**Анализ**

*Тагове: най-дълга нарастваща подредица, дърво на Фенуик, персистентни структури от данни, персистентно сегментно дърво*

Условието иска от нас да намерим минималния брой строго намаляващи подредици, на които може да се разбие дадена редица (масив). След като се направи ключово наблюдение се оказва, че тази бройка всъщност е големината на най-дългата нарастваща подредица. Така заявките стават модификация на класическия алгоритъм, така че да поддържа промяна на един елемент (независими промени). Понеже задачата е псевдо-интерактивна, то трябва да имаме онлайн алгоритъм, който да поддържа заявките. Това налага някакъв вид персистенция на стандартните алгоритми за намиране на най-дълга нарастваща подредица.

Първата подзадача е за 11 точки. Тя е предвидена за най-простия алчен алгоритъм за решаване на задачата. Тук има заявки, но те са, заради малкото ограничение на ***N***. Нека започнем отляво-надясно и намираме намаляващите подредици малко по-малко. Вече сме построили някакви подредици от досегашните числа и се чудим къде трябва да добавим текущото. Ясно е, че можем само на тази подредици, които завършват с по-голямо число. Ако няма такава, то нямаме друга опция освен да направим нова подредеца. Нека си мислим, че имаме такива краища. Освен това можем да направим следното алчно наблюдение – най-добре е да добавим текущия елемент при най-малкия възможен край (който е по-голямо число). Това е така, защото колкото по-голям е даден завършек толкова повече възможни числа можем да добавим. Затова най-оптимално е да „закрием“ най-малкия възможен край. Подзадачата е предвидена за най-лесния начин за реализиране на този подход. Очакваната сложност тук е .

Втората подзадача е за 17 точки. Нека при предния алгоритъм, когато добавяме нова подредица, я слагаме най-накрая (както е естествено) като последна. Едно лесно наблюдение, което можем да направим е, че краищата на подредиците, които ще строим, винаги ще са подредени в ненамаляващ ред. Това означава, че за да намерим къде трябва да сложим текущия елемент, можем да направим двоично търсене, а не линейно. Така сложността става .

Третата подзадача е за 24 точки. Тя е съществена стъпка към пълното решение. Ясно е, че трудно можем да разберем как се променя минималната бройка при смяна на един елемент с досегашния алгоритъм. Затова трябва да направим важно наблюдение каква точно е търсената бройка. В предната подзадача казахме, че краищата образуват ненамаляваща редица. Това може да ни накара да мислим в посока за най-дълга ненамаляваща подредица. Всъщност ако разгледаме една такава редица, то тя трябва да е в различни подредици при разбиването (защото числата трябва строго да намаляват в една и съща подредица). Това означава, че търсената минимална бройка ще е поне колкото дължината на най-дългата ненамаляваща подредица. От друга страна казахме, че краищата образуват ненамаляваща редица. Почвайки от края на последната редица, можем последователно да намерим ненамаляваща подредица на началната, използвайки следния алгоритъм. Нека сме на редица *k* и елемент *t*. Тогава можем да се върнем в момента, в който е поставен елемент *t* и да отидем на края на предната редица в този момент, който със сигурност ще е по-малък или равен на *t* и да го включим в началото на текущата ненамаляваща подредица. Така ще стигнем до първата редица и с това ще намерим ненамаляваща подредица с дължина колкото е търсения отговор. Но това трябва да е по-малка или равно на дължината на най-дългата ненамаляваща подредица. Така тази дължина се явява и долна, и горна граница за отговора, който търсим. Това означава, че отговорът, който търсим е просто дължината на най-дългата ненамаляваща подредица. Сега да видим какво може да се случи с нейната дължина, когато се промени едно число. Очевидно може да нарасне най-много с 1. Да помислим как може да установяваме това. Щом се увеличи с 1, то този елемент, който се променя, трябва да участва в новополучената най-дълга ненамаляваща подредица. Така че трябва да имаме начин да следим при промяна на елемент коя е най-дългата ненамаляваща подредица, включваща го. Това става най-лесно като открием коя е най-дългата ненамаляваща подредица, завършваща в този елемент и коя е най-дългата ненамаляваща, започваща от този елемент (като сумираме тези две дължини и извадим 1, получаваме търсената дължина). Второто нещо можем да си мислим, че е аналогично на първото, ако гледаме масива отдясно-наляво и търсим най-дългата намаляваща подредица. Понеже ограниченията са малки, можем да си позволим следния лесен начин да намираме тези неща при промяна на елемент. Правим класическия алгоритъм с двоично търсене за намиране на най-дълга ненамаляваща подредица в двете посоки, като на всяка стъпка си запазваме какви са числата в масива с най-малките завършвания за всяка дължина (по който се прави двоичното търсене). По този начин потенциално може да имаме квадратична памет, но това не е проблем заради малкото ***N***. Сега като имаме промяна просто трябва да погледнем подходящия масив за съответната позиция и да направим двоично търсене в него, за да определим нужната дължина. Така че този алгоритъм ни позволява при промяна на елемент бързо да намираме дължината на най-дългата ненамаляваща подредица, която го включва. Дотук можем да преценяваме дали отговорът се увеличава с 1 – в зависимост от това дали тази дължина стане по-голяма от досегашната максимална. Ако е равна на досегашната максимална също е ясно, че отговорът не се променя. Остава да видим какво се случва, ако е по-малка. Ако е много малка, то най-дългата ненамаляваща в целия масив може да намалее само най-много с 1 (ако случайно я разваляме, то можем да не гледаме текущото число и така имаме оставаща ненамаляваща подредица с дължина с 1 по-малко). Така че интересният случай е само ако най-дългата ненамаляваща подредица, включваща променения елемент, е с дължина с 1 по-малка от досегашната (преди промяната). За да преценим дали за целия масив наистина намалява дължината, то трябва да знаем дали този елемент е бил в някакъв смисъл централен, т.е. дали всяка най-дълга ненамаляваща подредица в масива го е включвала. Предвид ниските ограничения това може да преизчислим лесно – просто намираме дължината на най-дългата ненамаляваща без всеки елемент и гледаме дали намалява с 1 или не. Така вече сме готови с всички случаи при заявката. Крайната сложност по време за тази подзадача е , а по памет .

Четвъртата подзадача е за 22 точки. Тук ще оптимизираме предната реализация. Нека първо оптимизираме последното нещо, за което говорихме – изчисляването на нужните елементи за построяването на най-дълга ненамаляваща подредица. Нека си фиксираме някакъв елемент *a[i]*, който участва в най-дълга ненамаляваща подредица. Ясно е, че елементите с индекс > *i*, които са с по-малка стойност, няма как да участват в най-дълга нарастваща подредица с такъв елемент. За тези надясно, които са с по-голяма или равна стойност е възможно. Същото ще важи за елементите наляво, които имат стойност по-голяма от *a[i]* – те няма как да участват в една най-дълга нарастваща подредица с елемента *a[i]*. Ако има най-дълга нарастваща подредица, която не ползва *a[i]*, то тя ще има число вдясно, което ще е по-малко от *a[i]* или число вляво, което ще е по-голямо от *a[i]*. При допускане на противното можем да получим лесно противоречие, защото просто ще можем да прибавим елемента *a[i]* към уж най-дългата ненамялаваща и да получим по-дълга. Това ни дава възможност за следния алгоритъм. В първи пас отдясно-наляво в дърво на Фенуик ще записваме стойностите, за които имаме най-дълга ненамаляваща подредица. Когато стигнем до *a[i]*, трябва да направим една префиксна заявка, за да видим имало ли е по-малка стойност досега, която е участвала в най-дълга нарастваща подредица и да си отбележим ако сме намерили, за да знаем, че този елемент не е нужен за формирането. Аналогично ще имаме втори пас отляво-надясно, където ще гледаме за по-големите стойности от *a[i]* вляво. Сега остана да изясним втората част, при която преди просто пазехме всички „състояния“ на масива с минималните завършвания за всяка дължина. Голяма част от информацията се повтаря (за всяка позиция се променя само едно число дефакто). Тук вече трябва да използваме някаква идея за персистенция, която би трябвало лесно да може да се измисли без човек да се е запознавал по темата. Ще направим персистентен масив с най-малките завършвания за всяка дължина, което ще рече, че ще можем сравнително бързо да се „върнем“ при негово старо състояние. За всяка дължина ще имаме вектор, в който ще записваме освен стойността на минималния завършек и индекс, когато за първи път сме го намерили. Така при изпълняване на алгоритъма за най-дълга ненамаляваща подредица, вместо да правим копиране, просто ще сложим в този вектор нов елемент, накрая на вектора, със съответната стойност и индекс. Разбира се, това ще направим и в двете посоки. Сега като имаме заявка, например за позиция *i*, отново искаме да направим двоично търсене по дължината. Като сме фиксирали с двоичното търсене някаква дължина *l*, гледаме във вектора за *l*, за да открием каква е била стойността на тази клетка във време *i*. Това можем да открием с друго двоично търсене по вектора с промени, за да открием най-близкия момент по-малък или равен на *i*, когато ще се е случила последната промяна до този момент. Сложност на това решение по време е , разбира се логаритъмът за търсене във векторите е доста малък в общия случай, паметта на решението е .

Последната подзадача е за 27 точки. Тук ще оптимизираме заявката, но на цената на повече памет, както става в много случаи. Досега се бяхме концентрирали само върху алгоритъма за намиране на най-дългата ненамаляваща подредица с двоичното търсене. Другият известен алгоритъм е този със сегментно дърво, където записваме за всяка стойност дължината на най-дългата ненамялаваща подредица, завършваща в нея. Разбира се, отново ще търсим някаква персистенция на сегметното. Но при него има доста по-добра от тази при масива. На всяка позиция в масива извършваме само по една заявка в сегментното дърво, т.е. променяме най-много върхове. Това, което се прави по принцип, е просто да копираме само променените върхове и да ги навържем със старите непроменени върхове. Така лесно можем да се върнем в старо състояние и освен това допълнителната памет е малко. Просто трябва да пазим за всяка позиция, къде се намира корена на сегментното, от там нататък ние сме навързали по подходящ начин върховете, така че като се спускаме да не мислим за състояния. Крайната сложност по време на решението е , но по памет е .

Вече можем да кажем защо и максималното число са толкова малки, като в решенията няма някакви големи константи. Проблемът да са по-големи е, че двете решения за последните две подзадачи ще станат неразличими. Както отбелязахме по-рано, е доста лек, освен това се ползва доста ефективно кеша за вътрешното двоично търсене. Докато при сегментното дърво, при заявката за търсене, има скрита константа до 4, но по-лошото е, че паметта за пазенето на историята е голяма и така кеша не се ползва ефективно и се държи доста бавно спрямо другия начин. Затова при доста по-ниски ограничения, се получава ефективно различаване.

*Автор: Илиян Йорданов*