**Road Signs**

(Анализ)

Задачата Road Signs беше инспирирана от задачата Mysterious Road Signs, дадена наскоро в един от квалификационните кръгове на Google Code Jam 2018 (<https://codejam.withgoogle.com/2018/challenges/0000000000007764/dashboard/000000000003675b>).

Макар и с малка разлика (просто знаците не задължително трябва да са последователни), решението на варианта, който дадох, няма нищо общо с решенията на оригиналната задача. Fun fact: моят вариант е просто как бях разбрал задачата по време на състезанието (не бях видял, че трябва да са последователни).

Тази задача има много възможни подходи, и, за съжаление, много начини, по които може да се cheat-не. Някои от чийтовете, които ми хрумнаха, са относително лесни за измисляне, а в същото време хващат много точки (около 80 точки с това, което на мен ми хрумна, но с малко повече мислене и тестване изобщо няма да ме учуди да могат да се хванат и 100, особено ако има feedback).

Нека разгледаме няколко от решенията не-чийт решенията.

**O(2N \* N)**

Очевидно, можем да направим брутфорс решение по всички под-множества от знаци и да проверим дали има две точки, такива, че всеки от знаците да сочи поне една от тях. Това става със сложност O(2N \* N) и всъщност е по-сложно да се имплементира от следващите две, значително по-бързи решения, затова няма да задълбаваме много в тази идея.

**O(N3)**

Друга относително проста идея беше отново брутфорс, но вместо по подмножествата на знаците, по самите двойки точки. Тъй като имаме най-много 2N кандидата (всеки знак сочи по две точки), можем за всяка двойка да намерим нейната важност (броейки най-добрите). Имаме 2N варианта за първа точка, 2N/2 варианта за втора точка (няма смисъл втората точка да е с по-малки координати от първата, затова делим на две), и още O(N) да намерим колко знака сочат някоя от двете, постигаме O(N3) решение.

**O(N2)**

Вкарвайки малко мислене, можем да оптимизираме горното решение до O(N2). Отново ще фиксираме една от точките, но другата ще намерим чрез Greedy. След като сме фиксирали едната точка, итерираме всички знаци и:

* Ако знакът сочи фиксираната точка, го броим към важността
* Ако пък не я сочи, в хешмап запазваме двете точки, които сочи

Накрая в хешмапа ще имаме броя все още неизползвани знаци, които сочат всяка точка. Вземайки максимума и събирайки с вече преброените знаци, които сочат фиксираната точка, намираме отговора. Тъй като операциите с хешмап са O(1) имаме O(N) за да фиксираме първата точка и още O(N) за да приложим Greedy-то, давайки сумарно O(N2).

**O(N \* logN), set**

Можем да свалим сложността до O(N\*logN), оптимизирайки горната идея чрез ad-hoc структура данни. Бавното горе беше линейното грийди да намерим най-"популярната" точка, сочена от знаци, несочещи фиксираната точка. Вместо това ще си пазим няколко неща:

* Знаците, които сочат всяка точка (вектор)
* За всяка точка, колко знака я сочат в момента (масив)
* За всяко ниво на "популярност", колко точки имат тази популярност (масив)
* Подредено множество от точките, сортирани спрямо това колко знака ги сочат (map)

След като фиксираме дадена точка, можем да обходим вектора със знаците, които я сочат, и да ъпдейтнем масивите и map-а. Тъй като всеки знак сочи точно две точки, и всяка точка обхождаме точно по веднъж, то всяка точка ще вкараме и изкараме от масивите и map-а точно по два пъти. Това все пак е константен (О(1)) брой операции на точка, и тъй като операциите са със сложност O(1) за масивите и O(log) за map-а, получаваме сумарно O(N \* logN) за цялата задача.

Може би не е много ясно за какво ползваме мап-а. В него пазим всяка точка, сочена от поне един знак, и колко знака я сочат (като сортираме мап-а по това колко знака я сочат). Така чрез него можем бързо да намерим най-популярните други точки, освен фиксираната. Масивите пък ни помагат да можем по-бързо да смятаме отговора (тъй като, потенциално, много точки могат да бъдат с максимума – например, ако имаме N непресичащи се точки, всяка сочена точка се сочи по веднъж, но това е и максимума.

**О(N \* logN), sort**

Оказва се, че можем да минем и без set, а само със сортиране (което има значително по-ниска константа). Ако просто вземем всяка потенциално важна точка, заедно с броя знаци, които я сочат, и сортираме по броя знаци в намаляващ ред, получаваме масив с точките, подредени по важност.

Как премахваме set-a? Отново ще си пазим масивите с бройката знаци, сочещи всяка точка (работата с тях е константна). Вместо сета, вече имаме сортиран масив с точките по важност. Когато фиксираме някоя от двете точки в двойка потенциално "най-важни" точки, този масив става невалиден (тоест вече подредбата може да се промени), но не ни пречи да почнем да го обхождаме, докато намерим непроменен елемент (за всички променени от масивите с бройките знаем текущата им стойност, тоест можем да обновим отговора). Като стигнем до непроменен елемент, вече всички след него (независимо дали променени или непроменени) ще са по-малки или равни. Така можем да спрем обхождането на сортирания масив, тъй като със сигурност сме намерили максимума (или от текущия непроменен, или от обходените променени, вземайки реалната им стойност от масивите с бройките).

Разбира се, както и при горното решение, е възможно за някоя фиксирана точка да се наложи да обходим целия масив (тъй като всичките му елементи са променени). Това, обаче, ще се случи най-много два пъти, тъй като за да се е променил някой от елементите му, трябва някой знак да е сочил както елемента, така и текущата фиксирана точка. Така отново получаваме O(N) сложност за всяка фиксирана точка, но също O(N) амортизирана и за *всички* фиксирани.

**O(N)**

Можем да свалим теоретичната сложност дори повече, като забележим, че координатите са относително малки. Вместо да ползваме стандартно сортиране, можем да направим counting sort, постигайки O(L), където L е разликата между най-големите и най-малките координати (тоест, при тези ограничения L <= 4,000,000). От гледна точка на сложностите е малко грешно да наричаме това решение O(N) (при положение, че е O(L)), но тъй като заделянето на парче памет е значително по-бърза операция на клетка от стандартните (тоест да заделим масив с 4 милиона клетки е доста по-бързо, отколкото да направим 4 милиона други операции), в случая скоростта ще се доминира по-скоро от останалите O(N) операции, отколкото от тези O(L). Реално, обаче, тъй като разликата е доста малка, не се забелязва особено подобрение във времената между това и "бързото" O(N \* logN) решене.

**Заключение**

Това е една от по-лесните задачи, които съм давал. Единственото сложно беше да се навържат няколко прости структури данни, така че да ни дават бързо query и update на специфичния проблем, който имахме (грийдито). Разбира се, по-неопитните състезатели можеше да се заблудят и от това, че макар и \*някои\* от ъпдейтите да включват много точки (потенциално всички без една), то \*амортизирано\* те ще са доста бързи.

*Автор: Александър Георгиев*