

# Аллея

## Решение .

Найти наибольшую возрастающую подпоследовательность – классическая задача олимпиадной (и не только) информатики. Найти в сетях её решение несложно, но я уже рискнул дать эту задачу, хотя и в не совсем обычной упаковке, так что продолжу рисковать и изложу кратко её решение.

За что мы любим эту задачу? В первую очередь за то, что она является очень ярким и не совсем тривиальным примером применения динамического программирования.

Итак, имеется последовательность  $a_1, a_2, a_3, \dots, a_n$ . Обратите внимание! Я не сказал, что последовательность числовая, она и не должна быть числовой, главное, чтобы можно было сравнить любые два её элемента.

Будем последовательно считывать по одному элементы последовательности и каждый раз вычислять величину  $\text{maxlen}$  – длину самой длинной на настоящий момент возрастающей подпоследовательности. Для этого нам понадобится завести вспомогательный массив  $b$ :  $b_1, b_2, \dots, b_{\text{maxlen}}$ . В  $b_k$  ( $1 \leq k \leq \text{maxlen}$ ) будем хранить последний элемент наилучшей возрастающей подпоследовательности длины  $k$ . Считаем, что из двух возрастающих подпоследовательностей лучше та, у которой последний элемент меньше.

Сразу заметим, что массив  $b$  возрастает: для всех  $k$  ( $1 < k \leq \text{maxlen}$ )  $b_{k-1} < b_k$ . В самом деле,  $b_k$  – наименьшее возможное число, которым может оканчиваться возрастающая подпоследовательность длины  $k$ . Предпоследний элемент в этой (возрастающей!) подпоследовательности, назовём его  $c$ , меньше  $b_k$ . Следовательно, имеется возрастающая подпоследовательность длины  $(k-1)$ , оканчивающаяся на  $c$ , и поэтому  $b_{k-1} \leq c < b_k$ .

Рассмотрим процедуру обработки очередного элемента  $a_i$ .

Если  $b_{\text{maxlen}} < a_i$ , то существует возрастающая подпоследовательность длины  $\text{maxlen}+1$ , заканчивающаяся в  $a_i$ : увеличиваем  $\text{maxlen}$  на 1 и устанавливаем значение  $b_{\text{maxlen}}$  равным  $a_i$ .

Пусть  $b_{k-1} < a_i \leq b_k$  ( $1 < k \leq \text{maxlen}$ ). Это означает, что существует возрастающая подпоследовательность длины  $k$ , заканчивающаяся элементом  $a_i$ , и что никакая возрастающая подпоследовательность большей длины не может заканчиваться в  $a_i$ . При этом  $a_i \leq b_k$ , так что нам следует заменить значение  $b_k$  на  $a_i$ .

Если же  $a_i \leq b_1$ , то заменяем значение  $b_1$  на  $a_i$ .

Когда мы обработаем всю последовательность  $a_1, a_2, a_3, \dots, a_n$ ,  $\text{maxlen}$  будет содержать ответ на первый вопрос задачи – длину самой длинной возрастающей подпоследовательности.

Поиск  $k$ , удовлетворяющего соотношению  $b_{k-1} < a_i \leq b_k$  требует  $O(\log(\text{maxlen}))$  операций сравнения, если воспользоваться двоичным поиском.  $\text{maxlen} \leq n$ , всего требуется  $n$  поисков, следовательно, этот этап требует  $O(n \cdot \log n)$  операций сравнения.

Для того, чтобы найти какую-нибудь возрастающую подпоследовательность длины  $\text{maxlen}$  поступим вполне в духе динамического программирования: для каждого элемента  $a_i$  будем запоминать дополнительно номер того элемента, который мог бы оказаться предпоследним в самой длинной возрастающей подпоследовательности, оканчивающейся в  $a_i$ . Сделать это легко: когда мы заменяем значение  $b_k$  на  $a_i$ ,  $b_{k-1}$  содержит элемент последовательности, который

1. меньше  $a_i$
2. уже встречался в заданной последовательности, т.е. его номер меньше  $i$
3. является последним элементом некоторой возрастающей подпоследовательности длины  $(k-1)$

Одна неприятность – нам нужен номер этого элемента, а не его значение. Так за чем же дело стало – будем хранить в массиве  $b$  не сами элементы, а их номера. Это практически не изменит процедуры и заполнения массива  $b$  и поиска  $\text{maxlen}$  – надо только в  $b_k$  вписывать  $i$  вместо  $a_i$ , а в сравнениях использовать  $a[b_k]$  вместо  $b_k$ .

И тогда мы для каждого элемента будем знать, какой элемент надо поставить перед ним в самой длинной возрастающей подпоследовательности, оканчивающейся в нём. Восстановить самую длинную возрастающую подпоследовательность заданной последовательности теперь совсем легко – начинаем с любого элемента, в котором заканчивается возрастающая

подпоследовательность длины  $\max len$  и по шагам откатываемся назад вплоть до её первого элемента.

Всё это потребует дополнительно одного массива и  $O(\max len)$ , т.е.  $O(N)$  операций.

Казалось бы, всё. Нет, не совсем. У нас осталась ещё одна довольно ёмкая операция – сравнение деревьев, которая требует  $O(M)$  операций, где  $M$  – количество вершин в дереве. Конечно, в данной задаче  $M \leq 30$  – совсем небольшое число, но всё-таки...

Приглядимся повнимательнее к способу сравнения деревьев в задаче. Первым делом сравниваем количество сыновей у корней деревьев, и, если корни имеют разное количество сыновей, то процесс сравнения на этом заканчивается, а если одинаковое – то продолжается, причём сначала сравниваем поддеревья с корнем в левом сыне, а в случае их равенства – поддеревья в правом сыне. Что-то это напоминает... Лексикографическое сравнение это напоминает. Давайте соответственно и поступим – закодируем дерево такой строкой: сначала идёт количество потомков корня дерева, затем код левого поддерева, а потом код правого поддерева. Нехорошо получается – рассмотрим два таких дерева: первое дерево – это корень, к которому слева прицеплено некоторое дерево  $T$ , а правого сына нет; второе дерево – это корень, к которому справа прицеплено то же самое дерево  $T$ , а левого сына нет. Коды этих деревьев совпадают, а их надо как-то отличать. Заметим, что второе дерево меньше первого. Прекрасно, давайте корень с двумя сыновьями обозначать тройкой, корень с одним левым сыном – двойкой, с одним правым сыном – единицей, а корень без сыновей – нулём. Итак, кодируем дерево такой строкой: сначала идёт код корня дерева (0, 1, 2 или 3), затем код левого поддерева, а за ним 0-код правого поддерева. Нетрудно понять, что зная код дерева, можно однозначно восстановить само дерево (мы, правда, не сумеем восстановить номера вершин дерева, но это нас совершенно не интересует). Кроме того, вышеописанное преобразование дерева в строку сохраняет отношение «меньше». И таким образом, мы свели сравнение деревьев к лексикографическому сравнению двух строк, состоящих из символов 0, 1, 2, 3.

Это пока не ускоряет дело, но позволяет взглянуть на него с другой стороны. Ведь полученные коды деревьев – это просто числа, записанные в системе счисления с основанием 4. И если числа имеют одинаковую длину, то сравнить лексикографически их записи – это то же самое, что просто сравнить величины этих чисел. А если длины записей различаются? Тоже несложно – дополним слева более короткую строку необходимым количеством нулей – эта операция не изменит результат лексикографического сравнения. Понять это проще, чем объяснить, поэтому я воздержусь от объяснений. Вообще, дописывание к строке (к любой, не обязательно более короткой) минимального возможного символа в любых количествах не меняет результат лексикографического сравнения строк.

Так давайте, чтоб не мучиться, дополним код каждой из заданных строк до 30 символов – больше 30 вершин в заданных деревьях не может быть.

И последнее, четверичная система – это почти то же самое, что двоичная, надо только каждую четверичную цифру заменить на две двоичных: 3 – на 11, 2 – на 10, 1 – на 01, и 0 – на 00. 30 четверичных цифр дают 60 двоичных, которые прекрасно размещаются в одной 64-битовой целочисленной переменной. Итак, мы свели сравнение деревьев к одному сравнению двух чисел. Конечно, потребуется время на перевод деревьев в числа, но это себя окупает: перевод требует  $O(N \cdot M)$  операций, а поиск наибольшей возрастающей подпоследовательности – ещё  $O(N \cdot \log N)$ , итого получаем сложность  $O(N \cdot M + N \cdot \log N)$ . Если же не кодировать деревья числами, то алгоритм требует  $O(N \cdot \log N)$  сравнений деревьев, т.е. имеет сложность  $O(N \cdot \log N \cdot M)$ .