ShoeShopping

(решение)

Както повечето други задачи за Ели в С група, и тази изискваше няколко много основни алгоритъма (този път и структура данни).

Ще обработваме въпросите (query-тата) едно след друго, всяко независимо от останалите. Ако имаме въпрос от типа (money, maxPrice), то за да отговорим за него трябва да взимаме обувките в намаляващ ред на цената, започвайки от най-скъпите обувки с цена по-малка или равна на maxPrice. Един възможен подход е на всяка стъпка да обхождаме всички обувки и да търсим тези от тях, които са най-скъпи от невзетите, с цена <= maxPrice. Разбира се, този алгоритъм имплементира точно това, което се иска от нас в задачата, за съжаление е твърде бавен (неговата сложност по време е O(Q * N^2)). Поради дадените ограничения за различните тестове, състезателите трябваше да могат да преценят, че този алгоритъм би хванал около 30-40 точки. Така че той беше хубав избор за състезателите, които искаха някакви (макар и малко) сигурни точки.

Все пак, тъй като задачата беше предвидена за последно контролно за национален отбор (младша възраст), се очакваше много от участниците да се стремят към пълен (или почти пълен) брой точки. Нека разгледаме как можем да постигнем такова решение.

Първият от стандартните алгоритми, които ще навържем, беше сортиране – за да се постигне ефективен алгоритъм цените на обувките трябваше да бъдат сортирани в намаляващ ред на цената им. Това вече е сравнително тривиално за тази група (а едно време не беше!), главно поради наличието на вградени алгоритми за това, които състезателите отдавна се научиха да ползват.

С какво ни помага това, че обувките са сортирани? Ами след като веднъж сме намерили най-скъпите обувки, които Ели ще си вземе, то просто взимаме всички след тях докато имаме пари. Това намаля броя нужни операции с фактор от N, като сложността по време става O(Q * N). Наистина, за всяко куери ние обхождаме масива най-много веднъж (даже, в общия случай, по-малко от цяло обхождане). Обхождаме последователни негови елементи докато намерим най-скъпите обувки, които Ели ще си вземе, и после продължаваме нататък докато имаме пари. Тази оптимизация би спечелила още известен брой точки – с нея състезателите биха постигнали около 60-70 точки.

Следващата оптимизация е напоследък модерна в задачи за Ели в С група. С известна съобразителност състезателите можеха да се досетят, че броя обувки, които Ели ще си вземе рядко е много голям. Наистина, изискваше се наличието на много специфични тестове за да се наложи тя да си вземе повече от няколко чифта обувки. Само в няколко теста на нея й се налага да вземе над 100 или 1000.

Тоест след като сме намерили първите обувки ще ни се наложи да направим само няколко итерации за да намерим отговора за съответното куери. Тук на помощ идва техниката двоично търсене (binary search). След като масивът ни е сортиран, можем да ползваме двоично търсене да намерим най-скъпите обувки, които все пак са не по-скъпи от maxPrice. С тази оптимизация състезателите хващат около 80 точки.

Последната стъпка за 100-те точки беше най-нестандартната (за тази група) и изискваше известно мислене. Няма ли някакъв начин „набързо" да преброим колко обувки ще си вземе Ели, след като сме намерили първия чифт? В интерес на истината има! Ако има начин, по който да намерим сумата на всички елементи от i-ти до j-ти индекс в масив (i <= j), то бихме могли да ползваме второ двоично търсене (веднага след края на първото). Структури данни, които ни дават сумата на елементите в някакъв интервал има (примерно индексни дървета), но те са твърде сложни за тази група. За да се справим със ситуацията трябва да забележим, че елементите на масива не се променят между и по време на куеритата. Тоест можем да преизчислим някакви резултати и да ги ползваме за всяко от тях. В случая ще ползваме структурата данни „префиксен масив" – тоест втори масив, който ни пази сумите от първия до i-тия елемент. Нека например имаме масива a[] = {5, 1, 4, 3, 3, 13, 2}. Създаваме масив sums[] = {5, 6, 10, 13, 16, 29, 31}, който на i-та позиция съдържа сумата от първия до i-тия елемент на масива a[], включително. Този масив се създава сравнително лесно. sums[0] = a[0]. For i = 1; i < n; i++: sums[i] = sums[i – 1] + a[i]. Този прост цикъл прави точно това, което искахме ☺ Сега имаме лесен (и по-важното бърз) начин да намерим сумата на числата от началото до дадена позиция. Нас обаче ни интересува да можем да намерим сумата между два индекса, като първият не задължително е 0. Това става лесно. Ако търсим сумата в интервала [i, j] (i <= j), то ако i == 0 връщаме sums[j], иначе връщаме sums[j] – sums[i – 1]. Ако не разбирате защо това е така, си разпишете горния пример и разгледайте няколко различни двойки (i, j) – по някое време ще го „видите" и ще го запомните завинаги (поне така беше при мен).

Така, измислихме структура данни, която ни дава сумата на числата между два индекса. С първото двоично търсене сме намерили индекса, от който почваме (тоест i). С второто двоично търсене искаме да намерим втори индекс (сиреч j), такъв, че да е максимален и в същото време сумата на числата между i и j да е по-малка или равна на money. Това двоично търсене е малко по-сложно от първото (тоест функцията, която определя накъде местим границите е малко по-сложна от тази в първото), но все пак вярвам, че състезателите ще се справят с нея ако са се справили с първата. Като цяло най-сложното нещо в задачата беше да се измисли префиксния масив и да се навържат алгоритмите.

В крайна сметка сложността на последното решение е O(N + Q * (logN + logN)). Забележете, че тъй като двоичните търсения не са вложени (тоест едно в друго)

тяхната сложност се събира, а не умножава. Така решението върви много бързо за дадените ограничения. В сложността сме включили и O(N), която е нужна за четенето на входя и намирането на сумите в префиксния масив.

Автор: Александър Георгиев