Az előző részben megismerkedtünk a visszalépéses kereséssel, és alkalmaztuk a módszert az *n-vezér* probléma megoldására. A feladatban egy $n \times n$-es sakktáblán helyeztünk el vezéreket úgy, hogy azok ne üssék egymást. A megoldás abból állt, hogy sorrendben végighaladtunk az oszlopokon, és minden oszlopban megpróbáltunk úgy elhelyezni egy vezért, hogy ne álljon ütésben egyik lerakott vezérrel sem. Amikor ez egy oszlopban sikerült, akkor továbbléptünk a következő oszlopra, ha nem sikerült, akkor visszaléptünk az előző oszlopra. A feladat a visszalépéses keresés könnyű iskolapéldája volt, hiszen egy adott oszlopon belül n sorba tudtunk lerakni egy vezért, vagyis egy adott állásból könnyű volt a következőre, vagy az előzőre lépni.

Nézzünk meg egy kicsit összetettebb feladatot. A szeptemberi számban kitűzött **K. 508.** feladatban egy olyan $N \times N$ méretű falat kellett 1×2-es téglákból megépíteni, amelyet nem lehet vízszintes vagy függőleges vonallal két részre osztani úgy, hogy a vonal ne vágjon el egy téglát sem. Az eredeti feladatban egy ilyen, „földrengésbiztos" falat kértek 8×8-as méretben. Általánosítsuk a feladatot úgy, hogy $N \times M$ méretű, földrengésbiztos falat kívánunk építeni, szintén 1×2-es téglákból. Nézzük meg, hogy milyen N és M esetén adódnak megoldások, illetve mennyi azok száma.

Gondoljuk végig, hogy milyen eseteket kell megnéznünk. Biztosan tudjuk, hogy $\frac{N \cdot M}{2}$ téglát kell majd elhelyeznünk, mindegyiket vízszintesen, vagy függőlegesen. Végeredményként az összes tégla elhelyezésének módját szeretnénk tudni, tehát azt, hogy vízszintesen vagy függőlegesen áll, illetve hogy a fal mely részén helyezkedik el. Ezekre az adatokra a program futása során is szükségünk van. A visszalépéses keresést akarjuk alkalmazni, ezért ki kell találnunk a téglák elhelyezésének sorrendjét. Mivel a téglák nincsenek sorszámozva, ezért két elhelyezés azonos, ha abban két téglát kicserélünk. Hogy ne számoljuk meg az ilyen lehetőségeket többször, állapodjunk meg a következőkben:

1. minden téglát úgy helyezünk el, hogy az vagy fölfelé (a következő sorba), vagy tőle jobbra (a következő oszlopba) nyúlik át;
2. az első téglát a bal alsó sarokba rakjuk le, tehát kerüljön a tégla bal vagy alsó négyzete a falban az 1. sor 1. oszlopába;
3. minden más tégla bal alsó négyzetét helyezzük
   - 3.1. az előző tégla bal alsó négyzetének sorában a következő üres oszlopba, vagy ha nincs, vagy oda nem helyezhető el, akkor;
   - 3.2. az előző tégla bal alsó négyzete fölötti sorban az első üres oszlopba, vagy ha nem lehet;
   - 3.3. az előző tégla bal alsó négyzete fölötti második sorban az első üres oszlopba.

Így lehetőségünk lesz arra, hogy egy elhelyezett tégla után a következő helyét megtaláljuk. Könnyen igazolható, hogy a téglák lerakásának fenti szabályaival egy adott elhelyezés pontosan egyféleképp jöhet létre, és minden elhelyezés létrejön. Vagyis a tégla bal alsó négyzete helyének megválasztásával kapunk egy természetes sorrendet. Egy tégla egy adott helyen kétféleképp is elhelyezhető, ezért ezt a két lehetőséget is sorba állítjuk: legyen mindig a vízszintes az első, amit kipróbálunk, és a függőleges a második.

Ezek alapján egy tégla tulajdonságait a Tégla adatszerkezettel adjuk meg, amely a bal alsó négyzetének sor és oszlop értéke, valamint az elhelyezése, ami vagy azt mutatja, hogy nincs elhelyezve, vagy vízszintes, vagy függőleges.

Tégla rekord
  sor, osz : Egész
  el: (nincs, vizsz, fugg)
Tégla rekord vége

Az előbbi struktúrából összesen $\frac{N \cdot M}{2}$ darabra lesz szükségünk, tehát hozzunk belőle létre egy tk (téglák) tömböt. A megoldás úgy dolgozik majd, hogy a tk tömb elemeit igyekszik sorrendben feltölteni olyan módon, hogy egyetlen elhelyezés se ütközzön korábbi elhelyezésekkel. Két tégla nyilván úgy kerülhet egymást kizáró helyzetbe, hogy a falban azonos részt is elfoglalnak. Hogy ez ne forduljon elő, vezessünk be egy fal nevű, $N \times M$ méretű tömböt, amely az algoritmus működése közben mindig mutatja, hogy a falra helyezett négyzetháló melyik négyzete üres, és melyik foglalt.

Ha csak egyszerű falat kellene építenünk, akkor a fenti adatok elegendőek lennének az algoritmus működéséhez. Most azonban földrengésbiztos falat szeretnénk készíteni, tehát figyelnünk kell arra is, hogy a téglák élei ne hozzanak létre a falon belül egy teljes hosszúságú vonalat. Ezt legegyszerűbben úgy tudjuk elérni, hogy minden falon belül lévő szakasznál számoljuk, hogy az eddigi téglák milyen hosszú vonalat alkotnak. Ha az N sorból és M oszlopból álló falat egység oldalú négyzetekre bontjuk, akkor minden falon belüli vízszintes vonalra legföljebb $M-1$ összes hosszúságú téglaél eshet. Analóg módon hasonló feltétel kell teljesüljön minden függőleges, belső oszlopokon fekvő vonalra. Ezeket az adatokat egy svonal és egy ovonal tömbben tároljuk. Mindkettő i-edik értéke jelentse az i-edik sor fölötti, illetve i-edik oszloptól jobbra eső vonalon lévő téglaoldalak hosszát. A megoldás az előbb leírt adatok kezdeti értékének beállításával indulhat:

Előkészítés eljárás
  fal[] := szabad

sorvonal[] := 0
ovonal[] := 0
tk[] := (sor: 0, osz: 0, el: nincs)
db := 1
Előkészítés eljárás vége

Az utolsó sorban szereplő db változó megmutatja, hogy a visszalépéses keresés hányadik téglánál tart. Persze nem olyan egyszerű a dolgunk, hogy csak db értékét növeljük vagy csökkentjük előre- vagy visszalépéskor. Egy előrelépéskor ugyanis meg kell állapítanunk, hogy hova kerülhet a következő tégla. A korábban megfogalmazott sorrend szerinti elhelyezés alapján ezt a következő logikai fügvénnyel valósítjuk meg. A függvény mellékhatásként be is állítja a db-edik tégla helyét:

KövHely függvény: Logikai
Ha db=1 **akkor**
tk[db].sor := 1 : tk[db].osz := 1
KövHely := igaz
Különben
sor := tk[db-1].sor : osz := tk[db-1].osz
Ciklus
osz := osz+1
Ha osz>M **akkor** sor : = sor+1 : osz := 1
Ciklus amíg sor≤N és fal[sor][osz] = fogl
Ha sor≤N **akkor**
tk[db].sor := sor : tk[db].osz := osz
KövHely := igaz
Különben
KövHely := hamis
Elágazás vége
Elágazás vége
KövHely függvény vége

A megoldás fő részét, a visszalépéses keresést, szintén logikai függvényként adjuk meg: ha talál megoldást, akkor igaz, egyébként hamis értékkel tér vissza. Megoldás esetén a tk táblázatban megtalálható a téglák elhelyezése.

Backtrack függvény: Logikai
Ciklus amíg 1≤db **és** db≤$\frac{N \cdot M}{2}$
Elágazás tk[db].el **szerint**
nincs:
Ha KövHely **és** (LeVízsz **vagy** LeFügg) **akkor**
db := db+1
Különben
Vissza
Elágazás vége
vizsz:
Felvesz
Ha LeFügg **akkor**
db := db+1
Különben
Vissza
Elágazás vége
fugg:
Felvesz
Vissza
Elágazás vége
Ciklus vége
Backtrack := db>$\frac{N \cdot M}{2}$
Backtrack függvény vége

Az algoritmus a fent leírtakat valósítja meg, de azért nézzük részleteiben. Nyilván akkor hagyja abba a keresést, amikor az $\frac{N \cdot M}{2} + 1$-edik téglával kezdene foglakozni (tehát az összes többit jól elhelyezte), vagy amikor az első téglával sem boldogult, és visszalép, vagyis nem talált megoldást. A keresés fő ciklusán belül egy adott db-edik téglát kell elhelyezni. Ha ez a tégla még nincs lerakva, akkor először helyet keresünk neki, majd ha ez sikerül, akkor letesszük

vízszintesen, illetve ha ez utóbbi nem sikerül, akkor függőlegesen. A LeVízsz szintén egy mellékhatásos függvény, amely megvizsgálja, hogy a db-edik tégla (amelynek már találtunk helyet) elhelyezhető-e vízszintesen. Ha igen, akkor beállítja a tégla elhelyezését és lefoglal számára helyet a falban, majd igaz értékkel tér vissza. Hasonlóan dolgozik a LeFügg függvény. Több feltételt is meg kell vizsgálni: a tégla csak a fal szabad helyére kerülhet, és nem hozhat létre a lerakása vízszintes vagy függőleges vágást. Amennyiben a feltételek teljesülnek, akkor lerakja a téglát, vagyis módosítja az adatokat.

LeVízsz függvény: Logikai
  sor = tk[db].sor : osz = tk[db].osz
  Ha osz+1≤M **és** fal[sor][osz+1]=szabad **és**
    (osz+1=M **vagy** ovonal[osz+1]+1<N) **és**
    (sor=N **vagy** svonal[sor]+2<M)
  akkor
    fal[sor][osz] := fogl : fal[sor][osz+1] := fogl
    Ha osz+1<M **akkor** ovonal[osz+1] := ovonal[osz+1]+1
    Ha sor<N **akkor** svonal[sor] := svonal[sor]+2
    LeVízsz := igaz
  Különben
    LeVízsz := hamis
  Elágazás vége
LeVízsz függvény Vége

Ez a függvény is a korábban leírtak szerint működik. Az ovonal és svonal tömbök az adott oszlop, illetve sor és az utána következő közötti határvonalon lévő, téglával határolt részek hosszát mutatják. Ennek megfelelően N sor esetén az ovonal tömb egyik értéke sem lehet egyenlő N-nel, hiszen az éppen egy vágást jelentene. Természetesen az N-edik sor és az M-edik oszlop esetén nincs feltétel, mert itt már a fal határához értünk. A LeFügg függvény hasonló módon elkészíthető.

Nem valósítottuk még meg a Felvesz eljárást, aminek éppen az lesz a feladata, hogy a korábbi adatértékeket törölje, amikor pl. vízszintesből függőlegesre állítjuk a téglát, vagy függőleges helyzetből levesszük, mert visszalépünk. Ennek az elkészítése is nagyon egyszerű, hiszen csak szabaddá kell tenni a fal-ban a tégla által elfoglalt két négyzetet, valamint csökkenteni a svonal és ovonal megfelelő értékeit.

Felvesz eljárás
  sor = tk[db].sor : osz = tk[db].osz
  Ha tk[db].el=vizsz **akkor**
    fal[sor][osz] := szab : fal[sor][osz+1] := szab
    Ha osz+1<M **akkor** ovonal[osz+1] := ovonal[osz+1]-1
    Ha sor<N **akkor** svonal[sor] := svonal[sor]-2
  Különben
    fal[sor][osz] := szab : fal[sor+1][osz] := szab
    Ha osz<M **akkor** ovonal[osz] := ovonal[osz]-2
    Ha sor+1<N **akkor** svonal[sor+1] := svonal[sor+1]-1
  Elágazás vége
Felvesz eljárás Vége

A korábban elkészített KövHely, valamint a LeVízsz és LeFügg függvények nem csak megkeresték a db-edik tégla helyét és állását, hanem be is állították a tk és fal mennyiségeket, ezért a visszalépéses keresés ciklusában az előrelépés csupán a db mennyiség növelését jelentette. A Backtrack függvényben szereplő Vissza eljárás viszont a tk[db] adatait állítja alaphelyzetbe, valamint csökkenti db értékét. Visszalépés előtt az előbb megírt Levesz függvény szintén gondoskodik arról, hogy a fal mindig az aktuális tk értékekkel összhangban legyen. Így gyakorlatilag kész vagyunk.

Természetesen érdemes még az eredményeket megjelenítő eljárást készítenünk, amely megmutatja a talált megoldásokat. Pl. a téglákat sorszámuk utolsó számjegyével jelölve egyszerűen kiírhatók szöveges képernyőre vagy fájlba az eredmények. Nagy számú megoldás esetén persze már nem csak a megoldások, hanem azok száma is érdekel minket, így olyan főprogramot érdemes készíteni, ami számolja, hogy hányszor sikerült a Backtrack függvénynek igaz értékkel visszatérni. Egy lehetséges megoldás a következő eljrárás:

Falkészítés eljárás
  megoszám := 0
  Előkészítés
  Ciklus amíg db≥1
    Ha Backtrack **akkor**
      Falkiírás

```
        megoszám := megoszám+1
        db := db-1
      Elágazás vége
    Ciklus vége
Falkészítés eljárás Vége
```

Amennyiben egy megoldást találtunk, akkor elég a db mutatót csökkenteni, és a Backtrack függvényt újrahívni, hogy az egy újabb megoldást keressen, hiszen ilyenkor a visszalépéses keresés úgy folytatódik, hogy a db-edik téglának keres következő elhelyezést.

Érdekes a kész programmal megvizsgálni, hogy különböző N és M értékek esetén hány megoldás adódik. A legkisebb területű fal, ami földrengésbiztosan készíthető az 5×6-os, összesen 6 ilyen van. Az 5×8-asból 108, a 6×7-esből 124, míg a 6×8-asból 62. A **K. 508.** feladatban szereplő 8×8-as falból 25 506 különböző készíthető. Az 5×6-os megoldások a javasolt kiírással a következők:

```
144553  144355  124455  445523  441553  144553
189223  122390  129330  811923  891223  122903
589700  568890  889670  856900  890067  885907
566734  562774  155674  156774  155367  125667
112234  112334  122334  122334  122344  123344
```