АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА
ПРОЗОРЕЦ

Елементарно се съобразява, че дължината на страната на отвора трябва да бъде четно число – в противен случай няма как в квадрата да се поместят точно цяло число тухли.

Решение със сложност O(N^4)

Това е наивното решение, при което се генерират всички възможни квадрати с горен ляв ъгъл във всяко квадратче и четна дължина на страната и се проверява дали някоя от страните на генерирания квадрат не разполовява някоя тухла. Ако не разполовява, то такъв отвор удовлетворява условията на задачата, сравнява се дължината на страната му с текущо намерената максимална страна и т.н. Тук единствено трябва да се ограничават циклите до квадратчета, които са допустими за горен ляв ъгъл на квадрат, който не се допира до страната на стената и до допустима дължина на страната при избран горен ляв ъгъл. Такъв алгоритъм е реализиран във файл **window_n4.cpp** и решава подзадача 1.

Решение със сложност O(N^3)

При това решение се работи с точките, които са възли в мрежата от квадратчета на стената. Разглеждат се всички хоризонтални и вертикални отсечки, които минават само по страните на тухли без да разполовяват тухла. Такава отсечка ще наричаме максимална, ако не може да бъде продължена наляво или надясно(за хоризонтална) или надолу или нагоре(за вертикална) без да разполови някоя тухла или да излезе извън стената. Идеята е, че разглеждаме всеки два реда от точки, между които има четен брой квадратчета (четна дължина на евентуален квадрат) и за тях търсим непрекъснати вертикални отсечки, които ги пресичат, разстоянието между тях е същото като разстоянието между редовете и между пресечните точки на вертикалните отсечки с всеки от двата реда лежат непрекъснати хоризонтални отсечки. За целта трябва да можем бързо (за константно време) да отговаряме на два въпроса: лежат ли две точки, които са на една хоризонтала на една и съща хоризонтална отсечка или не; лежат ли две точки, които са на една и съща вертикала на една и съща вертикална отсечка или не. Това може да се постигне като се номерират от една страна всички максимални хоризонтални отсечки с числа от 1 до техния брой и всички максимални вертикални отсечки с числа от 1 до техния брой. За всяка точка се запомня номерът на максималната хоризонтална отсечка и номерът на максималната вертикална отсечка, на които принадлежи (в авторовото решение това става в два двумерни масива – *hl[i][j]* съдържа номера на макс. хоризонтална отсечка, на която лежи точка с координати *(i,j)*, а *vl[i][j]* – номерът на съответната макс. вертикална отсечка). Тази номерация позволява бързо да се отговаря на въпроса – могат ли четири точки, които са върхове на квадрат, да бъдат върхове на отвор, при правенето на който няма да се счупи нито една тухла – отговорът е: могат, ако горните две лежат на една и съща макс. хоризонтална отсечка (т.е. за тях в масива *hl* има едни и същи стойности) и долните две лежат на една и съща макс. хоризонтална отсечка, и левите две лежат на една и съща макс. вертикална отсечка (т.е. за тях в масива *vl* има едни и същи стойности), и десните две лежат на една и съща макс. вертикална отсечка.

И така вземайки два реда с четно разстояние между тях, започваме да циклим отляво на дясно и да правим квадрати със съответната лява страна и дясна страна, равна на стълба на лявата + разстоянието между двата реда. За всеки от тях проверяваме дали

става за отвор по описания по-горе начин. Такъв алгоритъм е реализиран във файл **window_n3.cpp** и решава подзадачи 1 и 2.

Решение със сложност O(N^2logN)

При това решение отново се се работи с точките, които са възли в мрежата от квадратчета на стената. Използват се и максималните хоризонтални и вертикални отсечки, както и масивите *hl* и *vl* от предното решение. Правят се и масиви *hlbeg, hlend, vlbeg u vlend*, в които за всеки номер максимална отсечка се помнят началото и края и (съответно за хоризонталните – номерата на началния и крайния стълб, а за вертикалните – номерата на началния и крайния ред). Разглеждайки точките като потенциални кандидати за горен, ляв ъгъл на кадрат, ги обхождаме по диагонали (главен и успоредни на него) от ляво надясно и отгоре надолу. Когато сме в дадена точка, определяме най-крайната надясно от нея хоризонтална точка, която може да е горен, десен ъгъл на квадрат (тя е на разстояние от разглежданата, равно на по-малката от двете дължини – от точката до десния край на макс. хоризонтална отсечка, на която тя лежи и от точката до долния край ма макс. вертикална отсечка, на която тя лежи). Ще казваме, че точката, която разглеждаме „покрива“ всички точки от диагонала, по който се движим, които попадат под определената хоризонтална отсечка. Ще казваме, че една точка от диагонала е все още активна, ако, при движението си по диагонала, все още не сме излезли от зоната, която тя покрива.

И така при движението по всеки диагонал ще поддържаме структура с активните точки от него – това са точките, за които все още има шанс да се окажат горен ляв ъгъл на квадрат, годен за отвор. Тази структура ще бъде пирамида, във върха на която ще стои точката, която покрива до най-ляв стълб (в програмната реализация това е направено чрез приоритетна опашка). Освен тази, структура ще поддържаме и структура с номерата на активните точки (номерата се определят от реда, по който ги обхождаме по диагонала). Каква е тази структура и за какво служи ще стане ясно след малко. И така, нека при обхождането се намираме в точка с номер *k* от диагонала (първата точка от диагонала е с номер 1, втората с номер 2 и т.н.) . Естествено, че по този номер (и по диагонала, на който се намираме, естествено) лесно можем да определим реда и стълба, на който се намира точката. В случая ни интересува стълба. По неговия номер можем да определим кои от вече обходените по диагонала точки стават неактивни и да ги извадим от приоритетната опашка и от структурата с номерата на активните точки. Забележете, че след тази операция в приоритетната опашка остават само точки, които покриват разглежданата. След това определяме на квадрат с каква максимална страна може да бъде *долен ляв ъгъл* разглежданата точка и какъв е номерът (от обхождането по диагонала) на точката, която би била горен десен ъгъл. И в структурата с номерата на активните точки търсим първата точка, която е с не по-малък от определения номер. Ако такава точка съществува, то тя ще бъде горен ляв ъгъл, а разглежданата – долен десен ъгъл на квадрат, който може да бъде отвор за прозорец. Изчисляваме страната му и нататък действаме по обичайния начин с текущия максимум. Каква трябва да бъде структурата с номерата на активните точки, за да можем двоично да отговаряме на въпроса – кое е първото число в структурата, не по-малко от зададено число. Освен това трябва да можем със сложност *log* да добавяме и премахваме елементи. Такава структура е двоично балансирано дърво, като най-удачно е да се използва *set* от STL.

И тъй всяка точка (почти всяка, защото тези, които не могат да бъдат горен ляв ъгъл на квадрат няма какво да правят там) от диагонала ще влезе и излезе по веднаж от приоритетната опашка и ще влезе и излезе в балансираното дърво с номерата. За всеки диагонал се започва с празна приоритетна опашка и балансирано дърво и в един

диагонал O(N) елемента. Тогава, обхождайки всички точки по указания по-горе начин, получаваме алгоритъм със сложност $O(N^2 \log N)$. Такъв алгоритъм е реализиран във файл **window_n2logn.cpp** и решава всички подзадачи.

Решение на Искрен Чернев

Интересна идея за решение с подобна сложност предлага Искрен Чернев. Той предлага две решения – идеята на първото е като на горното решение със сложност $O(N^2 \log N)$ и той го нарича бавно. Второто е със сложност нещо като $O(N^2 \log(\log N))$ и е наречено бързо. И двете на практика не дават по-бързи решения от изложеното по-горе, но представляват интерес (особено второто с disjoint-set) и затова поместваме анализа им тук. Реализацията на двете е във файл **window-Iskren.cpp**, като програмата реализира едното или другото решение според използвания макрос в началото на програмата.

За всеки диагонал се интересуваме за всяка клетка колко е полуквадрата долу-дясно и колко е горе-ляво. Полуквадрат е просто възможност да отрежем линии долу-дясно или горе-ляво с даден размер. Например долу-дясно полуквадрат на позиция 2 с големина 2, и горе-ляво полуквадрат на позиция 4 с големина 3 образуват един квадрат с големина 2. Но долу-дясно полуквадрат на позиция 2 с големина 2 и горе-ляво полуквадрат на позиция 5 с големина 3 не образуват нищо (защото долу-дясо полуквадрата е малък -- ако беше 3 щяха да образуват 3). Сега въпросът е как да намерим полуквадратите които се засичат най-бързо. Бавното (N^2 log N) решение пази във всеки момент кои са отворените (т.е тези които още стигат до текущата позиция) долу-дясно полуквадрати. За текущата позиция j, взимаме горе-ляво полуквадрата, нека той има размер A. Това значи, че се интересуваме от отворен долу-дясно полуквадрат на позиции между j - A и j - 1. За целта пазим в дърво всички отворени позиции, и търсим първата по-голяма от j – A (stl set + lower_bound). Когато един квадрат се затваря просто го махаме от дървото.

За бързото решение, това което променяме, е че пазим отворените квадрати като disjoint-set-ове. Когато един квадрат се затвори просто го сливаме с множеството на следващия полуквадрат. Така всяко множество съответства на отворен полуквадрат, а всеки затворен имплицитно сочи към следващия "кандидат", т.е следващия отворен след него (най-малкото число, по-голямо от него, ако си мислим за дърво). Друг начин да си мислим за бързото решение е, че пазим свързан списък с всички долу-дясно полуквадрати. Когато ни трябва най-големия отворен между j-A и j-1 просто взимаме j-A полуквадрата. Ако той е отворен го ползваме. Ако е затворен проверяваме j-A+1, j-A+2 итн. Само че веднъж като сме минали през j-A+1, j-A+2 итн и сме се спряли на B запомняме, че за всички затворени полуквадрати по пътя (j-A, j-A+1, j-A+2,...B+1) следващия отворен е B. Ако B се затвори в бъдеще, просто ще проследим неговия указател към потенциално следващия отворен.

Автори: Руско Шиков, Йордан Чапъров, Георги Георгиев, Искрен Чернев