковые статистики*), позволяющая производить все требуемые операции (индексный доступ к элементу, изменение элемента, удаление и вставка элемента) за $O(log(n))$ операций.

Динамические порядковые статистики основываются на би-

нарном сбалансированном дереве, где кроме значения элемента в вершине дерева хранится количество элементов в левом и правом поддеревьях (в [5] рассматривается реализация с красно-черным деревом: в вершине хранится сумма вершин в левом и правом поддереве плюс сама вершина). Сортировка в дереве идет по индексу. При операциях вставки или удалении производится балансировка дерева, в процессе которой поддерживается информация о количестве элементов в поддеревьях. Индексный доступ реализуется аналогично классическому поиску в бинарном дереве, зная количество элементов в поддеревья производится рекурсивное углубление в какое-то из поддеревьев.

Таким образом, у нас имеется структура данных, объединяющая индексный доступ к элементу, а также возможность эффективной вставки и удаления элемента. Далее в работе при доказательстве асимптотических сложностей алгоритмов будет учитываться дополнительный $O(log(n))$ член.

Объем занимаемой памяти для такой структуры составляет $O(n)$ [5].

1.7. ОСНОВНЫЕ ПОНЯТИЯ

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 $s[i..|s|]$.

1.8. СРАВНЕНИЕ СУЩЕСТВУЮЩИХ АЛГОРИТМОВ ДИНАМИЧЕСКОГО ИЗМЕНЕНИЯ СУФФИКСНЫХ МАССИВОВ

К настоящему времени существует только одна работа в данном направлении. В 2009 году вышла работа Салсона [30], в которой также рассматриваются алгоритмы динамического изменения

суффиксных массивов. В этой работе решены схожие задачи: изменение суффиксного массива при вставке символа и строки в середину существующей и при удаление подстроки.

Работа основана на динамическом преобразовании Барроуза-Уилера.

1.8.1. АЛГОРИТМ СЛИСЕНКО

В 1981 году опубликована работа О. А. Слисенко[5] «Поиск периодичностей подслов в реальное время». В работе рассмотрены задачи поиска вхождения в реальное время, т. е. во входной поток поступают символы текста, затем следует разделитель \$, после чего во входной поток поступают символы образца. При получении нового символа образца алгоритм дает ответ, входит ли образец в текст в реальное время (т. е. требуется константное число шагов[6])

В работах О. А. Слисенко ([7, 9]) строятся также алгоритм проверки симметричности в реальное время и поиск периодичностей (поиск самых длинных повторений) в реальное время.

1.9. О LCP И ПОИСКЕ МИНИМУМА НА ОТРЕЗКЕ

Одно из основных свойств lcp состоит в $lcp(i,j) = min_{k\in[i..j-1]}lcp_k$. Нам необходимо уметь эффективно решать эту задачу.

Тривиальный алгоритм поиска минимума на отрезке нам не подходит, ввиду линейной сложности.

В [6] задача поиска минимума на отрезке для статических массивов решается за $O(log(n))$ операций при помощи дерева отрезков. Для построения дерева отрезков требуется $O(n\,log(n))$ операций.

Есть производные структуры данных, основанные на дереве отрезков, которые позволяют искать минимум на отрезке за $O(log(n))$ операций и модифицировать элементы за $O(log(n))$.

[5] *Результаты получены ранее.*

[6] *Не путать с амортизированной единицей $O(1)$. Здесь константа задана и не зависит от входных данных.*

Но эти структуры данных не позволяют добавлять или удалять элементы массива. Построение таких структур данных также занимает $O(n\ log(n))$ операций.

В [31] представлена структура данных, основанная на сбалансированном дереве, позволяющая искать минимум на произвольном отрезке за $O(log(n))$ операций, а также изменять значение элемента, удалять и вставлять элемент за $O(log(n))$ операций.

При каждом изменении массива lcp (изменение значения элемента, удаление и вставка элемента) будем производить аналогичное действие над структурой данной для поиска минимума на отрезке.

Все структуры данных, рассмотренные в этом параграфе используют $O(n)$ памяти [5, 6, 31].

1.9.1. О LCP, LCA И ПОИСКЕ МИНИМУМА НА ОТРЕЗКЕ

Массив lcp выражает длину наибольшего общего префикса для соседних суффиксов в суффиксном массива. Чтобы найти наибольший общий префикс двух некоторых суффиксов в суффиксном дереве необходимо найти наименьшего общего родителя (lca, $least\ common\ ancestor$) для двух листьев, соответствующего этим суффиксам.

Очевидный алгоритм поиска наименьшего общего предка состоит в продвижении от каждого листа вверх к корню, помечая все узлы на своем пути. Найдя первый уже помеченный узел, мы тем самым найдем наименьшего общего предка. Как видно, сложность алгоритма линейная.

В [3, 10] рассмотрено несколько алгоритмов поиска наименьшего общего предка. Наилучшим является алгоритм с пропроцессингом за линейное время и с константным временем запроса (поиск наименьшего общего предка на полном двоичном дереве не является сложной задачей (достаточно «поиграть» с битовым представлением вершин), строится биективное отображение между суффиксным деревом и частью полного бинарного дерева).

Покажем связь между поиском наименьшего общего предка и задачей поиска минимума на отрезке.

Рассмотрим дерево (необязательно суффиксное) и две выделенные вершины (необязательно листья). Произведем поиск в глубину с левым обходом начиная с корня, записывая порядок посещения вершины в список L. При такой нумерации число, соответствующее некоторой вершине, меньше чисел сопоставляемых ее потомкам.

Для вычисления наименьшего общего предка для вершин i и j найдем любые вхождения элементов i, j в L (например, первые). Эти элементы ограничат некоторый интервал. Найдя минимум на этом интервале мы получим $lca(i, j)$. Заметим, что длина списка

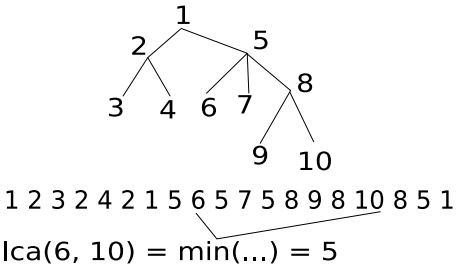

Рис. 3. Связь lca и поиска минимума на отрезке

L есть $O(n)$ (или $2n - 1$), так как каждое ребро дерева участвует дважды. То есть препроцессинг состоит в построении структуры данных для поиска минимума на отрезке за $O(n\,log(n))$ операций, а сам запрос lca для произвольных вершин производится за $O(log(n))$ операций (как поиск минимума на отрезке).

2. Полученные результаты

2.1. АЛГОРИТМ ПОСЛЕДОВАТЕЛЬНОГО ПОСТРОЕНИЯ СУФФИКСНОГО МАССИВА

Рассмотрим следующую задачу.

Имеется строка s и суффиксный массив sa для строки s. К строке s приписывается новый символ sym. Необходимо получить суффиксный массив для строки $sa + +sym$.

Конечно, можно построить суффиксный массив для строки $sa + +sym$ «с нуля». Нас интересует построение алгоритма, который может изменить существующий суффиксный массив за меньшее время, чем построение суффиксного массива заново.

Лемма 1 О наихудшем случае**.** *При добавление одного символа к строке может потребоваться не менее $O(n)$ операций для изменения суффиксного массива.*

Доказательство. Если строке $aa..a$ приписать символ b, то это приведет к обращению суффиксного массива. Это требует не менее $O(n)$ операций.

Высокоуровневый *алгоритм последовательного построения суффиксного массива* можно записать следующим образом:

1) Ко всем суффиксам приписать новый символ sym, поддерживая корректность суффиксного массива.

2) Добавить в суффиксный массив sym.

В таком виде этот алгоритм нас не устраивает, так как в первой части алгоритма требуется обработка всех суффиксов, что потребует не менее $O(n)$ операций.

Посредством следующей леммы покажем, что необходимо обрабатывать не все суффиксы.

Лемма 2 О граничных суффиксах**.** *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 $sa_i < sa_{i+1}$, поэтому добавление строки ns не изменит лексикографический порядок sa_i и sa_{i+1} суффиксов.

Утверждение $lcp_i = |sa_i|$ также означает, что $|sa_i| < |sa_{i+1}|$ и весь суффикс sa_i является префиксом для sa_{i+1} суффикса. 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 $sa_{i+1}[|sa_i| + 1..]$). Лемма сформулирована и доказана для случая добавления строки, что пригодится нам позже.

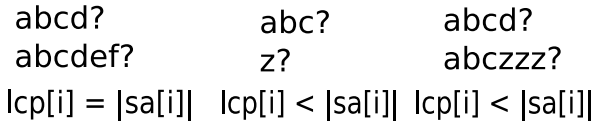

Рис. 4. Добавление новой строки изменяет положение суффиксов только если $lcp_i = |sa_i|$

Рассмотрим суффикс sa_i, для которого верно $lcp_i = |sa_i|$. При приписывании символа sym к суффиксу sa_i и сравнении $sa_i + +sym$ с sa_{i+1}-суффиксом возможны следующие варианты:

- $sym < s[sa_{i+1} + lcp_i]$. Тогда sa_i-суффикс будем лексикографически меньше sa_{i+1}-суффикса вне зависимости от очередных добавляемых символов. Необходимо снять пометку bo_i.

- $sym = s[sa_{i+1} + lcp_i]$. Тогда необходимо увеличить lcp_i на единицу и сохранить пометку bo_i. Следующий добавляемый символ может повлиять на порядок sa_i-суффикса относительно sa_{i+1}.

- $sym > s[sa_{i+1} + lcp_i]$. То есть sa_i-суффикс лексикографически больше sa_{i+1}-суффикса. Необходимо переместить sa_i-суффикс и модифицировать некоторые значения lcp. Значение bo_i определится в результате перемещения суффикса.

Первые два случая являются тривиальными.

Рассмотрим третий вариант. Здесь мы получили, что $sa_i + +sym > sa_{i+1}$. То есть необходимо перемещение суффикса на новое место и обновление дополнительных структур данных.

Докажем несколько полезных лемм.

Лемма 3 Об удалении**.** *При удалении sa_i-суффикса необходимо обновить значение $lcp_{i-1} \leftarrow min(lcp_{i-1}, lcp_i)$.*

Доказательство. Исходя из свойств lcp [26] имеет место $lcp(i, j) = min_{k \in [i..j-1]} lcp_k$. В частности, $lcp(i-1, i+1) = min(lcp_{i-1}, lcp_i)$. Теперь мы можем удалить суффикс. При помо-

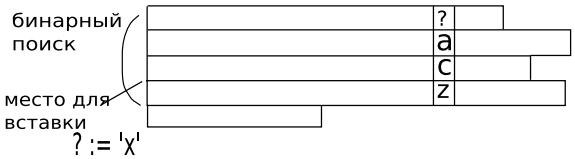

Рис. 5. Перемещение суффикса на новое местоположение

щи следующего алгоритма мы сможем определить новое местоположение и вставить $sa_i + +sym$.

Мы имеем текущее значение lcp_i и $sa_i + +sym > sa_{i+1}$. Найдем $r = min\{k|k > i \wedge lcp_k < lcp_i\}$. Позиция r означает, что на диапазоне $k \in [i..r]$ имеет место $lcp_k \geqslant lcp_i$. То есть на отрезке $[i + 1..r]$ имеется общий префикс для sa_i, и искомое местоположение находится на этом интервале. Найти требуемое местоположение можно посредством бинарного поиска на интервале $[i + 1..r]$, сравнивая новый символ sym с соответствующим символом sa_{med} суффикса (med представляет собой вспомогательную переменную для бинарного поиска). Найдя требуемое местоположение pos, необходимо обновить соседние значения массива lcp следующим образом: $lcp_{pos-1} = lcp_i$ (так как $sa_{pos-1} < sa_i + +sym$), а значение $lcp_{pos} = lcp_i \vee lcp_{pos} = lcp_i + 1$ в зависимости от символа sym и соответсвующего символа sa_{pos+1} суффикса.

Теперь мы можем перемещать (как операции удаления и вставки на новое место) суффикс.

Рассмотрим задачу поиска правой границы. Если абстрагироваться от суффиксных структур данных, необходимо для каждого элемента массива знать наиближайший элемент с большим индексом, значение которого меньше элемента.

Нас не устраивает очевидная реализация (для каждого элемента линейный проход вправо и поиск первого элемента меньше заданного) ввиду ограничений на сложность алгоритма.

Помимо этого интересует задача препроцессинга. То есть на основе суффиксного массива, построенного при помощи других

алгоритмов, достроить все дополнительные структуры данных, так чтобы можно было продолжить построение суффиксного массива.

Рассмотрим вариант, когда для каждого элемента будем хранить ссылку на следующий элемент, меньше данного. Обозначим массив как $link[]$. Покажем как построить массив $link$ на фазе препроцессинга за линейное время. Очевидная реализация имеет сложность $O(n^2)$.

```
for i ∈ 1 to n do
    while (stack.size > 0) ∧ (stack.peek > lcp[i]) do
        link[stack.top.ind] ← i;
    stack.pop();
    stack.push(val = lca[i], ind = i);
```

В стеке хранится возрастающая последовательность элементов (как дополнительный атрибут – индекс элемента). Производится линейный проход (по возрастанию индекса). Для всех значений, хранящихся в стеке (для которых пока не известен ближайший меньший элемент) и больших текущего элемента, мы устанавливаем ссылку на этот элемент и извлекаем эти значения из стека. Текущий элемент заносится в стек. Сложность алгоритма есть $O(n)$, так как каждый элемент максимум один раз заносится и извлекается из стека.

Заметим, что ссылка $link_i$ необходима только для тех суффиксов, у которых bo_i истина.

Имеются трудности с поддержкой $link$ в процессе работы алгоритма. В процессе перемещения суффиксов и изменения значений массива lcp необходимо поддерживать в корректном состоянии значения $link$. Эта проблема решаема, но мы не будем углубляться в этом направлении. Предложим более универсальное решение, которое пригодится нам в будущем.

Воспользуемся подзадачей поиска значения $lcp(i, j)$, где i, j – несмежные суффиксы. Как известно из свойств lcp (например, [26]) $lcp(i, j) = min_{k \in [i..j-1]} lcp_k$. В главе 1.9 показано, как

считать значение $lcp(i, j)$ для любых значений аргументов за $O(log(n))$ операций при помощи вычисления минимума на отрезке.

Тогда задачу поиска первого значения меньше текущего решим посредством бинарного поиска и поиска минимума на отрезке. Для вычисления $min\{k|k > i \wedge lcp_k < lcp_i\}$ будем искать минимум на отрезке $[i, x]$ (x – переменная в бинарном поиске, левая граница равна $i + 1$, правая – n) и считать характеристическую функцию $min_{k\in[i,x]}lcp_k < x$ (которая является монотонной относительно x).

Итого, мы можем искать правую границу для поиска за $O(log^2(n))$ операций.

Запишем более детализированную версию алгоритма последовательного построения суффиксного массива.

```
1   procedure Preprocessing()
2       Создать lcp
3       Создать DRMQ
4       Создать bo
5   end procedure Preprocessing
6   procedure SequentialSuffixArrayContructingGenerate(char sym)
7       Для всех суффиксов sa_i, помеченных bo_i,
8           Если sym < s[sa_{i+1} + lcp_i], то
9               bo_i ← false
10          Иначе Если sym = s[sa_{i+1} + lcp_i], то
11              lcp_i ← lcp_i + 1
12          Иначе
13              Найти правую границу бинарного поиска
14              Найти место для вставки sa_i (бинарный поиск)
15              Переместить sa_i на новое место
16              Обновить lcp
17              Обновить bo
18      Добавить в суффиксный массив sym
19      Обновить lcp
20      Обновить bo
```

2.1.1. ОЦЕНКА АЛГОРИТМИЧЕСКОЙ СЛОЖНОСТИ И ЗАТРАТ ПАМЯТИ

Оценим алгоритмическую сложность построенного алгоритма.

Сначала оценим сложность процедуры препроцессинга. Необходимо создать все дополнительные структуры данных для работы алгоритма только на основе суффиксного массива sa и самой строки s. Построение массива lcp требует $O(n)$ операций [21], но с учетом структуры хранения нам потребуется $O(n\,log(n))$ операций. Структура данных для эффективного поиска минимума значения на отрезке в строке 3 требует $O(n\,log(n))$ операций. Массив bo строится по определению за линейное время, но из-за структуры данных на его построение нам потребуется $O(n\,log(n))$ операций.

Итоговое время препроцессинга – $O(n\,log(n))$ операций.

Перейдем к оценке самого алгоритма.

На строках 7–17 имеется внешний цикл по всем суффиксам для которых верно $lcp_i = |sa_i|$. Количество итераций цикла может составлять $O(n)$ (для lcp-регулярных строк. Для lcp-естественных строк количество ограничено константой). Проверка условий и выполнение действий на строках 8–11 требует $O(log(n))$ операций. Для варианта с перемещением суффикса (строки 13–17) потребуется $O(log^2(n))$ операций. Для определения границы (строка 13) требуется $O(log^2(n))$ операций (бинарный поиск и поиск минимума на отрезке посредством $DRMQ$). Собственно сам бинарный поиск (строка 14) занимает так же $O(log^2(n))$ операций, так как имеются дополнительные затраты на структуры данных. Операция по перемещению суффикса рассматривается как операция удаления и вставки. По лемме об удалении на поддержку значений lcp нам потребуется $O(log(n))$ операций. При вставке и поддержке значений lcp и bo потребуется $O(log(n))$ операций. Итого весь цикл (строки 7–17) занимает $O(n\ log^2(n))$ операций для lcp-регулярных языков и $O(log^2(n))$ для lcp-естественных языков.

Для вставки нового символа sym в суффиксный массив sa

требуется $O(log^2(n))$ операций (как бинарный поиск по суффиксному массиву). Для конечных и небольших алфавитов можно вставить новый символ за $O(log(n))$ (храня дополнительную информацию о том, где начинаются суффиксы для каждого символа алфавита), но здесь оптимизация не скажется на общей асимптотике алгоритма. Поддержка дополнительных структур данных lcp и bo потребует $O(log(n))$ операций.

В результате мы получили, что сложность алгоритма последовательного построения суффиксного массива составляет $O(n\ log^2(n))$ операций для lcp-регулярных строк и $O(log^2(n))$ для lcp-естественных строк.

Для хранения всех дополнительных структур данных требуется $O(n)$ памяти [2, 5, 31].

2.1.2. ПРЕИМУЩЕСТВА АЛГОРИТМА И НЕДОСТАТКИ АЛГОРИТМА

Рассмотрим преимущества алгоритма.

Алгоритм работает с потоковыми данными. При поступлении информации о новых символах строки мы достраиваем существующий суффиксный массив, а не строим заново для полученной строки.

Имеется возможность препроцессинга. То есть для существующего суффиксного массива, построенного при помощи любого алгоритма, и строки построить все необходимые дополнительные структуры данных и продолжить построение суффиксного массива.

Также построенный алгоритм можно использовать для создания суффиксного массива «с нуля», добавляя по одному символу к пустому суффиксному массиву.

На каждом шаге поддерживается массив наибольших общих префиксов lcp.

Для lcp-естественных языков сложность добавления одного символа составляет $O(log^2(n))$ операций, а для построения суффиксного массива «с нуля» потребуется $O(n\,log^2(n))$ (что близко

к асимптотике построения суффиксного массива для полностью заданной строки).

Перейдем к недостаткам алгоритма.

Для lcp-регулярных строк добавление одного символа потребует $O(n\,log^2(n))$ операций. В лемме о наихудшем случае утверждается о не менее чем $O(n)$ операций.

Как видно из алгоритма, количество суффиксов, помеченных меткой bo_i, не обязательно уменьшается на каждой итерации.

Для создания суффиксного массива алгоритмом последовательного построения для строки $'aa..a'$ потребуется $O(n^2\,log^2(n))$ операций, так как на каждой итерации количество суффиксов sa_i для которых выполнено $lcp_i = |sa_i|$ будет равняться в точности длине текущей строки.

Наихудшим случаем будет также построение суффиксного массива «с нуля» для двойной копии большой строки. В процессе построения для второй копии количество помеченных bo_i суффиксов будет только увеличиваться, что приведет к асимптотике в $O(n^2\,log^2(n)$ операций.

2.2. АЛГОРИТМ БЛОЧНОГО ПОСТРОЕНИЯ СУФФИКСНОГО МАССИВА

К недостаткам алгоритма последовательного построения суффиксного массива относится то, что для lcp-регулярных строк на каждой итерации алгоритма может возрастать количество граничных суффиксов. Это приводит к квадратичному времени работы.

Этот недостаток мы устраним в алгоритме блочного построения суффиксного массива. Вместо добавления одного символа будем добавлять строку и покажем, что это можно сделать эффективнее, чем добавление по одному символу.

Будем добавлять не произвольной длины строку, а минимальную строку ns, не входящую в текущий суффиксный массив. То есть подстрока $ns[1..|ns|-1]$ является подстрокой s. Как выбрать такую строку ns будет показано ниже.

2.2.1. ПОСТРОЕНИЕ АЛГОРИТМА

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.
```

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.

По Лемме о граничных суффиксах для работы первой части алгоритма достаточно обрабатывать суффиксы, для которых верно $lcp_i = |sa_i|$.

Покажем, как определить отношение порядка для суффикса, для которого верно $lcp_i = |sa_i|$, со следующим (sa_{i+1}) после приписывания к обоим суффиксам строки 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 так же $ns[|sa_{i+1}|-|sa_i|+1..]$ и ns.

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 стро-

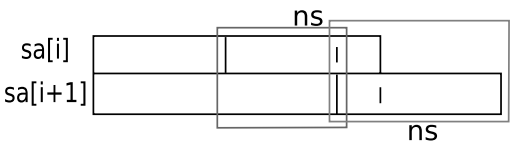

Рис. 6. Сравнение суффиксов разбивается на два сравнения

ку 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 [19]. 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 ниже.

Заметим, что аналогичным образом мы можем сравнивать любые суффиксы, а не только соседние. При сравнении $sa_i + +ns$ и $sa_j + +ns$ необходимо проверить, что $lcp(i,j) = |sa_i|$, иначе сравнение не имеет смысла, так как отношение порядка заведомо определено и не зависит от ns.

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 бинарным поиском, так как мы можем эффективно сравнивать $sa_i + +ns$ с $sa_j + +ns$.

2.2.2. ВТОРАЯ ЧАСТЬ АЛГОРИТМА

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 $O(k+log(n))$ [26]) 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. 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).

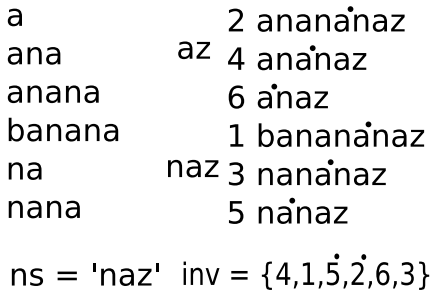

Рис. 7. К суффиксному массиву для слова $'banana'$ добавляем строку $'naz'$

Найдем наименьшее положение суффикса, префикс которого совпадает с $ns[2..|ns|-1]$. Обозначим местоположение такого суффикса как $pos_{ns.2}$. Для вставки $ns[2..]$ воспользуемся идеей из алгоритма последовательного построения суффиксного массива. Применим бинарный поиск по последнему символу $ns[|ns|]$

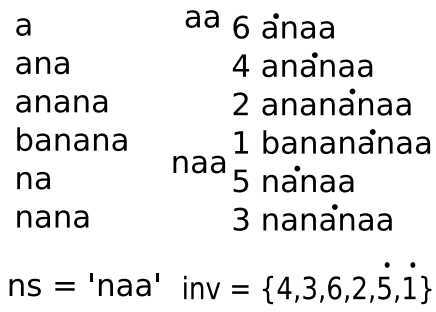

Рис. 8. К суффиксному массиву для слова $'banana'$ добавляем строку $'naa'$

на интервале от $pos_{ns.2} - 1$ до $mink|k > pos_{ns.2} \wedge lcp_k < |ns| - 1$ 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}). Элемент обратной перестановки $ins_{sa}[i]$ будет показывать положение в суффиксном массива суффикса с длиной $|s| - i + 1$.

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

2.2.3. ПОДРОБНАЯ ВЕРСИЯ АЛГОРИТМА

Запишем алгоритм более подробно.

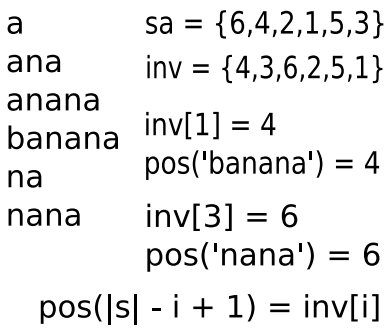

Рис. 9. Определение положения суффикса по его длине

```
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      Если sa_i ++ns > sa_{i+1} ++ns, то
12          перемещение sa_i ++ns,
13          обновление значения lcp, bo
14      иначе
15          обновление значений lcp, bo.
16
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

Следствие 1. *В процессе сравнения* $sa_i + +ns$ *с* $sa_j + +ns$ *можно вычислить* $lcp(sa_i + +ns, sa_j + +ns)$ *за* $O(log(n))$ *операций.*

Доказательство. Сравнение $sa_i + +ns$ и $sa_j + +ns$, как было показано выше, разбивается на два подсравнения. Первое подсравнение состоит в сравнении ns и некоторого суффикса. Так как мы уже добавили ns в суффиксный массив, то определяем значение lcp стандартным способом за $O(log(n))$ операций. Если потребуется второе подстравнение (некоторый суффикс ns и строка ns), то это производится запрос к суффиксному массиву для строки ns за $O(log(k))$ операций.

Так как $k <= n$, то общее время вычисления $lcp(sa_i + +ns, sa_j + +ns)$ составляет $O(log(n))$.

2.2.4. ОЦЕНКА АЛГОРИТМИЧЕСКОЙ СЛОЖНОСТИ АЛГОРИТМА И ЗАТРАТЫ ПАМЯТИ

Строка 6 (Построение суффиксного массива для строки ns) – потребуется $O(k)$ операций [19].

На строке 7 (Вставка в суффиксный массив sa строку ns) потребуется $O(k + log(n))$ на вставку [26]. 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) [26, 31].

В случае перемещения sa_i суффикса (строки 12–13) для поиска нового положения воспользуемся бинарным поиском. Тогда поиск нового положения суффикса займет $O(log^2(n))$ (бинарный поиск и вычисление lcp).

Обновление значений lcp на строках 13, 15 занимает $O(log(n))$.

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

Вставка всех суффиксов строки ns занимает $O(k\,log^2(n))$. Так как поиск первоначального места (строка 17) требует $O(k + log(n)$ ([26]), 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 $O(n\ log^2(n))$.

В процессе работы алгоритма используются только структуры данных с объемом памяти $O(n)$ (lcp, bo, inv_{sa}, RMQ, суффиксный массив, [5, 26, 31]).

2.2.5. 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 $O(n)$ [21].

Массив bo так же строится за $O(n)$, линейный проход по всем суффиксам $bo_i = (lcp_i == length(sa_i))$. Заметим, что число помеченных суффиксов может составлять $O(n)$.

Структура RMQ нам требуется при основной работе алгоритма. Построение занимает $O(n\,log(n))$ операций [31]. 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 [31].

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.

2.2.6. ВЫБОР БЛОКА

Покажем, как выбрать минимальную подстроку, не входящую в текущий суффиксный массив.

Так как мы работаем с потоковыми данными, то считаем, что в качестве входа имеется новый блок данных D. Бинарным поиском по длине определяем минимальную строку (префикс D), не входящую в качестве подстроки в s. В процессе поиска возможны два варианта.

Мы нашли требуемую строку ns. Тогда применяем алгоритм блочного построения суффиксного массива и удаляем префикс в блоке данных D размером $|ns|$.

Весь блок D является подстрокой s. Тогда возможны следующие варианты.

- Продолжим ожидание данных.
- Продолжим построение суффиксного массива при помощи алгоритма последовательного построения.
- Принудительно к блоку D припишем стоп-символ \$ вне алфавита. Тогда D ++\$ будем искомой строкой. Все поисковые запросы к суффиксному массиву будут работать корректно. Для продолжения построения суффиксного массива при получении нового блока данных необходимо удалить стоп-символ из суффиксного массива. Алгоритм удаления строится в следующей главе.

2.2.7. ПРЕИМУЩЕСТВА И НЕДОСТАТКИ АЛГОРИТМА

Рассмотрим преимущества алгоритма.

Для работы алгоритма не обязательно иметь исходную строку целиком. Это позволяет использовать предложенный алгоритм в задачах с потоковыми данными.

В отличие от алгоритма последовательного построения суффиксного массива, 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 (по следствию леммы об удалении граничных суффиксов, если на некоторой итерации алгоритма суффикс будет помечен меткой bo, то на следующей итерации метка обязательно будет снята).

Для суффиксного массива (созданный любым другим алгоритмом) можно за время $O(n\ log(n)$ построить дополнительные структуры данных lcp, $bool$, RMQ [21, 31] 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 [5]).

Перейдем к недостаткам алгоритма.

Вследствие того, что ns является минимальной строкой, не входящей в текущий суффиксный массив, может показаться что в результате работы одной итерации алгоритма все метки bo у существующих суффиксов снимаются.

При обработке суффиксов сравнение разбивается на два подсравнения с возможным вырожденным вторым подсравнением. Если второе подсравнение вырождено, то гарантируется, что метка bo будет снята для этого суффикса ввиду выбора строки ns. Иначе, если для определения отношения лексикографического порядка строк $sa_i + +ns$ и $sa_{i+1} + +ns$ потребовалось второе подсравнение, то метка bo_i останется, если суффикс ns является префиксом ns.

Для слова $'banana'$ добавление слова $'nanan'$ не удаляет ни одного граничного суффикса. И так далее, добавление строк $'ana..ana'$ $((an) * a)$ и $'nana..an'$ $('(na) * n')$ не удалит ни одного граничного суффикса.

Это не приводит к квадратичному времени работы (для этого примера), так как длина добавляемой строки с каждым шагом

удваивается. Для того чтобы алгоритм имел квадратичную сложность, необходимо, чтобы количество граничных суффиксов на каждой итерации алгоритма было $O(n)$ и добавление на каждой итерации строки ns ограниченной длины сохраняло бы количество граничных суффиксов на уровне $O(n)$. Конструктивных примеров пока не известно.

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, [19]) заново для всей строки асимптотически окажется лучше, чем модификация существующего суффиксного массива алгоритмом блочного построения суффиксного массива.

Лемма о наихудшем случае работает и здесь.

2.3. УДАЛЕНИЕ ПОДСТРОКИ

Рассмотрим следующую задачу. Из суффиксного массива sa для строки s необходимо удалить подстроку sd и получить модифицированный суффиксный массив для измененной строки. Положение удаляемой подстроки sd ничем не ограничено (в начале, в середине, в конце). Без ограничений общности будем считать, что удаляемая подстрока sd находится внутри строки s. Позиция подстроки sd задается началом sd_{begin} и концом sd_{end}.

Лемма 4 О наихудшем случае при удалении**.** *При удалении одного символа из строки на модификацию суффиксного массива может потребоваться не менее $O(n)$ операций.*

Доказательство. Пример, для которого потребуется не менее $O(n)$ операций на изменение суффиксного массива при удалении одного символа из строки, противоположен примеру из Леммы о наихудшем случае.

Удаление из строки $aaa..aab$ последнего символа $'b'$ приведет к «развороту» суффиксного массива, что потребует не менее $O(n)$ операций.

Высокоуровневый алгоритм удаления подстроки из суффискного массива состоит из двух частей:

```
Удалить все суффиксы, начинающиеся с
   позиции из \in [sd_begin, sd_end].

Переместить суффиксы, на которые повлияет
удаление sd.
```

Считаем, что у нас есть все дополнительные структуры данных, используемые в алгоритма блочного построения суффиксного массива (а именно, lcp, bo, inv_{sa}, RMQ).

Тогда для реализации первой части алгоритма достаточно пробежаться по суффиксам нужной длины $inv_{sa}[i]|i \in [sd_{begin}, sd_{end}]$ и удалить их. По Лемме об удалении, при удалении sa_i суффикса изменяем $lcp_{i-1} \leftarrow min(lcp_{i-1}, lcp_i)$ [26].

Перейдем ко второй части алгоритма.

Заметим, что суффиксы, находящиеся справа от удаляемой подстроки sd, не изменят своего положения. То есть рассматривать стоит только суффиксы, для которых $sa_i < sd_{begin}$.

При удалении подстроки, суффиксы могут переместиться «вверх», в отличие от алгоритма блочного построения суффиксного массива (где суффиксы перемещаются только «вниз»).

Суффикс sa_i в результате удаления подстроки sd может переместиться вниз только при условии, что $sa_i + lcp_i > sd_{begin} \wedge sa_i < sd_{begin}$. Условие с lcp показывает, что позиция, в которой sa_i отличается от sa_{i+1}, находится правее начала удаляемой подстроки, что может изменить положение sa_i суффикса. Аналогично, необходимо выполнение условия $sa_i + lcp_{i-1} > sd_{begin} \wedge sa_i < sd_{begin}$ для того чтобы суффикс sa_i переместился вверх в результате удаления подстроки sd.

То есть нам необходимо обработать все суффиксы, для которых $sa_i < sd_{begin} \wedge (sa_i + lcp_i > sd_{begin} \vee sa_i + lcp_{i-1} > sd_{begin})$.

Покажем, как обработать такой суффикс.

После удалении подстроки sd из sa_i суффикса произведем сравнение sa_i^{-sd} с соседними суффиксами sa_{i+1} и sa_{i-1}. Это надо сделать эффективно, так как очевидное сравнение за линейный проход в итоге приведет к квадратичному времени.

Сначала рассмотрим сравнение sa_i^{-sd} и sa_{i+1} суффиксов. При удалении строки sd из sa_i суффикса происходит сдвиг $s[sd_{end}+1..]$ на $|sd|$ влево.

Возможны три варианта:

- В sa_{i+1} суффиксе подстрока sd еще не удалена или отсутствует (т. е. $sa_{i+1} > sd_{end}$). Тогда необходимо $s[sd_{end}+1..]$ сравнить с соответствующим sa_{i+1} подсуффиксом, а именно, с $sa_{i+1}[sa_i - 1 + sd_{begin}]$.

- В sa_{i+1} суффиксе подстрока sd уже удалена, и в sa_{i+1} строка sd находилась правее, чем в sa_i (т е. $sa_i < sa_{i+1}$). Тогда сравнение разбивается на два. Рассмотрим этот вариант ниже.

- В sa_{i+1} суффиксе подстрока sd уже удалена, и в sa_{i+1} строка sd находилась левее, чем в sa_i (т. е. $sa_i > sai + 1$). Тогда необходимо сравнить $s[sd_{end}+1..]$ с соответствующим sa_{i+1} подсуффиксом с учетом того, что в sa_{i+1} подстрока sd уже удалена, а именно, $sa_{i+1}[sa_i - 1 + sd_{end}]$.

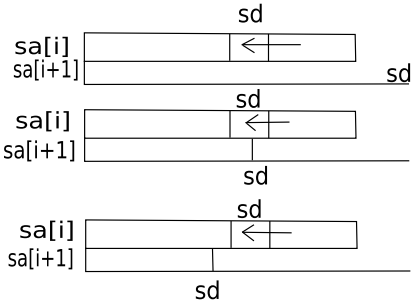

Рис. 10. Разбор случаев при сравнении sa_i^{sd} с sa_{i+1} суффиксом

Первый и третий вариант похожи: необходимо сравнить остаток $s[sd_{end}+1..]$ с соответствующим подсуффиксом sa_{i+1} (в последнем варианте – учитывая, что из sa_{i+1} подстрока sd уже удалена).

Второй вариант осложнен тем, что сравнение разбивается на два подсравнения. Сначала надо сравнить $s[sd_{end}+1..sa_i-sa_{i+1}+sa_{end}+1]$ с $s[sd_{begin}-1-sa_i+sa_{i+1}..sd_{begin}-1]$. Затем, в случае равенства, перейти ко второму сравнению, $s[sa_i-sa_{i+1}+sa_{end}+2..]$ с $s[sd_{end}+1..]$.

При сравнении sa_i^{-sd} и sa_{i+1} суффиксов вычислим значение lcp_i. Значение lcp_i определяется в процессе сравнения строк (это осуществимо, так как все сравнения производятся над текущими суффиксами).

Получив значение lcp_i, и в случае, если $sa_i^{-sd} > sa_{i+1}$, необходимо переместить sa_i^{-sd} суффикс. Как и в алгоритме блочного построения суффиксного массива, зная lcp_i, находим новое местоположение для суффикса, воспользуемся бинарным поиском.

Если в результате сравнения имеет место $sa_i^{-sd} < sa_{i+1}$, то производим сравнение sa_i^{-sd} с sa_{i-1} суффиксом. Рассуждения аналогичны для сравнения sa_i^{-sd} с sa_{i+1} суффиксом.

Запишем алгоритм удаления подстроки из суффиксного массива и оценим его алгоритмическую сложность.

```
1   procedure Remove(string sd)
2   Для всех суффиксов из i ∈ sd_begin..sd_end
3       Удалить sa_i суффикс
4       Обновить lcp
5
6   Для всех суффиксов i, для которых верно
7   sa_i < sd_begin ∧ (sa_i + lcp_i > sd_begin ∨ sa_i + lcp_{i-1} > sd_begin)
8       compare_{i,i+1} ← Сравнить sa_i^{-sd} и sa_{i+1}
9       Вычислить lcp_i
10      Если compare_{i,i+1}
11          Переместить sa_i^{-sd}
12          Обновить lcp
13      Иначе
14          compare_{i,i-1} ← Сравнить sa_i^{-sd} и sa_{i-1}
15          Вычислить lcp_{i-1}
16          Если compare_{i,i-1}
17              Переместить sa_i^{-sd}
160
```

18 Обновить lcp
19 **end procedure**

Перейдем к сложностной оценке алгоритма. Обозначим длину $|sd|$ как k.

На фазу удаления суффиксов (строки 2–4) потребуется $O(k\,log(n)$ операций, так как необходимо удалить k суффиксов. На обработку одного суффикса требуется $O(log(n))$ операций: удаление из структуры данных занимает $O(log(n))$, обновление lcp так же требует $O(log(n))$ операций (по лемме об удалении).

Количество итераций для цикла ограничено сверху максимальным значением lcp. Для lcp-регулярных строк это значение есть $O(n)$, для lcp-естественных – ограничено константой.

Операция сравнения двух суффиксов требует $O(log(n))$ операций (при сравнении производится вычисление значения lcp и поиск положения суффикса по его длине).

Для поиска нового местоположения суффикса при перемещении воспользуемся бинарным поиском. Для этого потребуется $O(log^2(n))$ операций, так как на сравнение требуется $O(log(n))$ операций.

Итоговая сложность алгоритма составляет $O(n\,log^2(n))$ операция для lcp-регулярных строк и $O(log^2(n))$ операций для lcp-естественных строк.

2.3.1. ПРЕПРОЦЕССИНГ

Для работы алгоритмы достаточно иметь все структуры данных при препроцессинге в алгоритме блочного построения суффиксного массива. Сложность препроцессинга составляет $O(n\,log(n))$ операций.

2.3.2. ОЦЕНКА ИСПОЛЬЗУЕМОЙ ПАМЯТИ

Для работы алгоритма требуется $O(n)$ памяти, так как в процессе работы используются только структуры данных из алгоритма блочного построения суффиксного массива. Такая проверка и

обновление информации потребует $O(log(n))$ операций, что никак не влияет на общую асимптотику алгоритма.

2.3.3. ПОДДЕРЖКА ИНФОРМАЦИИ О ГРАНИЧНЫХ СУФФИКСАХ

В процессе работы алгоритма удаления блока из суффиксного массива мы не используем информацию о граничных суффиксах (суффиксы, для которых верно $lcp_i = |sa_i|$). Такая информация необходима для алгоритма блочного построения суффиксного массива.

Будем поддерживать информацию о граничных суффиксах в процессе работы алгоритма. Для этого достаточно проверять условие граничности суффикса при сравнении и перемещении суффиксов.

2.3.4. НАИХУДШИЙ СЛУЧАЙ

Как доказано в Лемме (о наихудшем случае при удалении), удаление одного символа может потребовать не менее $O(n)$ операций. Для указанного в Лемме примере (из строки $'aaa..aab'$ удаляется последний символ $'b'$) для алгоритма удаления блока из суффиксного массива потребуется $O(n\ log^2(n))$ операций (так как строка $'aaa..aab'$ является lcp-регулярной).

3. Приложения построенных алгоритмов

В этом разделе мы будем рассматривать приложения алгоритмов последовательного и блочного построения суффиксного массива и блочного удаления. Здесь мы не будем обсуждать преимущества и недостатки построенных алгоритмов, так как это рассматривалось в конце каждой главы для соответствующего алгоритма.

Алгоритм последовательного построения суффиксного массива является частным случаем блочного построения как по постановке задачи, так и частично по методу решения. Так

же алгоритм блочного построения является более эффективным чем алгоритм последовательного построения (наихудший случай $O(n\,log^2(n))$ против $O(n^2\,log^2(n))$ для построения с нуля для lcp-регулярных строк).

Поэтому в основном будет рассматриваться связка алгоритмов блочного построения и блочного удаления.

Алгоритм последовательного построения суффиксного массива можно использовать на практике для

- более простой технической реализации;
- для узконаправленной задачи (посимвольное добавление);
- для более экономичного использования памяти.

В остальных случаях рекомендуется использовать алгоритм блочного построения суффиксного массива.

Существующие алгоритмы построения суффиксных массивов ([19, 22, 26] требуют наличие всей строки. Предложенные алгоритмы позволяют динамически строить и изменять суффиксный массив.

3.1. ПРИЛОЖЕНИЕ ДЛЯ БАЗ ДАННЫХ И ФАЙЛОВЫХ СИСТЕМ

Покажем, как можно использовать предложенные алгоритмы при реализации дополнительных индексных структур в базах данных и файловой системе. В этих предметных областях имеется довольно большое число строковых объектов (поле записи в базе данных, имя файла в файловой системе), для которых необходима эффективная реализация функции поиска.

Для большинства файловых систем поиск файла по имени происходит рекурсивным перебором всех файлов в дереве из директорий и проверкой того, что имя файла удовлетворяет шаблону.

В базах данных при выборке записей по некоторому шаблону происходит проверка всех записей на удовлетворение поля шаблону.

Если хранить все объекты в одной общей строке и поддерживать суффиксный массив при изменении строки, то запрос на поиск подстроки можно выполнять за $O(m + log(n))$ операций [3].

В общем виде, простая конкатенация всех объектов в одну общую строку приводит к lcp-регулярным строкам (ничто не запрещает поместить в базу данных миллион слов $'banana'$ или создавать только файлы $'readme.txt'$), так как размер имени файла обычно ограничен некоторой небольшой константой (порядка 300) и в базе данных есть естественные ограничения на размер записи. Хотя физических ограничений нет, данные больших объемов, подверженные изменениям, часто можно разбить на более мелкие части. Например, большие романы (такие как «Война и мир» Л. Н. Толстого) состоят из нескольких томов, каждый том состоит из нескольких частей, глав. Не имеется практического смысла искать подстроку, являющуюся суффиксом третьего тома и содержащую префикс четвертого тома.

Преобразуем lcp-регулярную строку в lcp-естественную (с условием, что размер объекта небольшой). В общую строку будем приписывать не просто объекты (имя файла или поле записи), а объекты, разделенные уникальными метаданными и стоп-символами. То есть общая строка будет выглядеть как

$$data_1\$metaData_1\$data_2\$metaData_2\$data_3\$metaData_3\$$$

В качестве метаданных для баз данных предлагается использовать уникальный идентификатор записи (уникальный ключ), а для файловой системы – уникальные данные для файла (ссылка на хранение дополнительной метаинформации файла, адрес смещения относительно начала диска). Так как уникальная метаинформация, разделенная стоп-символами \$, нигде более в строке не встречается, то максимальное значение lcp для такой строки ограничено максимальным размером $data_i$, что ограничено константой. Таким образом, мы свели задачу к lcp-естественным строкам.

Добавление нового объекта происходит в конец строки при помощи алгоритма блочного построения суффиксного массива.

Удаление объекта производится алгоритмом блочного удаления из суффиксного массива. Изменение объекта производится как операция удаления объекта и добавления изменений в конец строки. Все эти операции производятся за $O(k\,log^2(n))$ операций.

Предлагаемое решение для индексации имен файлов в файловой системе имеет некоторые пока не решенные задачи. Например, задача о сужении поиска (когда производится поиск файла по имени в заданной поддиректории). Необходимо из всего индекса имен файлов (суффиксного массива) в процессе поиска выделить только имена файлов, принадлежащие конкретной директории.

3.2. ПОИСК НАИБОЛЬШЕЙ ОБЩЕЙ ПОДСТРОКИ

Задача поиска наибольшей общей подстроки в тексте на практике используется в системах класса *CPD* (*copy paste detector*) или *DCD* (*Duplicate Code Detection*)(примеры систем PMD, Simian). При реализации больших проектов программист может не воспользоваться повторно-используемым кодом, а продублировать логику программы в другом месте (просто скопировав часть программы). Такое приложение теряет гибкость, так как при изменении необходимо поддерживать код в более чем одном месте. Системы *CPD* позволяют искать скопированные участки программного кода.

Постановка задачи: дана строка. Найти два суффикса, общий префикс которых максимален.

Самое очевидной решение имеет сложность $O(n^5)$ (за квадрат перебираем первый суффикс, еще за квадрат – второй суффикс, и еще линия на поиск общего префикса).

Следующее по очевидности решение имеет сложность $O(n^3)$ (перебираем начало первого суффикса за линейное время, затем перебираем начало второго суффикса, линейный проход на поиск общего префикса).

В качестве несложной реализации предлагается использовать бинарный поиск по длине результата, вычисление хеш-функций для всех подстрок заданной длины (при помощи алгоритма Рабина-Карпа), при одинаковых значениях хеш-функции производится посимвольное сравнение строк. Такое решение тре-

бует $O(n\,log(n))$ операций: ($O(log(n))$ требуется на бинарный поиск, $O(n)$ – на алгоритма Рабина-Карпа, для хранения значений хеш-функции от подстрок будем использовать хеш-таблицу с доступом в $O(1)$).

Теперь перейдем к решениям задачи о наибольшей подстроки в терминах суффиксных структур данных.

В результате построения суффиксного дерева для строки задача поиска наибольшей общей подстроки сводится к поиску самого глубокого узла (вершина, в которой происходит «раздвоение») в суффиксном дереве. То есть поиск двух листов, высота lca которых будет максимальна. Для этого требуется $O(n)$ операций: ($O(n)$ операций на построение суффиксного дерева, $O(n)$ операций на проход по суффиксному дереву).

После построения суффиксного массива для строки задача поиска наибольшей общей подстроки сводится к поиску максимального значения в массиве lcp. Здесь требуется тоже $O(n)$ операций: ($O(n)$ – на построение суффиксного массива, $O(n)$ – на построение lcp по суффиксному массиву, $O(n)$ – на поиск максимума в lcp).

Все эти предложенные решения статичны, т. е. требуют наличия всей строки. Используя алгоритмы блочного построения суффиксного массива и удаления блока из суффиксного массива мы сможем решать задачу поиска наибольшей общей строки динамически, т. е. имеется возможность изменять строку и в любой момент произвести запрос на поиск наибольшей подстроки.

Для этого достаточно хранить все значения lcp в структуре данных для поиска максимума (например, двоичная куча, или бинарное сбалансированное дерево) и производить изменения с этой структурой синхронно с изменениями в массиве lcp. При изменении массива lcp изменение в структуре данных для поддержки максимума производится за $O(log(n))$ операций (для двоичной кучи и сбалансированного дереве [5]), что не сказывается на общей асимптотике алгоритмов блочного построения и удаления из суффиксного массива.

В любой момент можно запросить максимальное значение

lcp (для двоичной кучи поиск максимума есть $O(1)$, для сбалансированного дерева – $O(log(n))$).

3.3. ПОИСК НАИБОЛЬШЕЙ ОБЩЕЙ ПОДСТРОКИ ДЛЯ ДВУХ СТРОК

Модифицируем задачу о поиске наибольшей подстроки. Дано две строки s_1 и s_2. Необходимо выделить в каждой строке по суффиксу так, чтобы их общий префикс был максимален.

Здесь применимы два очевидных решения: за $O(n^5)$ и $O(n^3)$, как и в задаче о наибольшей общей подстроки.

При помощи динамического программирования эту задачу можно решить за $O(n^2)$. В двумерной таблице элемент $a_{i,j}$ равен длине наибольшей подстроки, заканчивающейся в $s_1[i]$, $s_2[j]$ символах соответственно.

Имеется переход:

$$a_{i,j} = \begin{cases} a_{i-1,j-1} + 1, & \text{если } s_1[i] = s_2[j], \\ 0, & \text{если } s_1[i] \neq s_2[j]. \end{cases}$$

Максимальное значение в матрице определяет наибольшую общую подстроку для двух строк. К недостаткам этого алгоритма относится квадратичное требование по памяти. Все рассмотренные до этого решения имели линейные затраты по памяти.

Решение динамическим программированием применимо и к первоначальной задаче о поиска наибольшей общей подстроки для одной строки. Необходимо запретить использовать суффиксы, которые начинаются в одной позиции. Тогда переход записывается следующим образом:

$$a_{i,j} = \begin{cases} 0, & \text{если } i = j, \\ a_{i-1,j-1} + 1, & \text{если } s_1[i] = s_2[j], \\ 0, & \text{если } s_1[i] \neq s_2[j]. \end{cases}$$

Для этой задачи применимо решение посредством бинарного поиска по длине результирующей строки и вычисления хеш-функции для подстрок этой длины. Необходимо объединить обе строки s_1 и s_2 в общую, разделив их стоп-символом \$. Оценка сложности алгоритма остается неизменной – $O(n\ log(n))$.

Формулируя задачу для суффиксных массивов, необходимо для строки $str = s_1\$s_2$ построить суффиксный массив и найти суффикс sa_i такой, что он принадлежит одной строке, sa_{i+1} принадлежит, соответственно, другой строке, а lcp_i – максимален. Для такого решения требуется $O(n)$ операций ($O(n)$ на построение суффиксного массива для строки str, $O(n)$ на построение lcp, $O(n)$ на поиск максимума).

Построив суффиксные деревья для строк s_1 и s_2 за $O(n)$, объединим их в одну структуру. Тогда, используя динамическое программирование, поднимаясь с листьев, будем проставлять в вершинах индекс строки (т. е. информацию о том, к какой строке s_1 или s_2 принадлежит данный суффикс). Тогда для решения задачи о поиске наибольшей общей подстроки необходимо найти вершину с наибольшей высотой, для которой в построенном множестве есть два числа $\{1, 2\}$, что можно сделать за $O(n)$ операций. Для построения таких множеств во всех вершинах методом динамического программирования потребуется $O(n)$ операций. Итоговая сложность алгоритма составляет $O(n)$ операций.

Используя алгоритм блочного построения суффиксного массива и алгоритм удаления подстроки, задачу о поиске наибольшей общей подстроки для k строк мы сможем решать динамически. Для этого необходимо, как и для предыдущей задачи, строить суффиксный массив для строки $s_1\$s_2$ и поддерживать значения lcp в структуре данных для быстрого поиска максимума только для тех суффиксов sa_i, для которых sa_{i+1} принадлежит противоположной строке (при перемещениях суффиксов такая проверка осуществляется за $O(log(n))$ операций, что никак не сказывается на общей асимптотике алгоритмов работы с суффиксным массивом).

В любой момент можно произвести запрос на получение наи-

большей общей подстроки для двух строк и за $O(1)$ получить ответ.

3.4. ПОИСК НАИБОЛЬШЕЙ ОБЩЕЙ ПОДСТРОКИ ДЛЯ K СТРОК

Обобщим задачу на k строк[7]. Пусть даны строки $s_1, s_2 \dots s_k$. Необходимо в каждой строке выбрать по одному суффиксу так, чтобы общий префикс всех этих суффиксов был максимален.

Отметим, что «жадное» решение здесь неприменимо. Решение с поиском наибольшей общей подстроки для $r = substring_{max}(s_1, s_2)$, а затем решение задачи для строк $r, s_3, s_4, \dots, s_k$ является неверным. Контрпример прост: $'abcbb', 'abcabb', 'bb'$.

Тривиальные полиномиальные решения имеют сложность $O(n^{2k+1})$ или $O(n^{k+1})$. Практической ценности, как и для частных случаев ($k = 2$), они не имеют.

Решение динамическим программированием также не имеет практической ценности, так как потребуется k-мерная матрица.

Решение с помощью алгоритма Рабина-Карпа применимо и для k строк. Сложность алгоритма осталась без изменений $O(n\,log(n))$.

Решение с использованием суффиксных деревьев имеет линейные затраты по времени. Строим суффиксное дерево для каждой строки s_i за $O(n)$. Объединяем все суффиксные деревья в одну структуру за $O(n)$. При помощи динамического программирования проставляем для всех вершин метку о том, что данная вершина содержит суффикс из s_j строки. Вершина с наибольшей высотой, в которой проставлены все метки $\{1, 2, \dots, k\}$ и будет решением (точнее, строка от корня до этой вершины, наибольший общий префикс для всех строк).

3.5. ДИНАМИЧЕСКОЕ РЕШЕНИЕ С ИСПОЛЬЗОВАНИЕМ СУФФИКСНЫХ МАССИВОВ

Построим суффиксный массив для общей строки $s_1\$_1 s_2\$_2 \dots \$_{k-1} s_k$ и создадим lcp для суффиксного масси-

[7] *«Две строки хорошо, а четыре лучше» [3].*

ва. В данном случае все разделители $\$_i$ должны быть различны и не принадлежать алфавиту Σ.

Для решения задачи о поиске наибольшей общей подстроки для k строк необходимо найти интервал в массиве lcp, такой что на нем присутствуют суффиксы всех строк и значение минимума на этом интервале максимально.

Покажем, как искать это значение динамически при изменении любой строки.

Для каждой строки s_i будем хранить в отсортированной структуре pos_i (основанной, например, на бинарном сбалансированном дереве) позиции вхождений суффиксов s_i строки в общую строку s. При перемещении любого суффикса в строке s производим аналогичную операцию для соответствующей структуры pos_i за $O(log(n))$ операций.

Утверждение 1. *Значение минимума на интервале меньше либо равно значению минимума на надынтервале.*

Поэтому мы не будем рассматривать интервалы, у которых на границах имеются суффиксы из одной строки s_i (например, интервалы $[2, 2, 3, 1]$, $[1, 2, 3, 3]$ нас не интересуют).

В процессе работы алгоритмов блочного построения суффиксного массива и удаления блока из суффиксного массива производятся следующие изменения над массивом lcp: изменение значения элемента, удаление элемента, вставка элемента.

При каждом изменении lcp будем искать интервалы (содержащие суффиксы всех строк s_i), значение которых может измениться.

Пусть производится изменение над элементом lcp_i.

Для каждой строки s_l ($l \in [1..k]$) найдем максимальное значение в массиве pos_l такое, что $pos_l[left_{idx}] <= i$. Это значение будет левой границей интервала, начало которого принадлежит строке s_l. Для нахождения правой границы интервала для каждой строки s_p $p \in [1..k]$ найдем в массиве $pos_p[]$ минимальное значение, которое больше или равно $left$. Среди этих значений выберем максимальное. Это и будет правой границей интервала. Как видно из построения, на этом интервале присутствуют суф-

фиксы всех строк s_i $i \in [1..k]$, и интервал является минимальным (в том смысле, что ни один его подынтервал не будет содержать суффиксы всех строк).

Поиск интервалов при изменении элемента lcp занимает $k\,log(n)$, так как поиск левой и правой границы для одного интервала требует $log(n)$ операций (бинарный поиск по соответствующей структуре pos_i), количество интервалов не более k (левая граница определяется по индексу строки s_i).

Удаление элемента lcp состоит из следующих фаз:

- создание всех интервалов;
- удаление интервалов из структуры данных по поддержанию максимума;
- собственно удаление элемента lcp;
- создание всех интервалов на модифицированном массиве lcp;
- поиск минимума на всех полученных интервалах и добавление этой информации в структуру для поддержания максимума.

При вставке элемента в массив lcp производятся аналогичные действия. При изменении значения элемента lcp производится поиск всех интервалов и обновление значений минимумов в структуре данных по поддержанию максимума.

Искать значение минимума на интервале мы уже умеем за $O(log(n))$.

Итоговая сложность алгоритма составляет $O(k\,log(n))$ операций на каждое изменение массива lcp.

3.6. ЗАМЕЧАНИЕ ДЛЯ АЛГОРИТМА РАБИНА-КАРПА

Стоит отличать задачи поиска первого вхождения от задачи поиска всех вхождений. Асимптотическая сложность алгоритма

Рабина-Карпа для задачи поиска первого вхождения составляет $O(n)$ операций, но для задачи поиска всех вхождений в худшем случае может потребоваться $O(n^2)$ операций. Например, для строки $'aa..aa'$ (длины $2n$ символов) найти все вхождения строки $'aa..aa'$ (длины n символов). Хотя переход к следующему значению хеш-функции и будет осуществляться за $O(1)$ операций, но так как имеется порядка $O(n)$ вхождений образца в текст, то для каждого вхождения потребуется посимвольное сравнение за $O(n)$ операций, что и приведет к квадратичному времени.

Поэтому для рассмотренных задач поиска наибольшей общей подстроки решение с помощью бинарного поиска и алгоритма Рабина-Карпа в худшем случае может потребовать $O(n^2\,log(n))$ операций.

3.7. ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ ЗАДАЧИ ПОИСКА НАИБОЛЬШЕЙ ОБЩЕЙ ПОДСТРОКИ

Как уже было отмечено, задача поиска наибольшей общей подстроки используется для обнаружения дублирующего кода в программных проектах.

Задача поиска наибольшей общей подстроки для двух строк может также использоваться для поиска копирования (плагиата) на уровне исходных кодов и текстовых работ.

Для поиска плагиата в тексте в качестве первой входной строки выступает сама работа (реферат, дипломная работа), вторая строка представляет собой соединенные в одну строку все известные печатные работы. Находим наибольшую общую подстроку s_{max}. Если s_{max} не заключена в кавычки (т. е. не является цитатой) и $|s_{max}|$ больше некоторой константы (например, 100 символов или 2 предложения), то это является потенциальным плагиатом. Иначе удаляем s_{max} из первой строки и рекурсивно продолжаем искать для работы с удаленной цитатой. Таким образом мы гарантируем, что все совпадающий части будут заключены в кавычки (оформлены как цитаты) или что размер наибольшего совпадения с существующими работами не более заданной

константы[8] .

Задача поиска наибольшей общей подстроки для k строк имеет массу приложений в биоинформатике [3], задача для частных случаев $k = 1$ или $k = 2$ биоинформатикам менее интересны. А. Леск [18] пишет: «Одна или две гомологические последовательности шепчут ... полное множественное выравнивание кричит во весь голос".

3.8. ПОИСК ДУБЛИРУЮЩЕГО КОДА НА ОСНОВЕ ИЗОМОРФИЗМА ДЕРЕВЬЕВ

Поиск дублирующего кода не справляется с переименованиями (переменных, функций, методов), так как исходный код программы рассматривается как простая строка.

Имеются более интеллектуальные методы поиска дублирующего кода.

При синтаксическом анализе программы строится абстрактное синтаксическое дерево (*AST*, *abstract syntax tree*). Задача обнаружения дублирования программной логики сводится к поиску изоморфизма поддеревьев.

Задача изоформизма подграфу является *NP*-трудной [4]. Определение сложностного класса задачи изоморфизма графов является открытой проблемой.

Изоморфизм деревьев с выделенным корнем строится за линейное время [1], без выделенного корня – за $O(n^2)$ (подвешиваем первое дерево за любую вершину, для второго дерева перебираем корень). Задача изоморфизма поддереву решается в качестве подзадачи при поиске изоморфизма деревьев и так же имеет линейную асимптотику.

Подход к поиску дублирующего кода на основе изоморфизма деревьев используется в системе *Conqat*.

---

[8] *Известен анекдотический случай, когда студент заменил все символы 'o' (в русской раскладке) на 'o' (в английской раскладке), что позволило пройти тест на списывание.*

3.9. ИТОГИ

В этой главе мы нашли применение построенных алгоритмов для индексации имен файлов в файловой системе, строковых записей в базе данных (имеется возможность удалять, изменять и добавлять объекты).

Решили задачу поиска наибольшей общей подстроки как для частных случаев (наибольшая общая подстрока, наибольшая общая подстрока для двух строк), так и для для k строк. Эти задачи решены для динамического случая, т. е. в процессе изменения строк производится запрос на поиск наибольшей общей подстроки. Ответ на запрос происходит гораздо быстрее, чем решение задачи для статического варианта задачи.

4. Заключение

В работе построены алгоритмы последовательного построения суффиксного массива, блочного построения суффиксного массива, удаления блока из суффиксного массива.

В алгоритме последовательного построения суффиксного массива производится модификация суффиксного массива при добавлении одного символа к текущей строке. Асимптотика алгоритма составляет для lcp-естественных языков $O(log^2(n))$, для lcp-регулярных – $O(n\,log^2(n))$. Последовательное добавление m символов для lcp-регулярных строк в худшем случае требует $O(m^2\,log^2(n))$ операций.

В алгоритме блочного построения суффиксного массива производится модификация суффиксного массива при добавлении строки. Добавление строки размером m требует $O(m\,log^2(n))$ операций.

В алгоритме удаления блока из суффксного массива производится модификация суффиксного массива при удалении подстроки. Удаление подстроки размером m требует $O(m\,log^2(n))$ операций.

Во всех построенных алгоритмах затраты на память составляют $O(n)$.

Построенные алгоритмы позволяют работать с потоковыми данными.

Для всех построенных алгоритмов имеется возможность препроцессинга.

При каждом изменении строки мы поддерживаем массив наибольших префиксов lcp.

В работе решена задача о наибольшей общей подстроки (для одной строки, для двух строк и общая задача для k строк) для динамического случая.

Построено практическое применение для индексации имен файлов в файловой системе и текстовых записей в базах данных.

Литература

1. АХО А., ХОПКРОФТ Д., УЛЬМАН Д.. *Структуры данных и алгоритмы.* – Вильямс, 2003.
2. ВИРТ Н. *Алгоритмы и структуры данных.* – СПб.: Невский диалект, 2008.
3. ГАСФИЛД Д. *Строки, деревья, последовательности в алгоритмах.* – СПб.: Невский Диалект; БХВ-Петербург, 2003.
4. КАСЬЯНОВ В. Н., ЕВСТИГНЕЕВ В. А.. *Графы в программировании: обработка, визуализация и применение.* – СПб.: БХВ-Петербург, 2003.
5. КОРМЕН Т., ЛЕЙЗЕРСОН Ч., РИВЕСТ Р. *Алгоритмы: построение и анализ.* – М.: МЦНМО, 1999.
6. ПРЕПАРАТА Ф., ШЕЙМОС М. *Вычислительная геометрия: введение.* – М.: Мир, 1989.
7. СЛИСЕНКО А. О. *Нахождение в реальное время всех периодичностей в слове* // Доклады АН СССР. – 1980. – Т. 251. – С. 48–51.
8. СЛИСЕНКО А. О. *Поиск периодичностей и идентификация подслов в реальное время* // Зап. научн. сем. ЛОМИ. – 1981. – Т. 5. – С. 62–173.

9. СЛИСЕНКО А. О. *Распознавание предиката симметрии многоголовчатыми машинами Тьюринга со входом* // Труды Мат. института АН СССР. – 1973. – Т. 129. – С. 30–202.
10. СМИТ Б. *Методы и алгоритмы вычислений на строках.* – Вильямс, 2006.
11. AHO A., CORASICK M. *Efficient string matching: an aid to bibliographic search* // Comm. ACM. – 1975. – Vol. 18. – P. 333–340.
12. BAASE S. *Computer Algorithms. 2nd ed.* – Reading, MA: Addison-Wesley, 1988.
13. BOYER R. S., MOORE J. S. *A fast string searching algorithm* // Comm. ACM. – 1977. – Vol. 20. – P. 762–772.
14. CHARRAS C., LECROQ T. *Handbook of Exact String-Matching Algorithms*, 2004.
15. FARACH M. *Optimal suffix tree construction with large alphabets* // In Proc. IEEE 38th Annual Symposium on Foundations of Computer Science. – 1997. – P. 137–143.
16. FARACH M., MUTHUKRISHNAN S. *Optimal logarithmic time randomized suffix tree construction* // In Proc. IEEE 23th International Conference on Automata, Languages and Programming. – 1996. – P. 550–561.
17. FARACH M., FERRAGINA P., MUTHUKRISHNAN S. *On the sorting-complexity of suffix tree construction* // J. ACM. – 2000. – Vol. 47(6). – P. 987–1011.
18. HUBBARD T. J. P., LEST A. M., TRAMONTANE A. *Gathering item into the fold* // Nature Structural Biology. – April, 1996. – Vol. 4. – P. 313.
19. KARKKAINEN J., SANDERS P., *Simple Linear Work Suffix Array Construction* // Proceedings of the 30th International Colloquium on Automata, Languages and Programming, LNCS 2719, Springer, 2003. – P. 943–955.
20. KARKKAINEN J., *Fast BWT in Small Space by Blockwise Suffix Sorting* // Theoretical Computer Science, 2006.
21. KASAI T., LEE G., ARIMURA H., ARIKAWA S., PARK K. *Linear-Time Longest-Common-Prefix Computation in Suffix*

Arrays and Its Applications // CPM. – 2001. – P. 181–192.

22. KITAJIMA J. P., NAVARRO G., RIBEIRO B., ZIVIANI N. *Distributed Generation of Suffix Arrays: a Quicksort-Based Approach* // Proceedings of the 4th South American Workshop on String Processing, Valparaiso, Chile, 1997. – P. 53–59.
23. KARP R., RABIN M. *Efficient randomized pattern matching algorithms* // IBM J. Res. Development. – 1987. – Vol. 31. – P. 246–260.
24. KNUTH D. E., MORRIS J. H., PRATT V. B. *Fast pattern matching in strings* // SIAM Journal on Computing. – 1977. – Vol. 6. – P. 323–350.
25. KNUTH D. E. *The Art of Computer Programming*, Volume 2: Sorting and Searching, Second Edition. Addison-Wesley, 1997.
26. MANBER U., MAYERS G.. *Suffix arrays: a new method for on-line string searches* // SIAM Journal on Computing. – 1993. – №22. – P. 953–948.
27. MCCREIGHT E. M.. *A Space-Economical Suffix Tree Construction Algorithm* // Journal of the ACM. – 1976. – Vol. 23(2). – P. 262–272.
28. NAVARRO G., KITAJIMA J. P., RIBEIRO B., ZIVIANI N., *Distributed Generation of Suffix Arrays* // Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, LNCS 1264. - 1997. – P. 102–115.
29. RYTTER W. *A correct preprocessing algorithm for Boyer-Moore string searching* // SIAM J. Comput. – 1980. – Vol. 9. – P. 509–512.
30. SALSON M., LECROQ T., LEONARD M., MOUCHARD L. *Dynamic extended suffix arrays* // Journal of Discrete Algorithms. – 2010. – Vol. 8. – P.241-257.
31. SHIBUYA T., KUROCHKIN I. *Match chaining algorithm for cDNA Mapping* // Algorithms in Bioinformatics: Third International Workshop, Budapest, WABI, 2003.
32. UKKONEN E. *On-line construction of suffix trees* // Algorithmica. – 1995. – Vol. 14(3). – P. 249–260.

33. WEINER P. *Linear pattern matching algorithm* // 14th Annual IEEE Symposium on Switching and Automata Theory. – 1973. – P. 1–11.

SYMBOL ARRAY PROCESSING

Pavel Ajtkulov, Udmurt State University, Izhevsk, postgraduate (ajtkulov@gmail.com).

Abstract: A suffix array for a string is a data structure, which allows searching all occurrences of the sample in linear time on the sample length. We build the algorithms for modifying a suffix array by adding one character, by adding blocks to the original string, and by removing the block from the string. We suggest applying these algorithms to index text entries in databases and file names in a file system. We also develop the algorithm for online search of the longest common substring in k-strings.

Keywords: string matching, suffix array, longest common substring .

Статья представлена к публикации
членом редакционной коллегии В. Г. Лебедевым