

Programozás 2

Programozási technika: Backtracking (visszalépéses keresés).
A nyolc királynő problémája. Egyéb feladatok: útkeresés
labirintusban, sakktábla bejárása huszárral, Sudoku megfejtése.

„Hogyan tud egy algoritmus okosan visszalépni, ha zsákutcába fut?”

Visszalépéses keresés (Backtracking)

Mi az a visszalépéses keresés?

A **visszalépéses keresés (backtracking)** lényege a feladat megoldásának megközelítése rendszeres próbálgatással. Olyan keresési technika, amely megpróbál egy megoldást építeni, és ha zsákutcába fut, visszalép az utolsó döntéshez. A visszalépés megvalósításához a programban általában rekurziót használunk.

Kódvázlat:

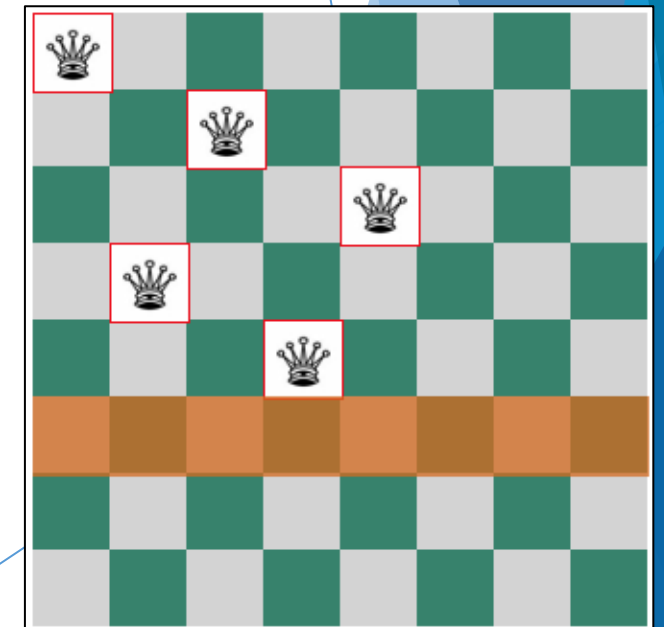
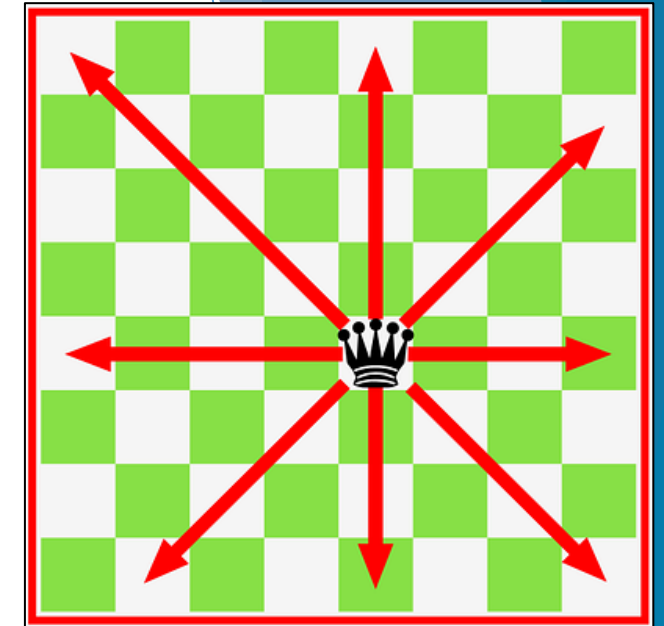
```
void továbblépés(állapot) {  
    if (megtaláltuk a megoldást) {  
        kiírjuk();  
    } else {  
        for (minden továbblépési lehetőség) {  
            if (érvényes továbblépés) {  
                továbblépés(új állapot);  
            }  
        }  
    }  
}
```

Feladat (nyolc vezér probléma): Helyezzünk el egy 8x8-as sakktáblán 8 királynőt (vezért) úgy, hogy azok ne üssék egymást!

Megoldás:

Elkezdjük keresni a feladat megoldását valamilyen rendszer alapján (pl. **mindegyik sorba megpróbálunk lerakni egy királynőt** úgy, hogy ne üsse az előző sorokba lerakott királynőket).

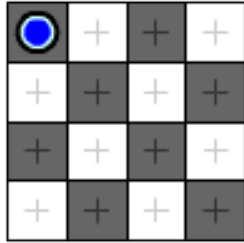
Ha valahol már nem tudunk továbblépni (valamelyik sorba már nem tudunk letenni királynőt, mert bárhova is tennénk az adott sorba, ütné az előző sorokban levőket), akkor visszalépünk egy korábbi döntési ponthoz (visszalépünk az előző sorhoz), és ott más lehetőséget választunk (ott máshova próbáljuk meg letenni a királynőt).



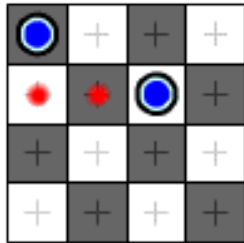
Megoldás keresésének menete egyszerűsített feladatra:

(Rakjunk le 4 királynőt egy 4x4-es táblára úgy, hogy azok ne üssék egymást.)

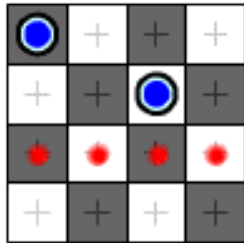
Megpróbáljuk lerakni
a királynőt az
1. sorba:



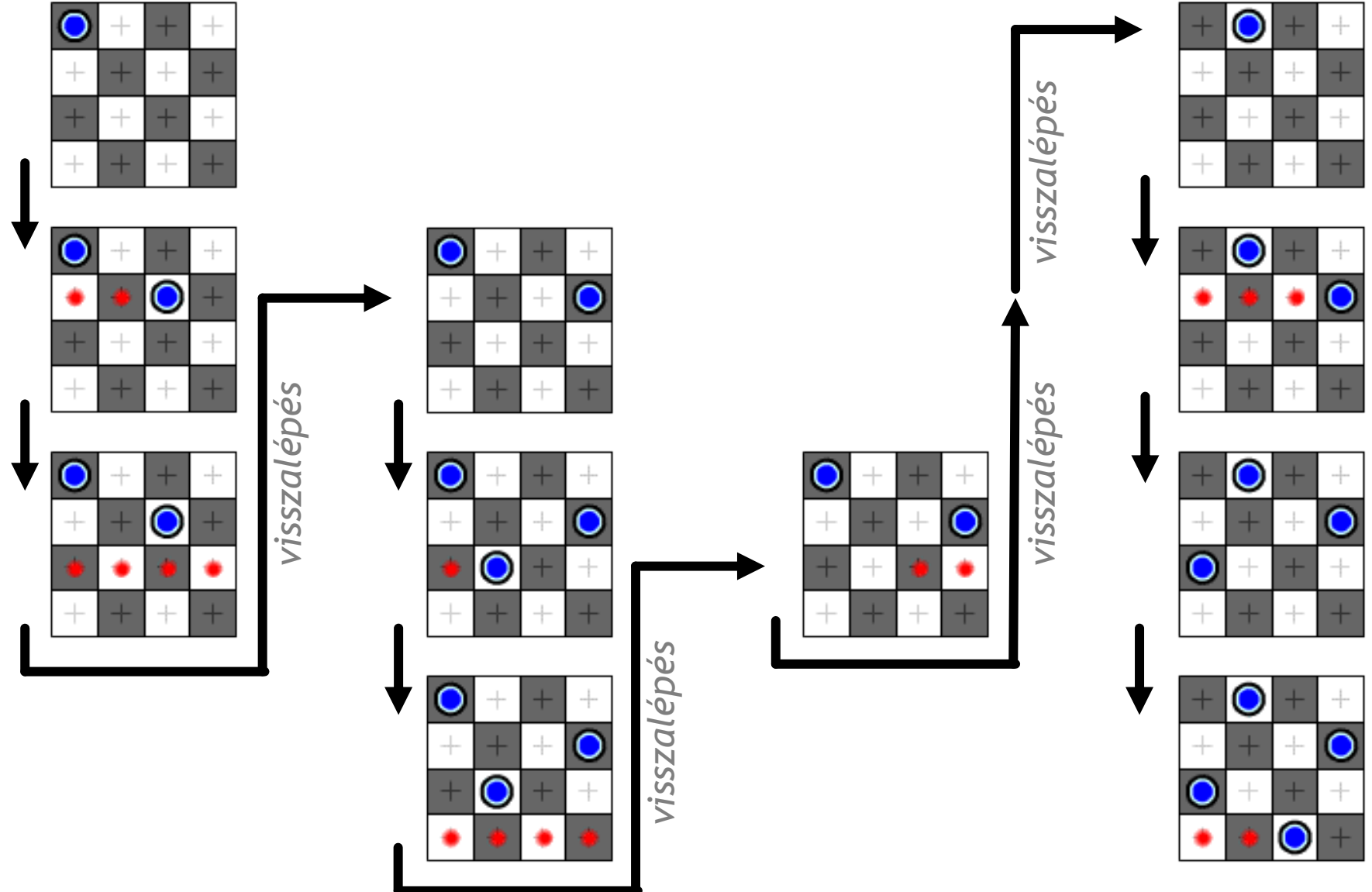
Megpróbáljuk lerakni
a királynőt a
2. sorba:



Megpróbáljuk lerakni
a királynőt a
3. sorba:



Megpróbáljuk lerakni
a királynőt a
4. sorba:



Vizualizáció a feladat megoldásának menetére 8x8 sakktáblán:

<https://liveexample.pearsoncmg.com/dsanimation/EightQueenseBook.html>

```
#include <stdio.h>

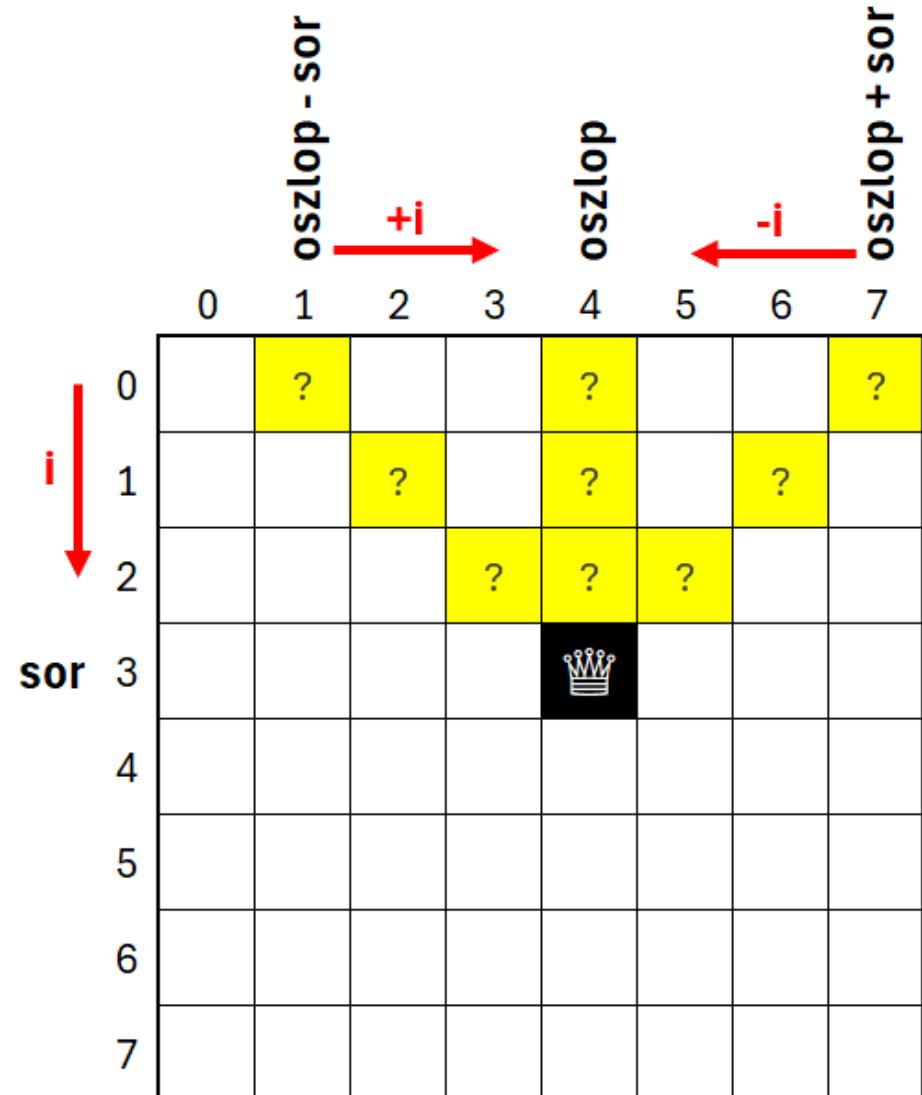
const int N = 8;

int sakktabla[N][N]; // 0 = ures mezó, 1 = kiralyo van a mezon

// matrix kiirasa (minden probalkozas utan)
void kiiras() {
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            printf("%d ", sakktabla[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

// megallapítja hogy lerakható-e a királynő a megadott [sor][oszlop]-ra

```
int letehető(int sor, int oszlop) {  
    int letehető = 1;  
    for (int i=0; i<sor; i++) {  
        // van-e már ebben az oszlopban királynő valahol felette?  
        if (sakktábla[i][oszlop]==1) {  
            letehető = 0;  
        }  
        // az "\" atloban van-e már  
        // királynő felette?  
        if (oszlop-sor+i>=0 &&  
            sakktábla[i][oszlop-sor+i]==1) {  
            letehető = 0;  
        }  
        // az "/" atloban van-e már  
        // királynő felette?  
        if (oszlop+sor-i<N &&  
            sakktábla[i][oszlop+sor-i]==1) {  
            letehető = 0;  
        }  
    }  
    return letehető;  
}
```



```
// letesz egy kiralynot az adott sor valamelyik oszlopaba (0..N-1),
// majd meghivja onmagat a kovetkezo sorra...
int letesz(int sor) {
    int oszlop = 0; // 0. oszloptol kezdjuk el probalni lerakni
    int kesz = 0; // van megoldas? (0=nincs, 1=van)
    do {
        // melyik oszlopba rakhato le?
        while (oszlop<N && letehető(sor,oszlop)==0) {
            oszlop++;
        }
        // ha talaltunk neki helyet, lerakjuk...
        if (oszlop<N) {
            sakktabla[sor][oszlop] = 1;
            printf("Királyno elhelyezése a(z) %d. sorba...\n", sor);
            kiiras();
            // majd ha van meg tovabbi sora a sakktablanak...
            if (sor<N-1) {
                // probalkozunk a kovetkezo sorral (tovabblepes)
                kesz = letesz(sor+1);
            } else {
                // ha utolso sorba is leraktuk (talaltunk megoldast)
                kesz = 1;
            }
        }
    }
}
```

```

// ha meg nincs megoldas, toroljuk az ebbe a sorba
// lerakott kiralynot az aktualis oszlopbol es
// es probaljuk lerakni a kovetkezo oszlopba...
if (kész == 0) {
    sakktabla[sor][oszlop] = 0;
    oszlop++;
}
}
} while (oszlop < N && kész == 0);
return kész;
}

```

Ez a program csak egy (első) megoldást keres meg. Hogyan lehetne módosítani, hogy az összes megoldást megkeresse?

A „kész” változóval nem állítjuk le a keresést, hanem amikor találunk egy megoldást („kész = 1” helyett), kiírjuk a megoldást, majd folytatjuk a keresést.

```

int main() {
    // kinullazzuk a sakktablat
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            sakktabla[i][j] = 0;
        }
    }
    // keressuk a megoldast
    int k = letesz(0);
    if (k==0) {
        printf("Nincs megoldas!\n");
    };
}

```


Megoldható ez a feladat backtracking használata nélkül is?

Igen, megoldható brute-force kereséssel is, de ez jóval nagyobb időbonyolultságú, mivel az összes lehetséges királynőelhelyezést a sakktáblán végig kell próbálni és ellenőrizni.

8x8-as sakktáblán ez 8 egymásba ágyazott ciklust jelentene:

```
for (int q0 = 0; q0 < 8; q0++) {  
    for (int q1 = 0; q1 < 8; q1++) {  
        for (int q2 = 0; q2 < 8; q2++) {  
            ...  
            for (int q7 = 0; q7 < 8; q7++) {  
                if (érvényes_elrendezés(q0, q1, ..., q7)) {  
                    kiírjuk();  
                }  
            }  
            ...  
        }  
    }  
}
```

Az ilyen egyszerű (naív) brute-force megközelítéssel:

Összesen $8^8 = 16\,777\,216$ lehetséges elrendezést kéne leellenőriznünk (ebbe bele tartozik olyan elrendezés is, amikor ugyanabban az oszlopban vagy átlóban több királynő is van a sakktáblán)!

Optimalizált brute-force (permutációs) megoldással:

Azokban az oszlopokban, ahol az előző sorokban már elhelyeztünk királynőt, nem próbálkozunk újra a királynő elhelyezésével. Így a keresési tér a királynők oszloppozícióinak permutációira szűkül.

Így már „csak” $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40\,320$ lehetséges elrendezést kell leellenőrizni (ebben még mindig bele tartozik olyan elrendezés is, amikor egy átlóban több királynő is van a sakktáblán).

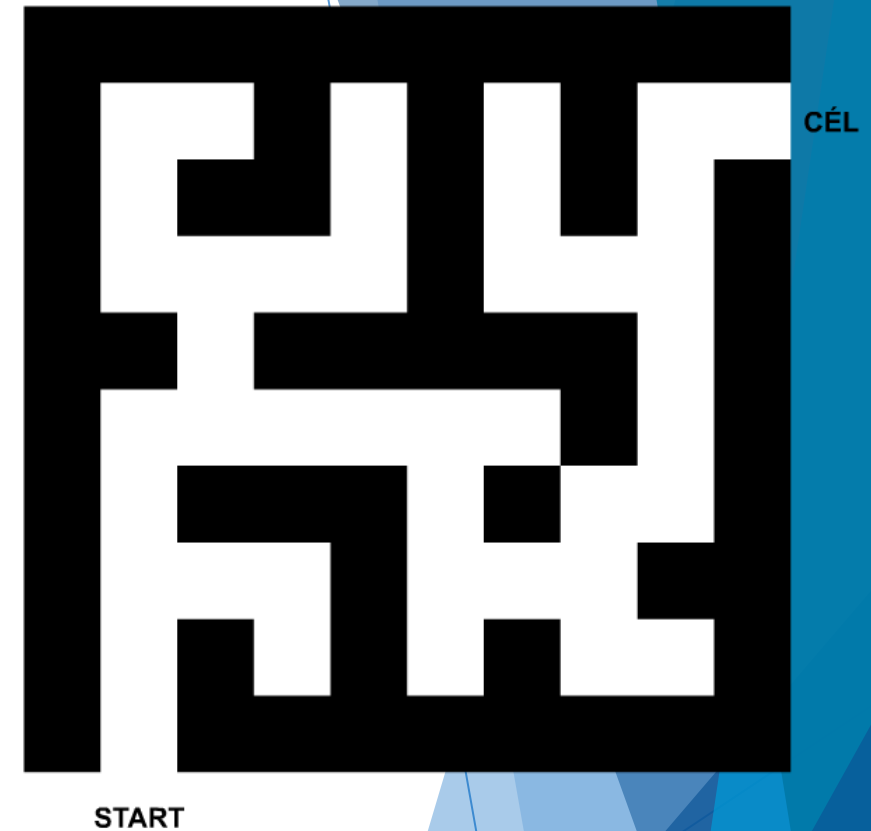
A backtracking technika alkalmazásával ez dinamikusan még tovább szűkül, csak az érvényes elrendezésekre.

Feladat (labirintus): Adva van egy labirintus, megadott bejárat és kijárat. Keressünk egy utat a bejárattól a kijáratig.

Hogyan lehet a labirintust tárolni a programban?

Attól függően milyen a labirintus szerkezete, többféle reprezentáció lehetséges. Néhány megoldás:

- ▶ Mátrix (leggyakoribb, mi is ezt fogjuk használni),
- ▶ Gráf alapú (pl. minden mezőt egy csúcsnak tekintünk, és él akkor van, ha egyik mezőből át lehet lépni a másikba),
- ▶ Koordináta-pár alapú (csak a járható mezőket tároljuk koordináta-párokként, pl: [(1,1), (1,2), (1,4), (1,6), (1,8), (1,9), (2,1), (2,4), ...]),
- ▶ ...



A mi esetünkben egy 10x10-es mátrixot használunk, melyben az egyes elemek értéke:

- Ezekon kívül a feladat megoldása közben még két értéket vehetnek fel a mátrix elemei:

- [illegible]

Kijáráshoz vezető útvonal keresésének menete:

- ▶ Elindulunk a **bejárattól** (a mátrix $[9,1]$ elemétől) és ezt megjelöljük 1-essel.
- ▶ Megnézzük, hogy mehetünk-e **felfele** (az adott hely feletti mező értéke 0 vagy 5). Ha mehetünk, akkor oda lépünk (tehát azt is bejelöljük 1-essel), majd innen próbálunk továbblépni. Ha nem mehetünk, akkor hasonlóan megnézzük a **jobbra**, majd a **lefele**, majd a **balra** irányt és abba az irányba megyünk, ahol szabad az útvonal (tehát ahol a mátrixban 0 vagy 5 van).
- ▶ Ha valahonnan egyik irányban sem tudunk már továbblépni, akkor ezt a helyet bejelöljük 2-essel (zsákutca), majd visszalépünk az előző helyre, ahonnan próbálunk más irányba továbbmenni (vagy innen is visszalépünk, ha nem vezet sehova).

[illegible]


```
// odalepunk a lapirintusba, majd keresunk tovább vezető utat
```

```
void lepes(int sor, int oszlop) {
```

```
    // eljutottunk-e a celig
```

```
    if (lab[sor][oszlop]==5) {
```

```
        kesz = 1;
```

```
    }
```

```
    // odalepunk a mezore
```

```
    lab[sor][oszlop] = 1;
```

```
    // ha nincs megoldas, akkor megprobalunk tovabblepni....
```

```
    if (kesz==0 && sor>0 && (lab[sor-1][oszlop]==0 || lab[sor-1][oszlop]==5)) {
```

```
        lepes(sor-1,oszlop); // felfele lepunk tovább
```

```
    }
```

```
    if (kesz==0 && oszlop<N-1 && (lab[sor][oszlop+1]==0 || lab[sor][oszlop+1]==5)) {
```

```
        lepes(sor,oszlop+1); // jobbra lepunk tovább
```

```
    }
```

```
    if (kesz==0 && sor<N-1 && (lab[sor+1][oszlop]==0 || lab[sor+1][oszlop]==5)) {
```

```
        lepes(sor+1,oszlop); // lefele lepunk tovább
```

```
    }
```

```
    if (kesz==0 && oszlop>0 && (lab[sor][oszlop-1]==0 || lab[sor][oszlop-1]==5)) {
```

```
        lepes(sor,oszlop-1); // balra lepunk tovább
```

```
    }
```

```
    // ha nem jutottunk megoldashoz, akkor visszlepunk
```

```
    if (kesz==0) {
```

```
        lab[sor][oszlop] = 2;
```

```
    }
```

```
}
```

	oszlop - 1	oszlop	oszlop + 1
sor - 1		? ^{1.}	
sor	? ^{4.}	1	? ^{2.}
sor + 1		? ^{3.}	

```
// foprogram
```

```
main() {
```

```
    kiiras();
```

```
    lepes(N-1,1);
```

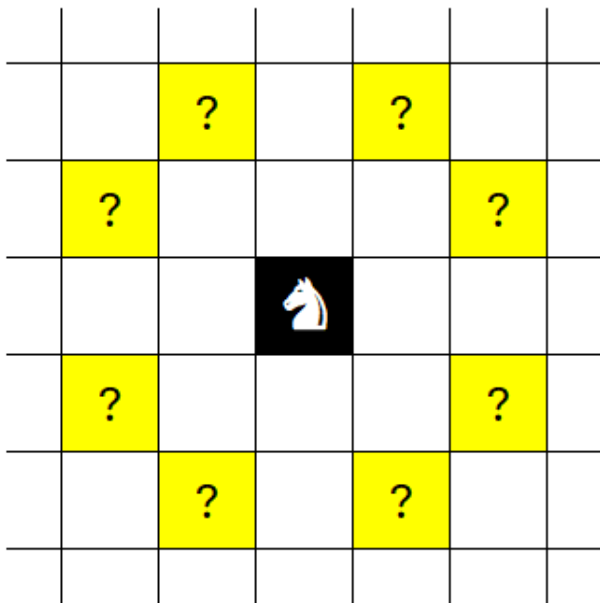
```
    kiiras();
```

```
}
```

Feladat (sakktábla bejárása huszárral):

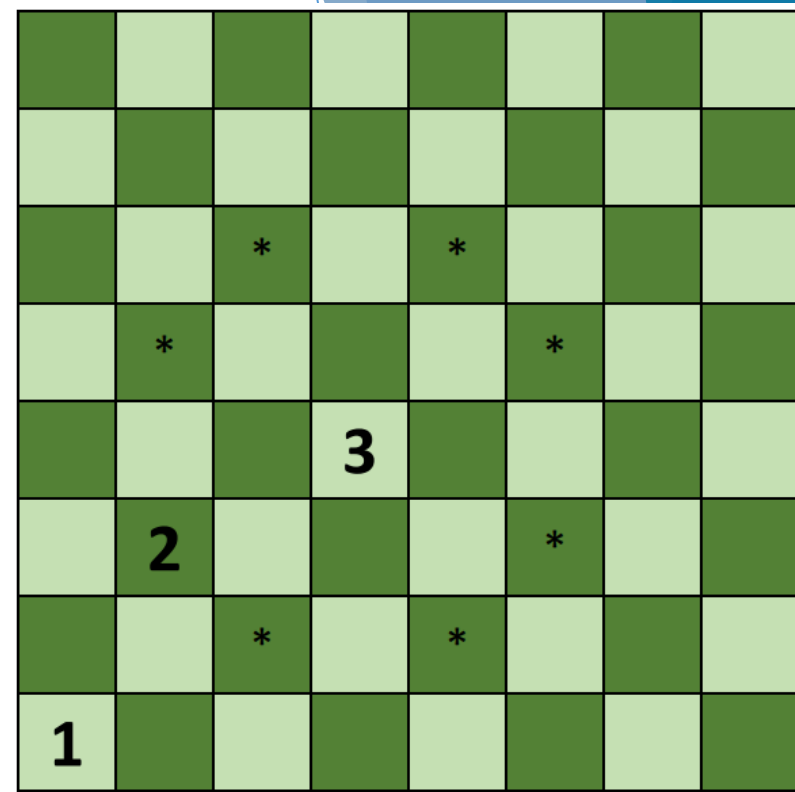
Keressünk egy olyan huszárugrás-sorozatot, amely minden egyes mezőt érint a sakktáblán pontosan egyszer (a huszár a bal alsó sarokból induljon).

Hová léphet a huszár tovább egy adott mezőről?



A megoldás során
figyelembe kell vennünk:

- ▶ Honnan jött a huszár? Oda nem léphet vissza!
- ▶ A továbblépés a sakktáblán belül esik-e? Csak a sakktáblán belüli mezőkre léphet!



A megoldás megkeresésének menete:

- ▶ A sakktáblát egy 8x8-as mátrixsal reprezentáljuk, ebben kezdetben minden elem értéke 0.
- ▶ Ahogy lépegetünk a huszárral (bal alsó sarokból kezdve), elkezdjük a mátrixba beírni sorban a számokat (1, 2, 3, ...).
- ▶ Ha valahonnan már nem tudunk sehová se továbblépni, akkor a mátrix adott elemének értékét visszaállítjuk 0-ra (visszalépés), majd megpróbálunk az előző helyről más irányba tovább lépni (vagy ha már nem tudunk onnan se továbblépni, akkor onnan is visszalépünk).

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	5	0	0
1	0	0	0	4	0	0	0	6
2	0	0	0	0	0	0	0	0
3	0	0	3	0	0	0	7	0
4	0	0	0	0	0	0	0	0
5	0	2	0	0	0	0	0	8
6	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0

```
#include <stdio.h>
```

```
const int N = 8; // sakktabla merete
```

```
// az adott helyrol hova lephetnenk tovabb...
```

```
// (mennyivel valtozik a sor es oszlopindex)
```

```
const int sorlepes[8] = { -2, -1, +1, +2, +2, +1, -1, -2 };
```


```
const int oszloplepes[8] = { +1, +2, +2, +1, -1, -2, -2, -1 };
```

```
int sakktabla[N][N];
```

```
int szamlalo = 1;
```

```
// matrix kiirasa a kepernyore
```

```
void kiiras() {  
    for (int i=0; i<N; i++) {  
        for (int j=0; j<N; j++) {  
            printf("%3d", sakktabla[i][j]);  
        }  
        printf("\n");  
    }  
    printf("\n");  
}
```

		oszloplépés				
		-2	-1	0	+1	+2
sorlépés	-2		-2,-1		-2,+1	
	-1	-1,-2				-1,+2
	0					
	+1	+1,-2				+1,+2
	+2		+2,-1		+2,+1	

```
// az adott sor oszlop a sakktabblan belül van-e es
// szabad-e a sakktabla adott helye?
// 0 = nem, 1 = igen
int szabadHely(int sor, int oszlop) {
    // az adott hely a sakktablan kívülre esik-e?
    if (sor<0 || sor>=N || oszlop<0 || oszlop>=N) {
        return 0;
    }
    // az adott helyen a sakktabla szabad-e
    if (sakktabla[sor][oszlop] == 0) {
        return 1; // igen, szabad
    } else {
        return 0; // nem, nem szabad
    }
}
```

```

// a huszarral az adott mezore lepunk, majd onnan tovabblepunk
// ugy, hogy ismet meghivjuk a lepes fuggvenyt
void lepes(int sor, int oszlop) {
    // odalepunk az adott poziciora
    sakktabla[sor][oszlop] = szamlalo;
    szamlalo++;
    // megprobalunk tovabblepni
    int hely = 0;
    do {
        // megnezzuk, hol van szabad hely...
        while (hely<8 && szabadHely(sor+szorlepes[hely], oszlop+oszlolepes[hely])==0) {
            hely++;
        }
        // ha van hely, ahova lephetunk, akkor tovabblepunk...
        if (hely<8) {
            lepes(sor+szorlepes[hely], oszlop+oszlolepes[hely]);
            // ha nem jutottunk megoldashoz
            if (szamlalo<=N*N) {
                hely++;
            }
        }
    } while (szamlalo<=N*N && hely<8);
    // ha nem jutottunk megoldashoz, akkor visszalepunk
    if (szamlalo<=N*N) {
        sakktabla[sor][oszlop] = 0;
        szamlalo--;
    }
}

```

```

main() {
    // matrix nullazasa
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            sakktabla[i][j] = 0;
        }
    }
    // megtesszük az első lepest
    lepes(N-1,0);
    // matrix kiirasa
    kiiras();
}

```

Feladat (Sudoku): Készítsünk programot, amely megtalálja egy előre megadott Sudoku rejtvény megoldását!

Milyen számok írhatók be az egyes mezőkbe?

Olyan számok, amelyek még nem szerepelnek sem az adott sorban, sem az adott oszlopba, sem az adott helyhez tartozó 3x3-as részben.

Megoldás keresésének menete:

- ▶ Kezdjük el sorfolytonosan beírni a lehetséges számokat.
- ▶ Ha valamelyik mezőbe már nem tudunk beírni semmilyen számot, akkor lépünk vissza egy mezőt és oda próbáljunk meg beírni másik számot (ha ott sincs már több lehetőség, akkor onnan is visszalépünk).

5	3	?		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
#include <stdio.h>
```

```
const int N = 9;
```

```
int sudoku[N][N] =
```

```
{ { 5, 3, 0, 0, 7, 0, 0, 0, 0 },  
  { 6, 0, 0, 1, 9, 5, 0, 0, 0 },  
  { 0, 9, 8, 0, 0, 0, 0, 6, 0 },  
  { 8, 0, 0, 0, 6, 0, 0, 0, 3 },  
  { 4, 0, 0, 8, 0, 3, 0, 0, 1 },  
  { 7, 0, 0, 0, 2, 0, 0, 0, 6 },  
  { 0, 6, 0, 0, 0, 0, 2, 8, 0 },  
  { 0, 0, 0, 4, 1, 9, 0, 0, 5 },  
  { 0, 0, 0, 0, 8, 0, 0, 7, 9 } };
```

```
// matrix kiirasa
```

```
void kiiras() {
```

```
    for (int i=0; i<N; i++) {  
        for (int j=0; j<N; j++) {  
            printf("%d ", sudoku[i][j]);  
        }  
        printf("\n");  
    }
```

```
    printf("\n");
```

```
}
```

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```

// beirhato a [sor][oszlop] pozicioba a megadott szam?
// (ugyanabban a sorban, oszlopban, 3x3 negyzetben
// nincs meg egy ugyanilyen szam?)
// return: 1 (true) = beirhato a szam
//          0 (false) = nem irhato be a szam
int beirhato(int sor, int oszlop, int szam) {
    // sor, oszlop ellenorzes
    for (int i=0; i<N; i++) {
        if (sudoku[sor][i]==szam || sudoku[i][oszlop]==szam) {
            return 0;
        }
    }
    // 3x3 negyzet leellenorzes
    int startSor = (sor / 3) * 3;
    int startOszlop = (oszlop / 3) * 3;
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            if (sudoku[startSor+i][startOszlop+j]==szam) {
                return 0;
            }
        }
    }
    return 1;
}

```

Diagram illustrating the 9x9 Sudoku grid with row and column indices. The grid is labeled with 'sor' (row) and 'oszlop' (column) indices. The grid contains numbers 1-9, with some cells highlighted in yellow and one cell containing a question mark.

5	3	?		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Diagram illustrating the 3x3 subgrid (3x3 negyzet) extracted from the 9x9 grid. The subgrid is labeled with 'startSor' (starting row) and 'startOszlop' (starting column) indices. The subgrid contains numbers 1-9, with some cells highlighted in yellow and one cell containing a question mark.

5	3	?
6		
	9	8


```

// return: 1 (true) = talaltunk megoldast
//          0 (false) = nem talaltunk megoldast
int megold() {
    for (int sor = 0; sor < N; sor++) {
        for (int oszlop = 0; oszlop < N; oszlop++) {
            if (sudoku[sor][oszlop] == 0) {
                for (int szam = 1; szam <= 9; szam++) {
                    if (beirhato(sor, oszlop, szam)) {
                        sudoku[sor][oszlop] = szam;
                        if (megold()) return 1;
                        sudoku[sor][oszlop] = 0;
                    }
                }
                return 0; // ha egyik szam sem jo
            }
        }
    }
    return 1; // nincs több ures mezo
}

```

```

int main() {
    if (megold()) {
        kiiras();
    } else {
        printf("Nincs megoldas!");
    }
}

```


Köszönöm a figyelmet!