Лекция по суффиксному массиву

Михаил Тихомиров, Александр Останин

3 марта 2015

1 Суффиксное дерево. Определение и простые свойства

1.1 Сжатый бор

Построим бор, содержащий некоторый набор слов $s_1, \ldots, s_k$. Количество вершин бора может достигать суммарной длины слов $|s_1|+\ldots+|s_k|$ (в зависимости от размера алфавита выгодно хранить внутри каждой вершины простой массив ссылок для всех возможных букв, либо ассоциативный массив только с существующими ссылками). Для сокращения количества вершин применим следующую оптимизацию: рассмотрим цепочку вершин бора, такую что из каждой вершины исходит единственное ребро в следующую. Сожмем такую цепочку в одно ребро, а вместо буквы напишем на нем всю последовательность букв с ребер, которые мы заменили. Более того, мы можем заметить, что эта последовательность букв обязательно является подстрокой некоторой строки s_i из набора, поэтому запишем на ребре только номер строки, а также начало и конец соответствующей подстроки.

После сжатия всех цепочек в боре мы получим

Определение. *Сжатый бор* (англ. *compressed trie*) — это корневое дерево, на каждом ребре которого написана непустая строка, обладающее следующими свойствами:

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

Утверждение. Количество вершин в сжатом боре составляет $O(k)$, где k — количество строк в наборе.

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

Сжатый бор занимает $O(k)$ памяти, однако, для операций с ним необходимо явно хранить все строки s_i, поэтому по памяти мы не выиграли (*убедитесь, что операции добавления и проверки слова все так же реализуются со сжатым бором за линейное время*). Зачем же он тогда нужен?..

1.2 Суффиксное дерево и его применения

Определение. *Суффиксным деревом строки s* (англ. *suffix tree*) называется сжатый бор, построенный на всех суффиксах s.

Мы уже знаем, что такой бор будет занимать $O(|s|)$ памяти. Однако, в этом случае явное хранение всех суффиксов по отдельности не требуется: они все есть в строке s. Это значит, что имея суффиксное дерево, мы умеем отвечать на запросы «является ли строка t суффиксом s», а также «является ли строка t префиксом суффикса, т.е. подстрокой s» за время $O(|t|)$ онлайн!

На самом деле, основная польза от суффиксного дерева заключается не в этом применении (для него есть более удобные решения), а в том, что оно позволяет получать много информации о строке s и ее подстроках. Вот несколько примеров.

- Чему равно количество различных подстрок строки s? Если спуститься в суффиксном дереве по пути, соответствующему подстроке, мы окажемся либо в вершине, либо посередине ребра (т.е. будет пройдена только часть подстроки, соответствующей ребру). Легко видеть, что количество различных подстрок s равно количеству различных *позиций* внутри суффиксного дерева. Количество различных позиций равно сумме длин подстрок, написанных на ребрах, плюс один (положение в корне = пустая подстрока). Значит, имея построенное суффиксное дерево для строки, мы очень просто умеем получать количество различных подстрок.

- Пусть у нас есть две подстроки строки s. Как определить длину их наибольшего общего префикса? Если опять смотреть на пути в дереве, видно, что общему префиксу двух строк сооветствует общий участок двух путей, идущих от корня. В терминах дерева можно сказать, что общему префиксу соответствует позиция в дереве, являющаяся *наименьшим общим предком (англ. least common ancestor, LCA)* двух положений для исходных подстрок. Зная позиции в дереве, соответствующие подстрокам, мы можем вычислять наименьшего общего предка при помощи любого из стандартных алгоритмов (двоичные подъемы либо оффлайновый алгоритм Тарьяна).

- Пусть мы хотим лексикографически упорядочить суффиксы строки s. Запустим обход суффиксного дерева, который в каждой вершине перебирает исходящие ребра в лексикографическом порядке первой буквы ребра. Легко видеть, что такой обход будет перебирать позиции в дереве в порядке лексикографического возрастания соответствующих строк. Отсюда следует, что порядок посещения обходом листьев (т.е. позиций, соответствующих суффиксам), и есть их лексикографическая сортировка. Этот порядок, построенный на суффиксах строки s, называется *суффиксным массивом* (англ. *suffix array*) для строки s.

Комментарий: вообще говоря, в суффиксном дереве не любому суффиксу будет соответствовать лист дерева; если суффикс имеет более двух вхождений в строку s как подстрока, то из его позиции можно пойти куда-то вниз. Чтобы всем суффиксам соответствовали листья, допишем к строке s в конце символ \$*, который не совпадает ни с каким другим символом строки. В дальнейшем будем считать, что строка s заканчивается на* \$.

Суффиксный массив сам по себе несет несколько меньше информации, чем суффиксное дерево. Однако, поговорить про него имеет смысл, потому что:

а) эффективные алгоритмы для построения суффиксного дерева достаточно сложны, для построения суффиксного массива существуют более простые алгоритмы (хотя и чуть менее быстрые);

б) для многих задач, для которых существует решение при помощи суффиксного дерева, существует и непосредственное решение при помощи суффиксного массива;

в) при необходимости по построенному суффиксному массиву строки (и самой строке) можно восстановить и суффиксное дерево.

2 Построение суффиксного массива. Алгоритм Касаи. Простейшие применения

2.1 Построение

Будем сортировать циклические сдвиги строки s; если считать, что символ `$` лексикографически меньше любого другого символа, то порядок сортировки будет таким же, что и порядок сортировки суффиксов.

Алгоритм будет состоять из нескольких шагов. После k-ого шага алгоритм будет получать лексикографический порядок для подстрок циклической строки $\overline{s} = s + s + \ldots$, имеющих длину 2^k и начинающихся в позициях $0, \ldots, |s| - 1$. Помимо лексикографического порядка подстрок, алгоритм будет поддерживать информацию о *классах эквивалентности* этих подстрок. В классе эквивалентности лежат все попарно равные подстроки текущей длины; классы эквивалентности можно упорядочить лексикографически и пронумеровать в этом же порядке. Для каждой подстроки алгоритм также будет поддерживать номер ее класса эквивалентности при лексикографическом упорядочивании (теперь результат лексикографического сравнения подстрок совпадает с результатом сравнения номеров соответствующих классов).

Например, после первого шага алгоритма для строки `abcabc$` мы имеем следующие массивы:

$array = (6, 3, 0, 4, 1, 5, 2)$ (подстроки длины 2 упорядочены лексикографически, в массиве записаны позиции начал этих подстрок; равные подстроки могут идти в любом порядке)

$classes = (1, 2, 4, 1, 2, 3, 0)$ (i-ый элемент равен номеру класса эквивалентности, в котором находится подстрока длины 2, начинающаяся в i-ом символе; равным строкам соответствуют одинаковые номера)

Нулевой шаг алгоритма выполнить легко: достаточно найти лексикографически порядок отдельных символов строки.

Пусть мы хотим выполнить $(k+1)$-ый шаг, имея результат k-ого. Как сравнить две подстроки циклической строки $\overline{s}$ длины 2^{k+1}? Если посимвольное лексикографическое сравнение для этих подстрок закончится в первой половине строки, то результат сравнения будет совпадать с результат сравнения первых половин этих подстрок; в противном случае (т.е. если первые половины совпали) результат определяется сравнением вторых половин. Поскольку результат сравнения подстрок длины 2^k совпадает с результатом сравнения соответствующих элементов массива $classes$, сортировка подстрок длины 2^{k+1} сводится к сортировке упорядоченных пар вида $(classes_i, classes_{(i+2^k) \bmod n})$. После сортировки пар нам необходимо пересчитать массив $classes$; это можно сделать одним проходом по отсортированному массиву пар.

Алгоритм найдет правильный порядок суффиксов, как только в каждом классе эквивалентности будет по одному элементу. Легко видеть, что это гарантированно произойдет через $O(\log |s|)$ шагов, когда в каждой подстроке будет хотя бы один `$`. Каждый шаг выполняется за время $O(|s| \log |s|)$, если сортировать массив пар какой-нибудь обычной сортировкой, либо за время $O(|s|)$, если использовать для этого поразрядную сортировку. Итоговое время составляет $O(|s|log^2|s|)$, либо $O(|s|\log|s|)$. Детали реализации и некоторые оптимизации можно прочитать на сайте e-maxx.ru.

2.2 Нахождение наибольших общих префиксов соседних суффиксов (алгоритм Арикавы-Аримуры-Касаи-Ли-Парка)

Определим функцию $lcp(i,j)$ как длину наибольшего общего префикса суффиксов строки s, начинающихся в позициях i и j. Пускай у нас построен суффиксный массив $array$ для строки s. Тогда верно следующие утверждение:

$$lcp(array_i, array_j) = min(lcp(array_i, array_{i+1}), \ldots, lcp(array_{j-1}, array_j).$$

Действительно, представим, что $lcp(array_i, array_j) \geqslant k$, тогда во всех суффиксах $array_{i+1}$, $\ldots array_{j-1}$ первые k символов такие же, как и в $array_i$ и $array_j$, иначе где-то нарушится лексикографический порядок.

Это означает, что если мы найдем значения $lcp(array_i, array_{i+1})$ для всех i, задача нахождения $lcp(array_i, array_j)$ для произвольных i и j сведется к задаче RMQ. Для краткости будем обозначать $lcp_i = lcp(array_i, array_{i+1})$.

Будем вычислять lcp_i в порядке уменьшения длины суффикса $array_i$; обозначим дополнительно p_k позицию в массиве $array$ суффикса, начинающегося в позиции k. Для суффикса, равного всей строке s, вычислим значение lcp_{p_0} явно посимвольным сравнением с суффиксом, начинающимся в позиции $array_{p_0+1}$.

Теперь вычислим $lcp_{p_{k+1}}$, зная значение $lcp_{p_k} = lcp(k, array_{p_k+1})$. Если $lcp_{p_k} = 0$, вычислим $lcp_{p_{k+1}}$ посимвольным сравнением. Пусть $lcp_{p_k} \geqslant 1$; тогда $lcp(k, array_{p_k+1}) = lcp(k+1, array_{p_k+1}+1)+1$, поскольку вторая пара суффиксов получается из первой отрезанием первого символа от обоих суффиксов. Кроме этого, поскольку суффикс k лексикографически меньше суффикса $array_{p_k+1}$, суффикс $k+1$ также меньше суффикса $array_{p_k+1}+1$. Отсюда $lcp(k+1, array_{p_k+1}+1) = min(lcp(k+1, array_{p_{k+1}+1}), \ldots) \leqslant lcp_{k+1}$; т.е. $lcp_{k+1} \geqslant lcp_k - 1$. Иными словами, имеет смысл начинать посимвольное сравнение для вычисления lcp_{k+1} не с начала строки, а с позиции $lcp_k - 1$.

Очевидно, алгоритм находит массив lcp_k правильно. Какова сложность работы этого алгоритма? Применим амортизационный анализ: после каждого успешного посимвольного сравнения значение lcp_k увеличивается на 1, на каждом шаге $lcp_{k+1} \geqslant max(lcp_k - 1, 0)$, и $lcp_k \leqslant n$ для любого k; (суммарное количество сравнений) = (количество успешных сравнений) + (количество неуспешных сравнений) $\leqslant$ (n + количество уменьшений lcp_k) + (n) = $O(n)$. Итак, приведенный алгоритм строит массив lcp_k за линейное время (при условии того, что суффиксный массив уже построен).

2.3 Количество различных подстрок

Вспомним, что суффиксный массив — это лексикографический порядок обхода листьев в суффиксном дереве. Посчитаем количество различных подстрок в дереве другим способом: сперва пройдем путь от корня до первого листа в порядке обхода, на этом пути мы пройдем $|s_0|$ различных позиций, где s_0 — первый суффикс в суффиксном массиве. Теперь пойдем ко второму листу: поднимемся на высоту $lcp(s_0, s_1)$, и спустимся ко второму листу. Новые позиции будут пройдены только на той части пути, где мы спускаемся ко второму листу, поэтому новых позиций в дереве мы посетим $|s_1| - lcp(s_0, s_1)$. Рассуждая тем же образом для остальных листьев, получим, что количество различных позиций в дереве (а значит. и количество различных подстрок) равно $\sum |s_i| - \sum lcp(s_i, s_{i+1})$.

2.4 Поиск подстроки в строке

Сначала научимся проверять вхождение строки t в s за время $O(|t| \cdot \log n)$, где $n = |s|$. Поскольку на строках у нас задан лексикографический порядок, можно запустить бинарный поиск и найти первый в порядке суффиксного массива суффикс p_i, который лексикографически не меньше чем строка t. Тогда подстрока t входит в s, если и только если $lcp(p_i, t) = |t|$. Сравнивать лексикографически t и суффикс можно за $O(|t|)$, поэтому требуемая асимптотика достигнута.

Теперь улучшим наш алгоритм до асимптотики $O(|t| + \log n)$. В каждый момент у нас есть левая и правая граница бинарного поиска l, r, будем поддерживать для них числа $prefL = lcp(p_l, t)$ и $prefR = lcp(p_r, t)$. Пусть $m = (l + r)/2$, теперь мы должны сравнить лексикографически суффикс p_m и строку t. Пусть, без ограничения общности, $prefL \geqslant prefR$. Тогда если $lcp(p_l, p_m) < prefL$, то можно сразу сдвинуть правую границу: $r = m$, $prefR = lcp(p_l, p_m)$, так как в таком случае искомый суффикс точно лежит в отрезке между p_l и p_m. Если же $lcp(p_l, p_m) \geqslant prefL$, то, поскольку $lcp(p_l, t) = prefL$, то $lcp(p_m, t) \geqslant prefL$. Тогда найдем $lcp(p_m, t)$, стартовав с $prefL$. После этого мы узнаем лексикографический порядок между p_m и t. Получается, что для сравнения p_m и t мы сделали не больше чем $lcp(p_m, t) - prefL + 1$ шагов. Очевидно, что на следующем шаге максимальное среди чисел $prefL, prefR$ станет равно $lcp(p_m, t)$. Значит, на этой итерации мы сделали не больше шагов, чем увеличился $max(prefL, prefR)$ при переходе на следующую итерацию. Значит, суммарно при таком процессе затраченное время на сравнение суффиксов и t будет $O(|t|)$. Поскольку происходит $O(\log n)$ итераций бинпоиска, итоговая асимптотика $O(|t| + \log n)$.

3 Построение суффиксного дерева по суффиксному массиву

Будем идти по суффиксному массиву слева направо и поддерживать указатель на позицию в дереве последнего рассмотренного суффикса. Берём lcp с новым суффиксом: теперь нам надо подняться по дереву так, чтобы глубина вершины совпала с этим lcp. Разрезаем ребро, ставим новую вершину и проводим из неё ребро вниз с меткой, соответствующей остатку нового суффикса, из которого выкинули общий префикс с предыдущим суффиксом. Переставляем указатель на конец нового ребра. В итоге по каждому ребру суффиксного дерева мы пройдём не более двух раз: вниз и вверх. Поэтому такой алгоритм строит суффиксное дерево по суффиксному массиву за $O(n)$.

4 Решение строковых задач с помощью суффиксного дерева и суффиксного массива

4.1 Максимальная по длине строка, имеющая два непересекающихся вхождения

Решение с помощью суффиксного дерева. В суффиксном дереве мы можем посчитать для каждой вершины максимальный и минимальный по индексу начала суффикса лист в поддереве. Это делается простой динамикой по дереву. Далее для каждой вершины v ответ нужно прорелаксировать минимумом из этой разности для вершины v и длины строки, соответствующей пути от корня до v. Решение за $O(n)$.

Решение с помощью суффиксного массива. Чтобы решить задачу с помощью суффиксного массива, можно сначала сделать бинпоиск по ответу. Пусть мы хотим понять, существует ли такая строка длины k. Пройдемся по суффиксному массиву двумя указателями l, r, поддерживая их так, чтобы lcp суффиксов p_l, p_r был больше либо равен k. При этом будем поддерживать set начал суффиксов, соответствующих текущему отрезку, т.е. $p_l, p_{l+1}, \ldots, p_r$. Для отрезка $[l; r]$ находим максимум и минимум в set-е, если разность между ними как минимум k, то нужная строка длины k существует. Иначе сдвигаем l на единицу вправо, далее сдвигаем r, пока lcp суффиксов p_l, p_r не меньше k, и продолжаем процесс. Получаем решение за $O(n \log^2 n)$. Если вместо set-а использовать два deque-а, в одном из которых будут храниться суффиксы в убывающем по индексам порядке (если индекс нового суффикса меньше индекса последнего суффикса в очереди, новый не добавляем; иначе же удаляем элементы с конца deque, пока новый индекс больше, чем последний в deque; после этого вставляем новый индекс в конец), а в другом — в возрастающем (то же самое), то алгоритм внутри бинпоиска будет выполняться за линейное время. Получим решение за $O(n \log n)$.

4.2 Рефрен строки

Рефреном строки s называется подстрока q, для которой произведение $|q|$ на количество её вхождений максимально. Вхождения при этом могут пересекаться.

Решение с помощью суффиксного дерева. В суффиксном дереве можно посчитать для каждой вершины v количество листьев $leaves[v]$ в её поддереве. Чтобы найти рефрен, нужно найти ту вершину, для которой произведение $leaves[v] \cdot path[v]$ максимально, где $path[v]$ — длина строки, соответствующей пути от корня до v. Получаем решение за $O(n)$.

Решение с помощью суффиксного массива. В терминах суффиксного массива задачу можно переформулировать так: среди отрезков $[l; r],\ 1 \le l \le r \le |s|$ нужно найти отрезок с максимальным значением

$$(r - l + 1) \cdot \min_{i=l,l+1,\ldots,r-1} lcp(p_i, p_{i+1}).$$

Тогда префикс суффикса $p[l]$ длины $\min\limits_{i=l,l+1,\ldots,r-1} lcp(p[i], p[i+1])$ является префиксом всех суффиксов $p_l, p_{l+1}, \ldots, p_r$, а значит входит в строку s как минимум $r - l + 1$ раз.

Чтобы найти максимум такой величины, будем поддерживать структуру данных, в которую будем добавлять индексы i в порядке увеличения $lcp(p_i, p_{i+1})$. Тогда при добавлении очередного индекса со значением lcp, равным k, надо найти ближайший слева l и ближайший справа r индексы, которые были добавлены раньше, и прорелаксировать ответ величиной $k \cdot (r - l)$, так как в таком случае все суффиксы $p_{l+1}, p_{l+2}, \ldots p_r$ имеют общий префикс длины k. Получаем решение за $O(n \log n)$.

4.3 Нахождение подпалиндромов

Пусть мы хотим для каждого i найти максимальный радиус чётного и нечётного палиндрома с центром в i (если палиндром нечетный, то центр — его середина, если четный, то центр — элемент слева от середины).

Решение с помощью суффиксного дерева. Построим суффиксное дерево для строки $s\#s^R$. Для каждого i найдем вершины (или место внутри ребра), соответствующие индексам i и $2 \cdot |s| - i$ (в 0-нумерации) — обозначим их u и v. Путь от корня до вершины $lca(u, v)$ задаёт максимальную половину нечётного палиндрома. Для нахождения радиуса максимального чётного палиндрома с центром в i нужно рассмотреть индексы i и $2 \cdot |s| - i - 1$. Получаем решение за $O(n \log n)$.

4.4 Наибольшая общая подстрока нескольких строк

Пусть даны строки $s_1, s_2, \ldots, s_k$. Нужно найти максимальную по длине строку t, входящую как подстрока в $s_1, s_2, \ldots, s_k$.

Решение с помощью суффиксного дерева. Построим суффиксное дерево для строки $s_1 a_1 s_2 a_2 \ldots s_{k-1} a_{k-1} s_k a_k$, где $a_1, a_2, \ldots a_k$ — различные разделители, т.е. символы, не встречающиеся в строках $s_1, s_2, \ldots, s_k$. После построения дерева выкинем все рёбра, ведущие из вершин, последняя метка пути от корня к которым является разделителем. Теперь те вершины, от которых достижимы все разделители, соответствуют строкам, которые входят во все k строк как подстроки. Решим эту задачу для $k = 1$. Поставим в листья дерева, в которые ведут ребра с разделителем, единицы. Отсортируем эти листья в порядке обхода: $v_1, v_2, \ldots, v_m$. Теперь добавим минус единицу в вершины $lca(v_1, v_2), lca(v_2, v_3), \ldots, lca(v_{m-1}, v_m)$ и посчитаем для каждой вершины сумму в поддереве. Тогда сумма в поддереве вершины равна 1, если в ней есть хотя бы одна помеченная вершина (так как в поддереве лежат все помеченные листья и все вершины $lcp(v_i, v_{i+1})$), и равна 0 в противном случае. Теперь можно сделать то же самое для каждого из k разделителей и посчитать сумму в поддеревьях. Теперь вершина суффиксного дерева соответствует общей подстроке всех строк тогда и только тогда, когда сумма в поддереве равна k. Получаем решение за $O(n \log n)$ (а если использовать алгоритм Тарьяна для поиска lca, будет $O(n\alpha)$).

Решение с помощью суффиксного массива. Построим суффиксный массив для строки $s_1 a_1 s_2 a_2 \ldots s_{k-1} a_{k-1} s_k a_k$, где $a_1, a_2, \ldots a_k$ — различные разделители. Для каждого суффикса запомним самый левый разделитель. Теперь пройдёмся по суффиксному массиву двумя указателями $[l; r]$ так, чтобы среди $p_l, p_{l+1}, \ldots, p_r$ были суффиксы каждой строки s_i. При сдвиге указателей количество строк, суффиксы которых есть в текущем отрезке, пересчитываются за $O(1)$, если хранить массив с количеством суффиксов в текущем отрезке для каждой из строк. Тогда для отрезка $[l; r]$, удовлетворяющего описанному условию, нужно прорелаксировать ответ величиной $lcp(p_l, p_r)$. Это можно делать с использованием sparse table или дерева отрезков. Если обозначить $n = \sum\limits_{i=1}^{k} |s_i|$, то получаем решение, работающее $O(n \log n)$ времени и $O(n)$ (дерево отрезков) или $O(n \log n)$ памяти.

4.5 Задача про слабые продолжения (Moscow SU Tapirs Contest 1, Petrozavodsk Winter 2014)

Пусть дана строка s. Скажем, что строка *y (сильно) продолжает* строку x, если после каждого вхождения x в s следует вхождение y.

Скажем, что строка *y слабо продолжает* строку x, если после каждого вхождения x либо идет вхождение y, либо длина оставшегося суффикса меньше, чем $|y|$.

Требуется найти количество пар (x, y) подстрок s таких, что y слабо продолжает x.

Решение с помощью суффиксного дерева. Построим суффиксное дерево для строки $s\#$.

Заметим, что для строки u количество сильных продолжений — это просто количество символов, которые надо пройти от места на ребре, соответствующего u, до следующей вершины. Осталось посчитать количество слабых продолжений за исключением сильных продолжений, назовем такие продолжения *очень слабыми*.

Рассмотрим вершину v, пусть её предок p. Тогда для всех префиксов строки, соответствующих пути от корня до v, которые длиннее чем строка, соответствующая пути от корня до p, если взять их в качестве x, то количество очень слабых продолжений x будет для них одинаково. Поэтому осталось найти количество очень слабых продолжений для строки, соответствующей вершине v, тогда мы сможем пересчитать нужную величину для всех строк, соответствующих ребру (p, v).

Будем считать глубиной вершины количество символов на пути от корня к этой вершине. Пусть самый глубокий лист в поддереве v имеет глубину h_1, а следующий по глубине лист — h_2. Тогда если $h_1 = h_2$, то у текущей строки нет очень слабых продолжений. Иначе ответ для v равен просто $h_1 - h_2$: для каждой глубины h от $h_2 + 1$ до h_1 мы можем приписать только одну строку к текущей строке так, чтобы итоговая длина равнялась h и итоговая строка была подстрокой s.