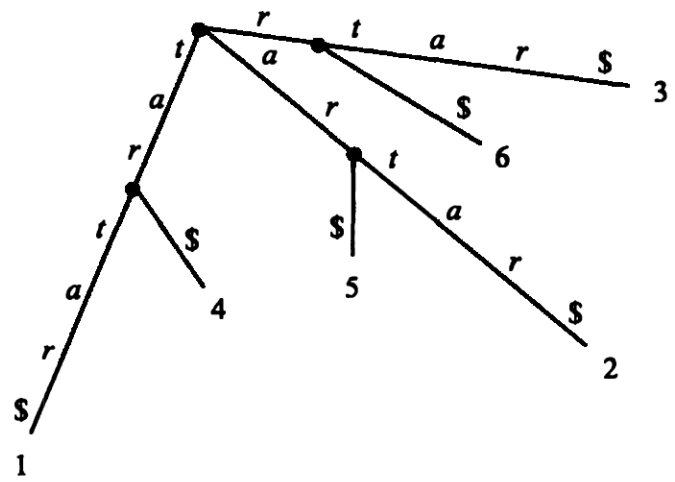

Рис. 7.5. Лексический обход суффиксного дерева в глубину посещает листья в порядке 5, 2, 6, 3, 4, 1

В качестве детали реализации отметим, что если ветви, выходящие из каждой вершины дерева, организованы в *упорядоченный* цепной список (как об этом говорилось в п. 6.5 на с. 152), то издержки на лексический поиск в глубину будут такими же, как на любой другой поиск в глубину. Каждый раз, когда этот поиск должен выбирать для перехода дугу, выходящую из v, он берет следующую дугу из цепного списка этой вершины.

7.14.2. Как искать образец, используя суффиксный массив

Суффиксный массив для строки T позволяет реализовать очень простой алгоритм поиска всех вхождений любого образца P в T. Здесь ключевым является тот факт, что если P входит в T, то все места этих вхождений будут расположены в Pos рядом. Например, $P = issi$ встречается в $mississippi$, начиная с позиций 2 и 5, которые в Pos стоят рядом (см. рис. 7.4). Таким образом, поиск вхождений P в T просто выполняет двоичный поиск в суффиксном массиве. Рассуждая подробнее, предположим, что P лексически меньше, чем суффикс в средней позиции Pos (т.е. суффикс $Pos(\lceil m/2 \rceil)$). В этом случае первое место в Pos, содержащее такую позицию, где P входит в T, должно быть в первой половине Pos. Аналогично, если P лексически больше, чем суффикс $Pos(\lceil m/2 \rceil)$, то места, где P входит в T, должны находиться во второй половине Pos. Следовательно, используя двоичный поиск, можно найти в Pos наименьший индекс i (если он существует), для которого P в точности совпадает с первыми n символами суффикса $Pos(i)$. Аналогично можно найти и наибольший индекс с таким свойством i'. Итак, образец P входит в T, начиная с любого места, задаваемого числами от $Pos(i)$ до $Pos(i')$.

Лексическое сравнение P с любым суффиксом занимает время, пропорциональное длине общего префикса этих двух строк. Она не превосходит n, следовательно:

Теорема 7.14.2. *При использовании двоичного поиска в массиве Pos все вхождения P в T могут быть найдены за время* $O(n \log m)$.

Конечно, истинное поведение алгоритма зависит от того, сколько длинных префиксов P встречается в T. Если их очень мало, то конкретное лексическое сравнение

действительно потребует времени $\Theta(n)$, и в общем граница $O(n \log m)$ будет довольно пессимистичной. В "случайных" строках (даже при больших алфавитах) этот метод работал бы за ожидаемое время $O(n + \log m)$. Если в T встречается много длинных префиксов P, метод можно улучшить с помощью двух приемов, описанных в следующих двух пунктах.

7.14.3. Простой ускоритель

В случае двоичного поиска обозначим через L и R, соответственно, левую и правую границы "текущего интервала поиска". В момент старта имеем $L = 1$ и $R = m$. На каждой итерации двоичного поиска проверка идет в позиции $M = \lfloor (R + L)/2 \rfloor$ массива Pos. Поисковый алгоритм помнит наибольшие префиксы $Pos(L)$ и $Pos(R)$, совпадающие с префиксом P. Пусть l и r обозначают, соответственно, длины этих префиксов. Положим $mlr = \min(l, r)$.

Значение mlr можно использовать, чтобы ускорить лексическое сравнение P и суффикса $Pos(M)$. Так как массив Pos обеспечивает лексический порядок суффиксов T, то для любого индекса i между L и R первые mlr символов суффикса $Pos(i)$ должны быть такими же, как у суффикса $Pos(L)$ и, следовательно, у P. Поэтому лексическое сравнение P и суффикса $Pos(M)$ можно начинать с позиции $mlr + 1$, а не с начала.

Поддержание mlr во время двоичного поиска немного усложняет алгоритм, но позволяет избежать многих ненужных сравнений. В начале поиска, когда $L = 1$ и $R = m$, P явно сравнивается с суффиксами $Pos(1)$ и $Pos(m)$, что дает значения l, r и mlr. Однако наихудшее время для этого улучшенного метода остается прежним — $O(n \log m)$. Майерс и Манбер [308] сообщают, что использование только mlr позволяет ускорить поиск в практических случаях до оценки наихудшего случая $O(n + \log m)$, которую мы вначале объявили. Так что мы представим полный метод, который гарантирует эту улучшенную оценку наихудшего случая, только из-за его элегантности.

7.14.4. Сверхускоритель

Назовем проверку символа в P *избыточной*, если этот символ был уже проверен ранее. Цель ускорения заключается в уменьшении числа избыточных проверок символов до не более одной на каждую итерацию двоичного поиска — следовательно, до не более $O(\log m)$ в общей сложности. Отсюда немедленно получится желаемая граница времени $O(n + \log m)$. Привлечение к этой цели только mlr результата не обеспечивает. Так как mlr есть минимум из l и r, то при $l \neq r$ все символы в P от $mlr + 1$ до максимума из l и r будут уже проверены, и любые сравнения этих символов будут избыточны. Теперь нужно определить, как начать сравнения с максимума из l и r.

Определение. $Lcp(i, j)$ есть длина наибольшего общего префикса (longest common prefix) суффиксов, определенных позициями i и j массива Pos, т.е. суффиксов $Pos(i)$ и $Pos(j)$.

Например, когда $T = mississippi$, суффикс $Pos(3)$ — это $issippi$, а суффикс $Pos(4)$ — это $ississippi$, и $Lcp(3, 4) = 4$ (см. рис. 7.4).

Для ускорения поиска алгоритм использует значения $Lcp(L, M)$ и $Lcp(M, R)$ для каждой тройки (L, M, R), встречающейся в ходе двоичного поиска. Предположим, что мы можем получить эти значения за константное время, и покажем, как они могут помочь в поиске. Далее покажем, как вычислять конкретные значения Lcp, требуемые поиском, во время препроцессинга T.

Как использовать значения Lcp

Простейший случай. На любой итерации двоичного поиска если $l = r$, то сравнивать P с суффиксом $Pos(M)$, начиная с позиции $mlr + 1 = l + 1 = r + 1$, как раньше.

Общий случай. Когда $l \neq r$, не умаляя общности, предположим, что $l > r$. Тогда возможны три подслучая:

1. Если $Lcp(L, M) > l$, то общий префикс суффиксов $Pos(L)$ и $Pos(M)$ длиннее, чем общий префикс P и $Pos(L)$. Поэтому P совпадает с суффиксом $Pos(M)$ и за символом l. Другими словами, символ $l + 1$ у суффиксов $Pos(L)$ и $Pos(M)$ один и тот же, и он лексически меньше, чем символ $l + 1$ образца P (из-за того, что P лексически больше, чем суффикс $Pos(L)$). Следовательно, все начальные позиции вхождений P в T (если они существуют) должны находиться справа от позиции M в Pos. Итак, на любой итерации двоичного поиска, когда возникает такой случай, никаких проверок P не требуется, L просто заменяется на M, а l и r остаются неизменными (рис. 7.6).

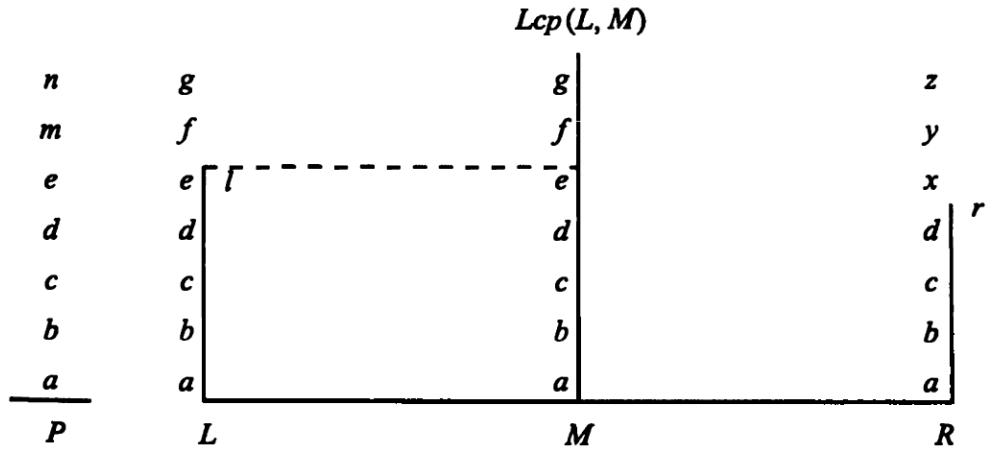

Рис. 7.6. Подслучай 1 сверхускорителя. Образец $P = abcdemn$ выписан вертикально снизу вверх. Суффиксы $Pos(L)$, $Pos(M)$ и $Pos(R)$ также выписаны вертикально. В этом случае $Lcp(L, M) > 0$ и $l > r$. Справа от M в массиве Pos должна встретиться любая стартовая позиция P в T, так как P совпадает с суффиксом $Pos(M)$ только до символа l

2. Если $Lcp(L, M) < l$, то общий префикс суффиксов $Pos(L)$ и $Pos(M)$ меньше, чем общий префикс суффикса $Pos(L)$ и P. Поэтому P совпадает с суффиксом $Pos(M)$ за символом $Lcp(L, M)$. Символ $Lcp(L, M) + 1$ у P и суффикса $Pos(L)$ один и тот же, и он лексически меньше, чем у суффикса $Pos(M)$. Следовательно, все начальные позиции вхождений P в T (если они существуют) должны находиться слева от позиции M в Pos. Итак, на любой итерации двоичного поиска, когда возникает такой случай, никаких проверок P не требуется, r заменяется на $Lcp(L, M)$, l остается неизменным, а R заменяется на M.

3. Если $Lcp(L, M) = l$, то P совпадает с суффиксом $Pos(M)$ за символом l. В этом случае алгоритм лексически сравнивает P с суффиксом $Pos(M)$, начиная с позиции $l + 1$. Таким образом, в результате этого сравнения определяется, которое из значений меняется, L или R, с соответствующим изменением l или r.

Теорема 7.14.3. *При использовании значений Lcp поисковый алгоритм делает не более $O(n + \log m)$ сравнений и работает за такое же время.*

Доказательство. Во-первых, простым анализом всех возможностей легко проверить, что во время двоичного поиска ни l, ни r убывать не могут. Кроме того, каждая итерация, которая заканчивает поиск, не проверяет ни одного символа из P или кончается после первого несовпадения, найденного на этой итерации.

В тех двух случаях ($l = r$ или $Lcp(L, M) = l > r$), когда алгоритм проверяет во время итерации символ, сравнения начинаются с символа $\max(l, r)$ образца P. Предположим, что на этой итерации проверено k символов P. Значит, обнаружено $k - 1$ совпадений, и в конце итерации $\max(l, r)$ увеличился на $k - 1$ (и это значение присвоено либо l, либо r). Следовательно, в начале любой итерации символ $\max(l, r)$ образца P может быть уже проверен, а следующий символ еще нет. Это означает, что на каждой итерации делается не более одного дублирующего сравнения. Таким образом, всего делается не более $\log_2 m$ избыточных сравнений. Число нелишних сравнений не превосходит n, что дает $n + \log m$ как оценку для общего числа. Вся остальная работа в алгоритме, очевидно, может быть выполнена за время, пропорциональное числу этих сравнений. □

7.14.5. Как получить значения *Lcp*

Значения *Lcp*, необходимые для ускорения поиска, вычисляются заранее в препроцессной фазе при создании суффиксного массива. Сначала рассмотрим, сколько значений *Lcp* нам понадобится (чтобы обеспечить все возможные пути двоичного поиска). Для удобства предположим, что m является степенью 2.

Определение. Пусть B — полное двоичное дерево с m листьями, в котором каждая вершина помечена парой целых чисел (i, j), $1 \leqslant i \leqslant j \leqslant m$. Корень B помечен парой $(1, m)$. Каждая нелистовая вершина (i, j) имеет двух детей, левый из них — с меткой $(i, \lfloor (i + j)/2 \rfloor)$ и правый — с меткой $(\lfloor (i + j)/2 \rfloor, j)$. Листья B имеют метки $(i, i + 1)$ (плюс еще один лист с меткой $(1, 1)$) и упорядочены слева направо в порядке возрастания i (рис. 7.7).

По существу, метки вершин определяют границы (L, R) всех возможных интервалов поиска, которые могут появиться при двоичном поиске в упорядоченном списке длины m. Так как B — двоичное дерево с m листьями, оно имеет в общей сложности $2m - 1$ вершин. Таким образом, нужно предварительно вычислить только $O(m)$ значений *Lcp*. Предположение, что эти значения можно накопить в препроцессинге $\mathcal{T}$ за время $O(m)$, выглядит правдоподобно. Но как именно? В следующей лемме мы покажем, что значения *Lcp* на листьях B легко накапливаются во время лексического обхода в глубину дерева $\mathcal{T}$.

Лемма 7.14.1. *При лексическом обходе в глубину дерева $\mathcal{T}$ рассмотрим внутренние вершины, проверяемые между посещениями листа $Pos(i)$ и листа $Pos(i + 1)$,*

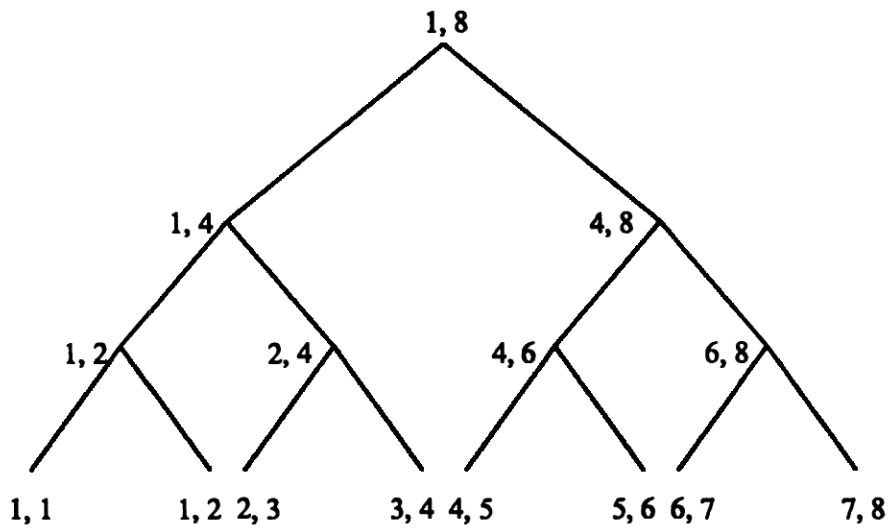

Рис. 7.7. Двоичное дерево B представляет все возможные интервалы поиска при выполнении двоичного поиска в списке длины $m = 8$

шла между i-м осмотренным листом и следующим. Среди этих внутренних вершин возьмем ближайшую к корню. Тогда $Lcp(i, i+1)$ равно ее строковой глубине.

Например, рассмотрим снова суффиксное дерево, показанное на рис. 7.5 (с. 193). $Lcp(5, 6)$ равно строковой глубине родителя листьев 4 и 1, т.е. 3, так как этот родитель помечен строкой tar. Значения $Lcp(i, i+1)$ для i от 1 до 5 равны, соответственно, 2, 0, 1, 0, 3.

Самое трудное в лемме 7.14.1 — прочесть ее. Когда это сделано, доказательство прямо следует из свойств суффиксных деревьев, так что оно оставляется читателю.

Если предположить, что строковые глубины вершин известны (а их можно найти за линейное время), то по лемме значения $Lcp(i, i+1)$ для i от 1 до $m-1$ легко вычисляются за время $O(m)$. Оставшиеся значения Lcp легко получаются согласно следующей лемме.

Лемма 7.14.2. $Lcp(i, j) = \min\{Lcp(k, k+1) \colon i \leqslant k < j\}$.

Доказательство. Суффиксы $Pos(i)$ и $Pos(j)$ строки T имеют общий префикс длины $Lcp(i, j)$. По свойствам лексикографического порядка для каждого k между i и j суффикс $Pos(k)$ должен также иметь этот общий префикс. Поэтому $Lcp(k, k+1) \geqslant Lcp(i, j)$ для каждого k между i и $j-1$.

Теперь по транзитивности $Lcp(i, i+2)$ должно быть не меньше, чем минимум из $Lcp(i, i+1)$ и $Lcp(i+1, i+2)$. Далее, $Lcp(i, j)$ должно быть не меньше, чем наименьшее из $Lcp(k, k+1)$, где k меняется от i до $j-1$. Это наблюдение в сочетании с рассуждением предыдущего абзаца доказывает лемму. □

С помощью леммы 7.14.2 оставшиеся значения Lcp для B можно найти, продвигаясь от листьев и полагая значение Lcp для любой вершины v равным минимуму из значений Lcp ее детей. Это, очевидно, требует времени только $O(m)$.

В итоге алгоритм сравнения строки и подстроки со временем $O(n + \log m)$, использующий суффиксный массив, должен заранее вычислить $2m - 1$ значений Lcp, сопоставленных вершинам двоичного дерева B. Значения, соответствующие листьям,

накапливаются за требующий линейного времени лексический просмотр дерева $\mathcal{T}$ в глубину, который используется при построении суффиксного массива. Остальные значения вычисляются по ним за линейное время проходом по дереву B снизу вверх, что дает следующую теорему.

Теорема 7.14.4. *Все необходимые значения Lcp можно получить за время $O(m)$, а все вхождения P в T можно найти, используя суффиксный массив, за время $O(n + \log m)$.*

7.14.6. Где встречаются задачи с большими алфавитами?

К работе с суффиксными массивами в значительной мере побуждают трудности использования суффиксных деревьев, если алфавит велик. Поэтому вполне естественно спросить, а когда же он велик?

Во-первых, существуют естественные языки, такие как китайский, с большими "алфавитами". Однако для нас чаще всего большой алфавит появляется, когда строка содержит числа, каждое из которых трактуется как символ. Простой пример такой ситуации — строка изображения, каждый символ которой задает цвет пикселя или градацию серого цвета.

Задачи поиска совпадений строк и подстрок, когда алфавит содержит числа, а строки P и T велики, встречаются также в вычислительных задачах молекулярной биологии. Примером может служить задача *совпадения карт* (map matching). *Карта рестриктаз* для единичного фермента определяет расположения в строке ДНК копий определенной подстроки (мест рестрикции). Каждое такое место может отделяться от следующего многими тысячами оснований. Следовательно, карта рестриктаз для единичного фермента представляется строкой из чисел, задающих расстояния между последовательными местами рестрикций. При рассмотрении этой последовательности в качестве строки каждое число является символом (огромного) базового алфавита. В более общем случае карта может отображать расположения многих различных образцов (независимо от того, являются ли они местами рестриктаз), так что строка (карта) состоит из символов некоторого конечного алфавита (представляющего известные рассматриваемые образцы), чередующихся с целыми числами, задающими расстояния между такими местами. Алфавит велик, так как велика область изменения чисел, и, так как часто расстояния известны с большой точностью, эти числа не округляются. Более того, множество известных рассматриваемых образцов и само велико (см. [435]).

Часто подстрока ДНК получается и исследуется без информации о том, где она размещается в геноме и не изучалась ли ранее. Если и новая, и ранее исследованная ДНК полностью расшифрованы и помещены в базу, то вопросы о предыдущей работе или локализациях могут быть решены точным сравнением строк. Но большинство изучаемых подстрок ДНК расшифрованы не полностью — получить карты проще и дешевле, чем полные последовательности. Ввиду этого возникает следующая задача о совпадениях *карт*, которая преобразуется в задачу о совпадениях *строк* с большими алфавитами:

> По имеющейся карте (рестриктаз) для большой строки ДНК и карте для меньшей строки определить, не является ли меньшая строка подстрокой большей.