**Анализ**

Едно наблюдение, което ще използваме е, че ако даден квадрат има център *(а, b)* и дадена точка има координати *(x, y)*, то минималната страна на квадрата, при която точката ще бъде покрита от него, е .

**Първа подзадача.** За всяка заявка разглеждаме всички точки. За всяка точка разглеждаме всички центрове и дали . Ако е вярно за поне един център, точката е покрита. Сложност – O( N\*M\*Q ).

**Втора подзадача.** За всяка заявка прилагаме неоптимизираната версия на метящата права. Сортираме квадратите заедно с точките спрямо абцисата. Когато обработваме нача-ло или край на квадрат, съответно добавяме или махаме краищата на интервала от точки, които заема спрямо ординатната ос в някакво множество. След това го сортираме и нами-раме интервалите от точки, които са покрити. Когато обработваме точка, проверяваме дали попада в някой от покритите интервалите с двоично търсене.   
Сложност – O( Q\*( N2\*log2(N) + M\*log2(N) ) ).

**Трета подзадача.** Отново прилагаме техниката на метящата права, но този път оптимизираната версия – със сегментно дърво. Вместо всеки път да сортираме множе-ството от интервали поддържаме два вида заявки – добавяме/изваждаме единица от всеки елемент в даден интервал (при обработка на начало и край на квадрат) и проверка дали даден елемент е положителен, когато искаме да видим дали дадена точка е покрита. Сложност – O( Q\*(N+M)\*log2(R) ), където R е разликата между най-ниската и най-високата точка/център.

**Четвърта подзадача.** Подобравяме алгоритъма от трета подзадача с компресия. За целта е най-подходящо да използваме map.

**Пета подзадача.** Трябва да намерим начин, който да не зависи толкова много от броя на заявките. За всяка точка намираме най-малкото S, при което бива покрита от ня-кой квадрат. Нека тази стойност е T[i], където i е номера й по реда на въвеждане. След това сортираме T[] масива и за всяка заявка с двоично търсене намираме броя на точките, чиято Т[] стойност е по-малка или равна на даденото S. Как обаче я намираме? Преди за-явките предварително за всяка точка разглеждаме всички центрове и намираме възможно най-малката стойност на . Сложност – O( N\*M+Q\*log2(M) ).

**Шеста подзадача.** Допълнителното ограничение ни позволява да оптимизираме момента на намиране на Т[] стойностите. Сортираме центровете по , което ще ги сортира също и по . За всяка точка прилагаме двоично търсене по отговора. Искаме да разберем дали за фиксирано S` текущата точка е покрита от поне един квадрат. С ново двоично тър-сене намираме първия и последния от сортираните центрове, за който . Та-ка получаваме интервал от центрове. По същия начин намираме интервала от центрове, за които . Ако двата получени интервала имат поне един общ център, има квадрат, който да покрива точката за фиксираното S`.  
Сложност – O( M\*log2(R)\*log2(N) + Q\*log2(M) ), където R е разликата между най-отдалечените точки/центрове.

**Седма подзадача.** Отново сортираме по центровете и използваме двоично търсене по отговора за всяка точка. Както в шеста подзадача намираме първия и последния от сортираните центрове, за който . Сега разглеждаме всички b-та от намерения интервал и искаме да намерим такова, за което да е изпълнено неравенството . Но те не са сортирани и не може да използваме двоично търсене. Ако пък ги сортираме ще изгубим информацията по . Затова ще използваме една доста популярна структура – merge sort дърво. Във всеки връх пазим сортирани b-тата от съответния интервал. Сега разбиваме намерения интервал от b-та по дървото и във всеки от върховете, които го формират пускаме друго двоично търсене, всяко от които да проверява дали има b, такова че . Общата сложност е   
O( M\*log2(R)\*( 2\*log2(N)+log2(N)\*log2(N) )+Q\*log2(M) ), макар и средната сложност на заявка по дървото да е по-малка log2(N)\*log2(N). Има и други решения, които на теория са по-бързи, но поради използването на sort() log(R) на брой пъти се бавят повече.

*Автор: Александър Гатев*