АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА
MatrixSearch

Както се споменава в условието, това е относително стандартна задача от интервюта за работа като програмист. В оригиналната задача, обаче, си представяте, че матрицата от числа някак си просто ви е дадена в паметта. Това, за съжаление, няма как да стане на състезание, като четенето на матрицата изисква O(N * M) само по себе си и обезмисля задачата (тъй като докато четете числата, можете и да проверявате дали търсеното се съдържа сред тях).

В нашата версия сме добавили допълнителен тип заявка "увеличаване", която е така направена, че да запазва основното свойство на матрицата – числата както в редовете, така и в колоните, да са в строго нарастващ ред. Сега вече можем да зададем повече въпроси, като четенето на първоначалната матрица вече не доминира сложността.
Съществуват много възможни решения. Тук сме изброили някои от тях (но има и други!).

Да правим точно каквото ни казват
Най-простото решение е просто да имплементираме операциите по най-тъпия възможен начин – с по два вложени цикъла. Така както питането за число (query-то), така и обновяването на подматрица (update-а) са със сложност O(N * M), което е относително бавно. Все пак, тази имплементация е не повече от 6-7 реда общо за query и update, така че предполагам немалко състезатели ще я напишат. Цялото решение би било със сложност O(Q * N * M) и всъщност за него са предвидени не малко точки (около 30-35).

Стандартно решение
Можем да имплементираме "стандартното" решение на оригиналната задача от интервюта. При него query-то е относително бързо, но пък не се справя с update-а на подматрица. Все пак, печелим бързи query-та, което не е малко.
Идеята е следната – тръгваме от горния десен ъгъл на матрицата. Ако числото в текущата клетка е по-голямо от числото, което търсим, вървим наляво. Ако пък е по-малко, вървим надолу. Така един вид образуваме нещо като стъпаловиден диагонал, с който се движим към клетката, в която потенциално ще е числото, което търсим. Нека разгледаме оригиналната матрица от условието на задачата и да кажем, че търсим числото 18. С ***bold и italic*** сме отбелязали пътя, по който би минало нашето търсене:

```
11 13 14 17
13 15 19 21
18 20 23 42
```

Чрез този тип търсене от всяка клетка се движим или надолу, или наляво, което означава, че най-много можем да минем **N** клетки надолу и **M** клетки наляво. От гледна точка на сложност, това търсене е O(N + M) – значително по-добре от досегашното O(N * M).
Все още, обаче, update-ът е O(N * M), съответно и цялото решение е O(Q * N * M) – макар и на практика да се представя значително по-добре. За това решение са предвидени около 60-70 точки.

Стандартно решение с индексни дървета
Както казахме, проблемът на горното решение е бавният update. Защо да не го оптимизираме тогава? Можем да ползваме структурата данни индексни дървета за да обновим цял подинтервал на матрицата със сложност O(logN * logM). Това е

значително по-бързо, но пък ни кара питането за стойността на дадена клетка да бъде също със сложност O(logN * logM), вместо досегашната O(1). Печелим едно, губим друго – като цяло това решение е съвсем малко по-бързо от предходното (макар и откъм сложност да изглежда по-добре: O(N * M + Q * (N + M) * (logN * logM) и би хванало 65-80 точки.

Хитро решение

Сега да разгледаме и решението за 100 точки. Вече казахме, че търсенето в стандартното решение е относително бързо – O(N + M). Как можем да запазим O(1) питането за клетка, като в същото време update-ът ни не е по-бавен от O(N + M)? Оказва се, че начина, по който обхождаме матрицата (надолу и наляво) позволява това.

За целта ще ни трябват два масива `int rowAdd[1000][1000]`, и `int colSub[1000][1000]`. При всеки ъпдейт (да кажем в клетка (R, C)) със стойност val ще прибавяме val към клетките rowAdd(R)(C), rowAdd(R)(C + 1), …, rowAdd(R)(M). Също така ще добавяме val и в colSub(R)(C), colSub(R + 1)(C), …, colSub(N – 1)(C).

Ще ни трябва и една допълнителна променлива `add`, в която ще пазим как досегашните update-и променят клетката, в която се намираме във всеки момент. Ще знаем, че когато достигнем някой ред трябва да добавим към add rowAdd(row)(col). Тъй като можем и да излезем от range-а на някое куери, което вече сме добавили в add, като мръднем наляво, ще ползваме и colSub(row)(col) да намаляваме add. По всяко време (ако сме променяли add правилно) ще можем да намерим стойността на текущата клетка като a(row)(col) + add (което е константна операция).

Нека разгледаме оригиналната матрица от условието на задачата и сме изпълнили първия update (ред 2, колона 2, стойност 4). Трите матрици ще изглеждат по следния начин:

Оригинална матрица a()()	*Допълнителна rowAdd()()*	*Допълнителна colSub()()*
11 13 14 17	0 0 0 0	0 0 0 0
13 15 19 21	0 4 4 4	0 4 0 0
18 20 23 42	0 0 0 0	0 4 0 0

Ако вземем пътя от предходния пример (макар че той би бил различен ако взимаме в предивд add) ще увеличим add от 0 на 4 в момента, в който се придвижим от 17 в 21, а после ще го намалим обратно на 0 в момента, в който преминем от 20 в 18.

Изпълнявайки втория ъпдейт (ред 1, колона 3, стойност 7) матриците биха изглеждали по следния начин:

Оригинална матрица a()()	*Допълнителна rowAdd()()*	*Допълнителна colSub()()*
11 13 14 17	0 0 7 7	0 0 7 0
13 15 19 21	0 4 4 4	0 4 7 0
18 20 23 42	0 0 0 0	0 4 7 0

(Забележете, че от примера горе не си личи, но ако по някое време два от ъпдейтите имат един и същ ред или една и съща колона, числата в rowAdd()() или colSub()() съответно ще съдържат сумата на стойностите на ъпдейтите.)

Това решение е със сложност O(N * M + Q * (N + M)) и хващща 100 точки.

Автор: Александър Георгиев