Более сложные алгоритмы на строках

Это тоже новая глава, в ней собраны чуть более сложные алгоритмы на строках. Пока написан текст только про бор, но когда-нибудь стоит добавить и суффиксный массив (правда до этого нужно самому научиться его писать и применять).

Бор

Пусть у нас есть набор строк $a_1, a_2, \ldots, a_n$ суммарной длины S, и к нам поступают запросы проверки, есть ли строка s среди строк этого множества. Для упрощения оценок в следующем абзаце, будем считать, что длина всех строк одинаковая и равна $|s|$. Также заметим, что для сохранения всех строк мы уже используем S памяти, поэтому количество потребляемой памяти не будет создавать нам проблем.

Понятно, что мы можем перебрать все строки a и сравнить их с s – на это потребуется $O(|s| \cdot n)$. Тут виден нюанс работы со строками – их сравнение работает за их длину. Но тем не менее, можно придумать более быстрое решение: за $O(|s| \cdot n \log n)$ отсортировать все строки и на каждый запрос делать бинарные поиск по отсортированному массиву за $O(|s| \cdot \log n)$. Но сейчас мы придумаем решение за $O(S)$ предподсчёта и $O(|s|)$ на запрос (то есть более оптимального решения быть не может, так как исходные строки и строки запроса нам всё равно придётся считать).

Для этого воспользуемся условием Фоно, а точнее построим структуру данных из него. У нас будет дерево, в котором каждая вершина будет хранить флаг – заканчивается ли в этой вершине строка (или сколько строк заканчиваются в этой вершине, если строки могут повторяться). А у каждого рёбра будем хранить ещё и букву, за которую оно отвечает. Рёбра можно хранить в `unordered_map` – тогда все оценки будут честными, или в массиве – тогда нужно что-бы A, размер алфавита (суммарное количество различных символов), можно было считать константой (обычно так и есть, например если все строки только из маленьких английских букв, то 26 вполне себе константа), так как в вершине будет храниться не $O(1)$ информации, а $O(A)$.

Заметим, что бор может даже больше, чем требуется в задаче поставленной выше. А именно, бор может добавлять строку в множество, искать строку в множестве, удалять строку и обходить строки в лексикографическом порядке.

- Начинаем построение с одной пустой вершины – корня.
- Чтобы добавить строку идём по её символам и смотрим, выходили ли мы из текущей вершины по такому символу. Если выходили, то переходим в соответствующую вершину и к следующему символу. А если ещё не выходили, то добавим в бор новую вершину и из текущей сделаем ребро в новую. В вершине, где строка закончится, нужно будет сделать +1 к флагу.
- Для поиска строки будем также идти по бору. В какой-то момент может не быть нужного ребра – значит нужной строки точно не было. А если же мы дошли до конца, то нужно будет проверить флаг на факт того, заканчивались ли там строки.
- Для удаления строки тоже спускаемся по бору, если какого-то ребра нет или в конце пути флаг пустой, то удалять ничего не нужно. А если дойти всё же смогли, то: уменьшаем флаг; если флаг обнулился и детей у вершины нет, то можно удалить текущую вершину и дальше удалять все родительские без детей и без флагов.
- Чтобы обходить строки в лексикографическом порядке поддерживать текущий путь от корня до вершины (не хранить в каждой вершине, а именно пересчитывать при обходе бора),

выводить текущий путь столько раз, сколько указано во флаге, а потом перебирать детей лексикографическом порядке и рекурсивно запускаться в них (пересчитав текущий путь). Стоит только оговориться, что в таком случае нужно стоит хранить рёбра выходящие из вершины в массиве размера A, или в `map`, чтобы каждый раз не сортировать рёбра.

Таким образом мы описали структуру данных, которая может добавить, удалить или найти строку во множестве. Если считать A константой, или использовать `unordered_map`, то тогда в каждой вершине будет выполняться константа операций и всего запрос для строки s выполнится за $O(|s|)$. Кроме того стоит добавить, что всего такой бор занимает $O(S)$ памяти, где S – сумма длин всех строк.

Кроме этого замечу, что кроме операции сортировки можно ещё научиться поддерживать поиск следующего и предыдущего лексикографически. Для этого спускаемся до слова, следующее или предыдущее для которого нужно найти, а чтобы найти следующее – поднимаемся по дереву, пока не найдётся ребёнок правее, переходим в него и дальше спускаемся всё время влево, пока не наткнёмся на вершину с флагом. Чтобы найти лексикографически предыдущее слово нужно также сделать подъём и спуск, только теперь в другую сторону.

Также добавлю, что традиционным способом хранения рёбер является массив, так как на практике операции с ним быстрые, а размеры алфавитов не такие большие (но формально бор будет занимать $O(AS)$ памяти). Но если у вас другая ситуация, то конечно же стоит использовать `unordered_map` или `map`.

Применение бора

Сам по себе бор мало где используется, но может являться удобной структурой для хранения данных в более сложных алгоритмах.

Суффиксный бор. Пусть нам дан один текст t и много запросов поиска подстроки s в тексте. Тогда, конечно же, можно использовать уже пройденные в прошлой главе алгоритмы. Но также можно использовать и бор, для этого добавим в бор все суффиксы t (то есть последний символ, два последних, и т.д. все символы) и все вершины бора помечаются терминальными (их флаг равен единице). Понятно, что если после этого выполнить поиск строки s в полученном боре, то найдётся ровно то, что мы хотели. Запросы поиск будут также выполняться за $O(|s|)$, но вот построение будет долгим и займёт $O(n^2)$ времени и памяти в худшем случае. Так что для разового запроса лучше использовать алгоритмы из прошлой главы.

Сверхбыстрый цифровой бор. Давайте вместо строк хранить числа в двоичной системе счисления (можно и в десятичной, но для двоичной можно всех детей хранить всего с помощью двух переменных). А если мы будем дополнительно поддерживать двусвязный список для всех добавленных чисел, то тогда операции по поиску следующего и предыдущего будут быстрыми – получим «быстрый цифровой бор». Чтобы его улучшить до «сверхбыстрого» нужно сделать «быстрый бор» на верхних слоях, а на нижних сделать дерево поиска, тогда можно показать, что все операции будут за $O(\log w)$ амортизировано, где w – длина двоичного представления чисел.

Ахо-Корасик. Возвращаемся снова к формулировке задачи для суффиксного бора, но немного переделаем: пусть нам дан набор строк s и для каждой строки из набора мы хотим находить все её вхождения в текст t, причём запросом считаем текст t. Тогда мы можем построить бор на этих строках s, дальше сделать над ним кое-какие преобразования, чтобы бор стал компактнее, а поиск был быстрее и после этого уже выполнить запрос по поиску в тексте t. Этот алгоритм работает за $O(|t| + |a|)$, где $|a|$ – длина ответа (то есть сколько раз строки s встретились в t).