Solution of problem «Secret — MAX»

Topic: dynamic programming (!), string searching (?).

AFAIK, there is no special property in the given digits' names; algorithm can work even reading the names as input data.

Overall this text, let's compare long numbers in such way: if lengths are different, longer is greater; for equal length, perform lexicographical compare. This may contradict standard integer comparison in case of leading zeros, f. e. 007 > 8.

For good performance in main stage of algorithm, we need table endsAt[][][], where the leftmost index $0 \le lang \le 3$ is one of four languages, the middle $0 \le dig \le 9$ is decimal digit, and the rightmost $0 \le pos < N \le 100000$ is position in input string. The element endsAt[*lang*][*dig*][*pos*] means "if we'll search digit *dig*'s name in language *lang*, starting from position *pos*, what will be the leftmost possible position just after end of the name?". If this name cannot be found till end of data, endsAt[*lang*][*dig*][*pos*] = MYNULL (special value, f. e. 0x7FFFFFFF).

Let's apply dynamic programming (DP) 5 times: for language #1, language #2, language #3, language #4 and for all four languages. For each of the 5 cases, we build $T_0[]$ and $T_1[]$ tables, where $T_0(j)$ means "what maximal (in described above sense) number (including numbers with leading zeroes) can be found in substring from position *j* till the end" and $T_1(j)$ means "what maximal number with non-zero first digit can be found in substring from position *j* till the end".

$$T_0(i) = \max_{\substack{0 \le dig \le 9 \\ lang \\ endsAt[lang][dig][i] \ne MYNULL}} (concat(dig, T_0(endsAt[lang][dig][i])))$$

$$T_1(i) = \max_{\substack{1 \le dig \le 9 \\ lang \\ endsAt[lang][dig][i] \ne MYNULL}} (concat(dig, T_0(endsAt[lang][dig][i])))$$

where "concat" means string concatenation; "*lang*" 4 times mean "only the selected language" and once mean "over all 4 languages". $T_1(0)$ is answer of the whole DP (the 1st answer of the whole problem is maximum of four $T_1(0)$ found for different languages, and the 2nd is $T_1(0)$ found for all four languages together).

It's important to ***not*** store $T_1(i)$ and $T_0(i)$ as ready sequences in *i*-th cell of DP array. Instead, we store for each $T_1(i)$ and each $T_0(i)$ three values: its length, its first (eldest) digit and the position in $T_0[]$ where the optimal sequence continues — it is endsAt[*lang*][*dig*][*i*] for that *dig* and *lang* where maximum was reached.

One comparison of $T_0(i)$ or $T_1(i)$ with concat(*dig*, T_0(endsAt[*lang*][*dig*][*i*])) ***can*** be performed in $\theta(1)$ time. We can quickly get either "lengths are different" or "whole sequences are the same" (not because "equal", but because "consisting of the elements located at the same places"). To ensure the last property, we need, after finding $T_0(i+1)$ and $T_1(i+1)$, initialize $T_0(i) := T_0(i+1)$ and $T_1(i) := T_1(i+1)$, and, comparing them with all possible values concat(*dig*, T_0(endsAt[*lang*][*dig*][*i*])), change the triple "length, first digit, continue pos" only when got strictly greater result (do ***not*** change the triple when got equal).

Solution of problem «Choosing a camera»

Topic: data structures (augmenting standard map / TreeMap, priority queue).

The main thing we need to solve the problem is to implement data structure allowing efficiently maintaining set of non-outdated cameras.

Suppose temporarily that there are no different offers with the same pixels and the same zoom. Consider mapping (using "map" of C++ STL or "TreeMap" of Java collections) from integers representing pixels to structures representing zoom, cost and offer's number. The items are sorted by pixels (that's provided by "map" / "TreeMap"). When the map contains only non-outdated cameras, it becomes also sorted (in reverse order) by zoom. Processing each "P" action, we firstly look at "lower_bound" / "ceilingKey". If it doesn't exist, the new camera is non-outdated because have the greatest pixels number. If lower_bound exists and its zoom is greater-equal than new camera's zoom, then the new camera is outdated (and will remain outdated forever), so we can forget about it. Else, the new camera is non-outdated. When we found that new camera is not outdated, we should not only insert it to the set, but also to check whether it made outdated some other cameras. If it did, the cameras just became outdated are some consecutive previous elements of the map, so we can move lower and erase elements until have erased begin or until see an element with strictly greater zoom.

Coming back to case when different offers may describe cameras with the same pixels number and the same zoom, we see that we cannot easily store all of them in the map. But we don't need this. It's more useful to compare costs, and replace old offer by new if and only if new cost is strictly less.

Strictly speaking, it means that the maintained set contains not all non-outdated camera but only non-outdated cameras which have hypothetical chance to be chosen (after appearing new offers, which are too expensive and make outdated other cameras non-outdated now). But, for shortness, let's continue using term "set of non-outdated".

The data structure described above maintains set of non-outdated, but cannot find quickly the minimal cost. So let's store not only the map, but also Boolean array isOptimal, where isOptimal[i] means "camera, introduced in action #i, is not outdated now". We change the isOptimal[i] when deciding not to include the camera into the set of non-outdated, or when deciding to include and when deciding to exclude. At last, let's store priority queue (binary heap) byCost, sorted by cost, and, for equal costs, by offer indice (root contains minimal cost; if there are different offers with equal minimal cost, the oldest among them).

Processing action "C", we don't deal directly with the set of non-outdated. If byCost's root contains non-outdated camera (we check it in isOptimal array), its index is the current choice. Otherwise, we extract elements from the priority queue until reach a non-outdated.

(Of course, clever programmer doesn't push to byCost any camera, which is already outdated when just had been proposed by shop. But it's typical that cameras may become outdated while storing in the priority queue…)

By the way, answer "–1" is possible if and only if "C" action is the first, without any "P" actions before.

Note that processing some concrete actions may be rather long (erasing from the map many elements just became outdated, or extracting from priority queue many cheap but now outdated elements). But each offer is included into the set of non-outdated at most once and is excluded at most once. Similarly, each offer is pushed to priority queue at most once and is extracted from priority queue at most once. So overall run time is in range $\Omega(n)$, $\mathbf{O}(n \log n)$.

Розв'язання задачі «Секрет — максимум»

Теми: динамічне програмування (!),пошук рядків (?).

Наскільки відомо автору задачі, у вказаних назвах цифр нема особливих властивостей; алгоритм може працювати навіть читаючи ці назви в якості вхідних даних.

Будемо порівнювати довгі числа таким чином: якщо довжини різні, довше вважається більшим; для чисел однакової довжини, застосовуємо лексикографічне порівняння. Це може суперечити стандартному порівнянню цілих чисел в разі початкових нулів, напр. 007> 8.

Для забезпечення ефективності основного етапу алгоритму, потрібен масив endsAt[][][], де лівий індекс $0 \le lang \le 3$ відповідає одній з чотирьох мов, середній $0 \le dig \le 9$ — десятковій цифрі, а правий $0 \le pos < N \le 100000$ — позиції у вхідному рядку. Елемент endsAt[*lang*][*dig*][*pos*] означає «якщо шукати ім'я цифри *dig* мовою *lang*, починаючи з позиції *pos*, якою буде найлівіша можлива позиція відразу після закінчення цього імені?». Якщо це ім'я не можна знайти до кінця даних, endsAt[*lang*][*dig*][*pos*] = MYNULL (особливе значення, напр. 0x7FFFFFFF).

Застосуємо динамічне програмування (ДП) 5 разів: для № 1, мови № 2, мови № 3, мови № 4 і для всіх чотирьох мов. Для кожного з 5 випадків, будуємо таблиці T_0[] та T_1[], де $T_0(j)$ означає «яке максимальне (в описаному вище сенсі) число (включаючи числа з ведучими нулями) можна знайти в підрядку від позиції j до кінця», а $T_1(j)$ означає «яке максимальне число з ненульовою першою цифрою можна знайти в підрядку від позиції j до кінця».

$$T_0(i) = \max_{\substack{0 \le dig \le 9 \\ lang \\ endsAt[lang][dig][i] \ne MYNULL}} (concat(dig, T_0(endsAt[lang][dig][i])))$$

$$T_1(i) = \max_{\substack{1 \le dig \le 9 \\ lang \\ endsAt[lang][dig][i] \ne MYNULL}} (concat(dig, T_0(endsAt[lang][dig][i])))$$

де "concat" означає конкатенацію рядків; "*lang*" 4 рази означає «лише вибрана мова» і один раз означає «серед усіх 4 мов». Головною відповіддю ДП є $T_1(0)$ (перша відповідь усієї задачі — максимум з чотирьох $T_1(0)$, знайдених для різних мов; друга — $T_1(0)$, знайдене для всіх чотирьох мов разом).

Важливо ***не*** зберігати $T_1(i)$ та $T_0(i)$ у вигляді готових послідовностей в i-й комірці масиву ДП. Замість цього, зберігаємо для кожного $T_1(i)$ та кожного $T_0(i)$ три значення: його довжину, його першу (старшу) цифру, та позицію в T_0[], звідки оптимальна послідовність продовжується — це endsAt[*lang*][*dig*][*i*] для тих *dig* та *lang*, де був досягнутий максимум.

Одне порівняння $T_0(i)$ або $T_1(i)$ з concat(*dig*, T_0(endsAt[*lang*][*dig*][*i*])) ***можна*** виконати за час $\theta(1)$. Можна швидко отримати або «довжини різні», або «послідовності співпадають» (не просто «еквівалентні», а «складаються з елементів, розташованих у тих самих місцях»). Для забезпечення даної властивості, потрібно, знайшовши $T_0(i+1)$ та $T_1(i+1)$, ініціалізувати $T_0(i) := T_0(i+1)$ та $T_1(i) := T_1(i+1)$, і, порівнюючи їх з усіма можливими значеннями concat(*dig*, T_0(endsAt[*lang*][*dig*][*i*])), міняти трійку «довжина, перша цифра, позиція продовження» тільки коли знайдено строго більший результат (***не*** змінюючи, коли знайдено рівний).

Розв'язання задачі «Вибір фотоапарату»

Тема: структури даних (розширення стандартного map / TreeMap, піраміда).

Головне, що потрібно для розв'язання задачі — реалізація структури даних, що дозволяє ефективно підтримувати множину не застарілих апаратів.

Тимчасово припустимо, ніби не буває різних пропозиції, де однакові і кількість пікселів, і кратність зума. Розглянемо відображення ("map" C++ STL або "TreeMap" колекцій Java) з цілих чисел–пікселів у структури, що містять зум, вартість і номер пропозиції. Елементи відсортовані за пікселями (це гарантується структурою "map" / "TreeMap"). Коли відображення містить тільки не застарілі фотоапарати, воно виявляється відсортованим (у зворотному порядку) також і за зумом. Обробляючи кожну дію "P", спочатку дивимося на "lower_bound" / "ceilingKey". Якщо його не існує, новий фотоапарат не застарілий, тому що має найбільшу кількість пікселів. Якщо lower_bound існує і його зум більший-рівний зуму нового апарату, то новий апарат застарілий (і це назавжди), так що ми можемо забути про нього. Інакше, новий апарат не застарілий. Щоразу, коли з'ясовуємо, що новий апарат не застарілий, треба не тільки вставити його у відображення, але й перевірити, чи не зробив він застарілими деякі інші апарати. Якщо це так, то апарати, які щойно стали застарілими, зберігаються у послідовних попередніх елементах відображення, так що ми можемо рухатися в бік його початку й вилучати елементи, доки не вилучимо початок або доки не знайдемо елемент зі строго більшим зумом.

Вертаючись до випадку, коли різні пропозиції можуть описувати апарати з однаковою кількістю пікселів і однаковою кратністю зума, бачимо, що не можна легко зберігати у відображенні їх усі. Але це й не потрібно. Краще порівнювати вартості і замінювати старі пропозиції новими, якщо і тільки якщо нова вартість строго менша.

Строго кажучи, це означає, що підтримуваний набір містить не всі не застарілі фотоапарати, а лише не застарілі апарати, які мають гіпотетичний шанс бути обраними (після появи нових, надто дорогих, пропозицій, які зроблять застарілими інші камери, не застарілі зараз). Але, задля стислості, продовжимо користуватися терміном «множина не застарілих».

Розглянута структура даних підтримує множину не застарілих, але не може швидко шукати апарат мінімальної вартості. Тому будемо зберігати не тільки відображення, але і логічний масив isOptimal, де isOptimal[i] означає «апарат, вперше згаданий у дії № i, на даний момент не застарів». Будемо змінювати isOptimal[i] при прийнятті рішення не включати апарат у набір не застарілих, або при прийнятті рішення включити та при прийнятті рішення виключити. Нарешті, будемо зберігати піраміду (вона ж черга з пріоритетами, вона ж бінарна купа) byCost, впорядковану за вартістю, а при рівних вартостях за номерами пропозицій (корінь піраміди містить мінімальну вартість, а якщо існують різні пропозиції з однаковими мінімальними вартостями, то найдавнішу серед них).

При обробці запитів "C", множина не застарілих не використовується безпосередньо. Якщо корінь byCost містить не застарілий апарат (перевіряємо це за масивом isOptimal), то індекс, що зберігається у корені, і є поточним вибором. Інакше, вилучаємо елементи з піраміди, доки не дійдемо до не застарілого.

(Звичайно, розумний програміст не заноситиме у byCost апарати, вже застарілі на момент оголошення пропозиції. Але апарати можуть застарівати за час зберігання у піраміді...)

До речі, відповідь "–1" можлива тоді й тільки тоді, коли першою дією є "C", без попередніх дій "P".

Відзначимо, що обробка деяких окремих дій може бути досить довгою (вилучення з відображення багатьох щойно застарілих елементів, або вилучення з піраміди багатьох дешевих, але вже застарілих елементів). Але кожна пропозиція включається до множини не застарілих не більше одного разу і виключається не більше одного разу. Аналогічно, кожна пропозиція не більше одного разу вставляється у піраміду і не більше одного разу вилучається. Отже, загальний час виконання знаходиться в діапазоні $\Omega(n)$, $O(n \log n)$.