Kodolányi János Egyetem

SUDOKU MEGOLDÁSI MÓDSZEREK

Megoldási módszerek programozása különböző fejlesztői platformokon.

Konzulens:

Dr. Pitlik László

Késztette:

Kovács Bálint, Kovács János, Mayerhöffer Orsolya, Csötönyi Viktória, Ott Alex, Bodicsi Viktor, Oláh Adrián

Üzemmérnök-informatikus alapképzési szak

LIN30 - Programozási alapelvek és módszertanok

Budapest

2024.

Kodolányi János Egyetem

SUDOKU SOLVING METHODS

Programming solution methods on various development platforms.

Supervisor:

Dr. László Pitlik

Created by:

Bálint Kovács, János Kovács, Orsolya Mayerhöffer, Viktória Csötönyi, Ott Alex, Viktor Bodicsi, Oláh Adrián

Computer Science Operational Engineering Bachelor’s Degree

LIN30 - Programming principles and methodologies

Budapest

2024.

**Tartalom**

[1. Előszó 4](#_Toc165192211)

[2. Preface 5](#_Toc165192212)

[1. Az alkalmazott módszerek bemutatása 6](#_Toc165192213)

[I. Megoldás *go* programnyelven írt algoritmus segítségével (Kovács János - IUO13K) 6](#_Toc165192214)

[A futtatókörnyezet paraméterei és alkalmazott szoftverek 6](#_Toc165192215)

[A módszer és előzmények leírása 6](#_Toc165192216)

[GO kód elkészítése 6](#_Toc165192217)

[A program felépítése 8](#_Toc165192218)

[A kód tesztelése 10](#_Toc165192219)

[II. Megoldás VBA-ban írt algoritmus segítségével (Kovács Bálint – OE64UF) 12](#_Toc165192220)

[A futtatókörnyezet paraméterei és alkalmazott szoftverek 12](#_Toc165192221)

[A megfelelő módszer keresése 12](#_Toc165192222)

[A VBA-kód elkészítése 13](#_Toc165192223)

[A VBA-kód felépítése és működése 15](#_Toc165192224)

[A kód tesztelése 16](#_Toc165192225)

[Az Excel tábla kialakítása és a felhasználói élmény optimalizálása 16](#_Toc165192226)

[III. Megoldás MS Excel Solver beépülő modul segítségével (Mayerhöffer Orsolya – IGJ96Q) 21](#_Toc165192227)

[Bevezetés 21](#_Toc165192228)

[1. Fejlesztési környezet 21](#_Toc165192229)

[1.1 Hardver 21](#_Toc165192230)

[1.2 Szoftver 21](#_Toc165192231)

[2. Első 4x4 megoldási módszer 21](#_Toc165192232)

[2.1 Google keresés 21](#_Toc165192233)

[2.2 ChatGPT 21](#_Toc165192234)

[2.3 Értekezletről készült felvétel 23](#_Toc165192235)

[2.4 Brainstorming osztálytársakkal 24](#_Toc165192236)

[3. Második megoldási módszer 25](#_Toc165192237)

[3.1 MIAU 25](#_Toc165192238)

[4. Harmadik megoldási módszer 27](#_Toc165192239)

[5. 9x9 megoldási módszer 29](#_Toc165192240)

[5.1 ChatGPT 30](#_Toc165192241)

[6. Konklúzió 30](#_Toc165192242)

[IV. Sudoku Excel Solver megoldás 4x4 és 6x6 táblázat alapján (Csötönyi Viktória - J908Q3) 31](#_Toc165192243)

[1. Bevezetés 31](#_Toc165192244)

[2. Célkitűzés 31](#_Toc165192245)

[3. Excel Solver 32](#_Toc165192246)

[3.1 Excel solver 4x4 táblázat 32](#_Toc165192247)

[3.2 Sudoku Solver 6x6 táblázat 33](#_Toc165192248)

[3.3. Eredmény 6x6 sudokunál 33](#_Toc165192249)

[4. Összegzés 35](#_Toc165192250)

[V. Generálás MS Excel Solver beépülő modul segítségével (Ott Alex András – IZFQB5) 36](#_Toc165192251)

[1. Bevezetés 36](#_Toc165192252)

[2. Feladat-értelmezés 38](#_Toc165192253)

[3. Első próba 39](#_Toc165192254)

[4. Második próba 40](#_Toc165192255)

[5. Harmadik próba 41](#_Toc165192256)

[6. Negyedik próba 41](#_Toc165192257)

[7. Ötödik próba 42](#_Toc165192258)

[8. Sudoku megoldása az Excel Solver segítségével: Konklúzió 43](#_Toc165192259)

[9. Hatodik próba 44](#_Toc165192260)

[10. Hetedik próba 44](#_Toc165192261)

[11. Nyolcadik Próba 46](#_Toc165192262)

[VI. Sudoku megoldás Python programnyelven (Bodicsi Viktor József – EJIOIM) 47](#_Toc165192263)

[1. Bevezetés: 47](#_Toc165192264)

[2. Program felépítése: 48](#_Toc165192265)

[3. A program működés közben: 54](#_Toc165192266)

[4. Felhasznált források: 57](#_Toc165192267)

[VII. Sudoku megoldó program Python nyelven, PyCharm környezetben Oláh Adrián – CW6W8K 58](#_Toc165192268)

[1. A Sudoku szabályai, felépítése: 58](#_Toc165192269)

[2. A Program felépítése (GUI, Sudoku megoldás, hibakeresés) 58](#_Toc165192270)

[3. Fő program (Definíciók, Live Monitoring, Érzékelők,Logikai változók) 62](#_Toc165192271)

[4.Eredmények megjelnítése, tkinter(felugró ablakok), szám bevitel billentyűparanccsal,Live monitoring GUI, új tábla igénylése üres mezőkkel 63](#_Toc165192272)

[5. Felhasznált források 65](#_Toc165192273)

[2. Az alkalmazott módszerek összehasonlítása OAM elemzés segítségével 66](#_Toc165192274)

[Objektumok: 66](#_Toc165192275)

[Attribútumok: 66](#_Toc165192276)

[3. Tantárgyi kapcsolatok 67](#_Toc165192277)

[LIN029 - Operációs rendszerek 67](#_Toc165192278)

[LIN026 - Matematikai alapok 67](#_Toc165192279)

[LSK001 - Kultúra, sport, munkahelyi jóllét 1. 67](#_Toc165192280)

[LKN490 - Európai civilizáció és identitás 67](#_Toc165192281)

[LIN028 - Hálózatok és számítógép architektúrák 67](#_Toc165192282)

[LIN027 - Adatszerkezetek és algoritmusok 67](#_Toc165192283)

[LNE178 - A jog szerepe a modern társadalmakban 68](#_Toc165192284)

[4. Összegzés 69](#_Toc165192285)

[5. Summary 69](#_Toc165192286)

[6. Rövidítésjegyzék, hivatkozásjegyzék, irodalomjegyzék 70](#_Toc165192287)

[6.1 Csötönyi Viktória\_J908Q3: 70](#_Toc165192288)

[7. Mellékletek 71](#_Toc165192289)

# Előszó

**A Sudokuról röviden:** A hagyományos sudoku egy logikai rejtvény, amely egy 9×9-es rácsból áll, tovább osztva 3×3-as négyzetre. A cél az, hogy minden sorban, oszlopban és 3×3-as négyzetben az 1-től 9-ig terjedő számok mindegyike egyszer szerepeljen, ügyelve arra, hogy ne ismétlődjenek. A játék kezdetén néhány szám előre ki van töltve, ezek segítenek a többi szám helyének megtalálásában. A sudoku megoldása logikai gondolkodást és türelmet igényel, mivel nem szükséges matematikai tudás hozzá. Ez a játék kiváló agyi edzést nyújt, fejleszti a problémamegoldó készséget és a figyelmet.

<https://hu.wikipedia.org/wiki/Sz%C3%BAdoku>

A dokumentum további részében a különböző Sudoku megoldási módszerek megszületéséhez vezető utakat járjuk körbe, majd elemezzük és egymással összehasonlítjuk az elkészült megoldórendszerek sikerességi rátáját és teljesítményét.

A részletes összehasonlítás egy külön OAM Excel-táblában kapott helyet.

# Preface

**A brief overview of Sudoku:** Traditional Sudoku is a logic-based puzzle that consists of a 9×9 grid, further divided into 3×3 squares. The objective is to fill every row, column, and 3×3 square with the numbers 1 through 9, ensuring that each number appears exactly once, without any repetition. At the start of the game, some numbers are already filled in, which helps in determining the placement of the remaining numbers. Solving sudoku requires logical thinking and patience, as no mathematical knowledge is necessary. This game provides excellent mental exercise, enhancing problem-solving skills and attention.

<https://hu.wikipedia.org/wiki/Sz%C3%BAdoku>

The remainder of the document explores the various methods of solving Sudoku, tracing the development of these techniques, then analyzing and comparing the success rates and performance of the completed solving systems.

The detailed comparison is placed in a separate OAM Excel sheet.

# Az alkalmazott módszerek bemutatása

## Megoldás *go* programnyelven írt algoritmus segítségével (Kovács János - IUO13K)

(v20240417\_2248)

### A futtatókörnyezet paraméterei és alkalmazott szoftverek

* Macbook pro M1 ARM, 16GB
* OS: Mac OS 14.4
* Office: Google sheet és MS Excel 16.83
* Github
* VS Code 1.87.2, aktuális go pluginokkal

### A módszer és előzmények leírása

Régóta szemeztem a go nyelvvel és most úgy éreztem ez jó alkalom lesz megismerni a szintaktikáját.

Az backtrack rekurzív algoritmus a korábbi excel solveres munkálataim során már megismertem, így azt használva fejlesztettem a megoldásomat. Volt egyfajta evolúció, ezek a git-ben megtalálhatóak.

GPT 4.0-át főleg a még nem ismert go nyelvi elemek keresése közben és a dokumentálás közben használtam.

### GO kód elkészítése

A kód az alábbi helyen található:

<https://github.com/kovacs213janos/sudokugo>







### A program felépítése

A program a klasszikus backtracking algoritmust implementálja Go nyelven. Az algoritmus próbálkozik minden lehetséges szám elhelyezésével a tábla minden üres cellájába, miközben biztosítja, hogy a megoldás érvényes maradjon. A program felhasználóbarát, kéri a felhasználó bevitelét, és visszajelzést ad a folyamat során, így nem csak egy hatékony Sudoku solver, hanem interaktív is. Az algoritmus iteratív megközelítése és a memória használat kiírása segít megérteni az algoritmus futtatásának komplexitását és erőforrás-igényét. A program hasznos lehet a backtracking algoritmusok tanulmányozására és az algoritmikus gondolkodásmód fejlesztésére. A kód strukturált és jól kommentezett, ami elősegíti a megértését és a további fejlesztés lehetőségét.

A program a következő elemekből áll:

#### Állandók és Globális Változók:

* Size: A Sudoku táblázat méretét definiáló állandó, ebben az esetben 9.
* Iteraciok: Globális változó, amely nyomon követi a rekurziós hívások számát a Sudoku tábla megoldása során.

#### Sudoku Táblázat Struktúra:

* type Sudoku [Size][Size]int: Egy 9x9-es tömböt definiál, amely egy Sudoku táblát képvisel.

#### Validációs Függvény (Validalas):

* Ellenőrzi, hogy egy adott számot elhelyezhetünk-e biztonságosan egy adott sorban és oszlopban, valamint a megfelelő 3x3-as részrácson belül.

#### Solver Függvény (Solver):

* Megpróbálja megoldani a Sudoku táblát a backtracking módszerrel.
* Végigiterál a tábla összes celláján, és ha üres cellát talál (értéke 0), megpróbálja beírni a lehető összes számot 1-től 9-ig.
* Ha egy szám helyesnek bizonyul, rekurzívan hívja önmagát a következő üres cellára lépve.
* Sikertelen próbálkozás esetén visszaállítja a cellát üresre (backtracking).
* Ha az összes cella ki van töltve, a tábla megoldottnak tekinthető.

#### Kiíratás Függvény (kiiratasTabla):

* Kiírja a Sudoku tábla aktuális állapotát.

#### Szám Bekérő Függvény (szambekeres):

* Bekéri a felhasználótól azt a számot, amely megadja, hány cellát kell üresen hagyni a táblán.

#### Random Törlés Függvény (randomRemove):

* Véletlenszerűen eltávolítja a megadott számú cellát a táblából, hogy feladványt hozzon létre.

#### Main Függvény (main):

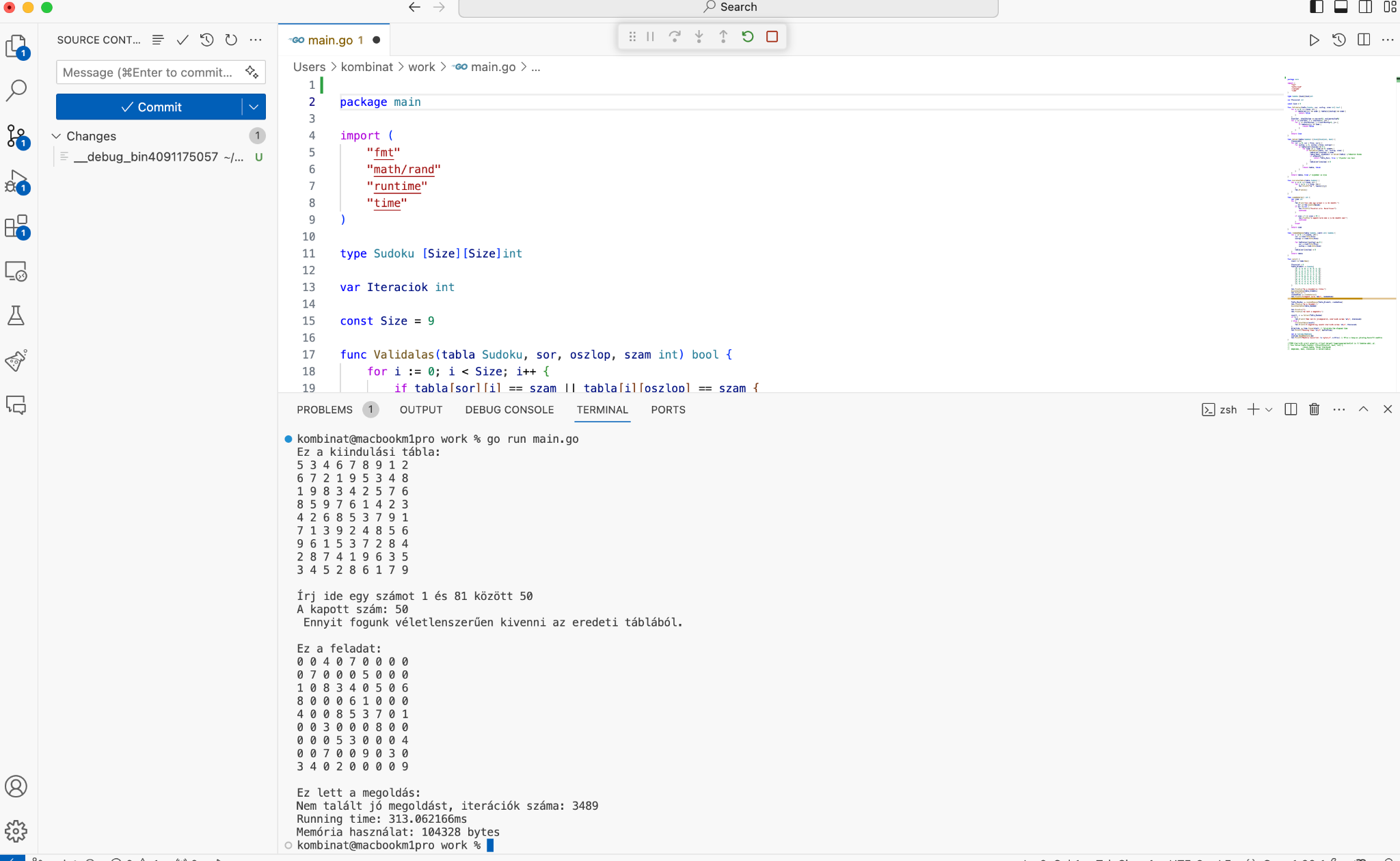
* A program belépési pontja.
* Inicializálja a táblát egy előre megadott megoldással.
* Meghívja a szambekeres függvényt a felhasználói interakcióhoz.
* Létrehoz egy feladványt a randomRemove függvény segítségével.
* Megoldja a feladványt a Solver függvény hívásával.
* Kiírja az eredményt és statisztikai információkat: a megoldáshoz szükséges iterációk számát és a program futásának idejét.
* Lekérdezi a memória használatot és kiírja azt.

#### Időmérés és Memóriahasználat:

* A main függvény méri a futási időt és lekérdezi a memóriahasználatot a runtime csomag segítségével.

### A kód tesztelése

Ez fejlesztés közben és később manuálisan történt, a közben talált hibák azonnal javításra kerültek.



## Megoldás VBA-ban írt algoritmus segítségével (Kovács Bálint – OE64UF)

(v20240417\_2248)

### A futtatókörnyezet paraméterei és alkalmazott szoftverek

* CPU: Intel Core i5-11600K @ 3,90 GHz
* RAM: 16 GB DDR4
* Háttértár: Samsung 970 EVO Plus 500GB NVMe
* OS: Windows 11 Pro, build: 22631.3296
* Office: Microsoft Office Professional Plus 2021

### A megfelelő módszer keresése

A feladat egy sudoku-megoldó rendszer elkészítése volt, Excel Solver segítségével. Mivel az Excel Solvert nem sikerült megfelelően paraméterezni a 9×9-es sudoku feladványok hatékony megoldásához, a ChatGPT 4 (továbbiakban: **GPT**) az alábbi megoldási módszereket javasolta lehetséges alternatívaként:

#### Brute force algoritmus

A brute force vagy teljes keresési algoritmus egyszerűen végigpróbálja az összes lehetséges kombinációt a rács kitöltésére, anélkül, hogy logikát vagy optimalizálási stratégiát alkalmazna. Ez az algoritmus rendkívül erőforrás-igényes.

A Sudoku valós brute force megoldása, amely egy 9×9-es grid minden cellájára és minden lehetséges számára kiterjed, a gyakorlatban nem megvalósítható a rendkívül nagy számú kombináció miatt.

A legrosszabb esetben a következő számú kombinációt kellene kipróbálni:

A Sudoku minden cellájába 1-től 9-ig terjedő számokat helyezhetünk, tehát 9 lehetőség van minden cellára. Mivel 81 cella van (9 sor × 9 oszlop), így a teljes kombinációk száma: 981, pontos számmal kifejezve:

*196627050475552913618075908526912116283103450944214766927315415537966391196809*

A tiszta brute force megközelítés a Sudoku megoldására nem praktikus. A backtracking algoritmus sokkal hatékonyabb, mivel csak az érvényes kombinációkat próbálja ki, figyelembe véve a Sudoku szabályait, ezáltal jelentősen csökkenti a szükséges próbálkozások számát.

[*https://en.wikipedia.org/wiki/Brute-force\_search*](https://en.wikipedia.org/wiki/Brute-force_search)

#### CSP algoritmus

A Constraint Satisfaction Problem (CSP) algoritmusok, mint a backtracking algoritmus kiterjesztése, olyan technikákat alkalmaznak, mint a változók értékkészletének szűkítése (domain reduction), az előretekintés (forward checking), és/vagy a konzisztencia algoritmusok (mint az AC-3 algoritmus). Ezek az algoritmusok a probléma korlátjainak kihasználásával próbálják csökkenteni a keresési teret és hatékonyabban találnak megoldást, ezért kiválóan alkalmasak a Sudokuhoz hasonló logikai rejtvények megoldására.

A (CSP) algoritmus teljes körű implementálása egy 9×9-es Sudoku esetében a VBA környezetben nagyon összetett feladat, ami magában foglalja az értékkészletek kezelését, korlátok dinamikus alkalmazását és a keresési stratégiák (pl. backtracking) implementálását.

[*https://en.wikipedia.org/wiki/Constraint\_satisfaction\_problem*](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem)

*A CSP módszer a szerző által szubjektíven megítélt szépsége miatt ugyan jelen írásban említésre került, azonban összetettsége okán kidolgozása nem valósult meg. A CSP módszertől lényegesen egyszerűbben implementálható VBA alatt egy klasszikus backtracking algoritmus, amely esetében a kódolási idő / kód teljesítmény arány az aktuális feladat megoldásához optimálisabb.*

#### Backtracking algoritmus

Ez az algoritmus egy klasszikus rekurzív megközelítés, amely próbálkozik minden lehetséges értékkel a rács üres celláiban, és visszalép (backtrack), ha az aktuális kitöltés nem vezet megoldáshoz. A backtracking algoritmus nagyon hatékony a Sudoku-féle problémák megoldására, mivel képes végigiterálni az összes lehetséges konfigurációt anélkül, hogy felesleges kombinációkat vizsgálna.

[*https://en.wikipedia.org/wiki/Backtracking*](https://en.wikipedia.org/wiki/Backtracking)

***A továbbiakban a backtracking algoritmussal foglalkozunk.***

### A VBA-kód elkészítése

A VBA-kód a GPT segítségével került megírásra.

A következő prompt megadása után, a GPT legenerálja a szükséges kód alapját:

„Tudnál írni egy Backtracking Algoritmust a sudoku 9x9-hez Excel VBA-ban? Az adatokat a C12:K20-ból vegye, a megoldási grid a Q12:Y20. A VBA kódban a változók suffixe tartalmazza a változó típust rövidítve, a változó nevétől egy "\_"-al elválsztva (pl.: \_str, \_int, \_bool). A functionok prefixe legyen "fun".”

A generált kód felülvizsgálata során csak minimális módosításokra van szükség, a kód alapvetően működőképes.

A kód egy újonnan létrehozott modulba (.bas) kerül beillesztésre.

**A kód:**

Function funIsSafe(grid As Variant, row As Integer, col As Integer, num As Integer) As Boolean

Dim idx\_int As Integer

Dim jdx\_int As Integer

Dim boxStartRow\_int As Integer

Dim boxStartCol\_int As Integer

'Sor ellenőrzés

For idx\_int = 1 To 9

If grid(row, idx\_int) = num Then

funIsSafe = False

Exit Function

End If

Next idx\_int

'Oszlop ellenőrzés

For idx\_int = 1 To 9

If grid(idx\_int, col) = num Then

funIsSafe = False

Exit Function

End If

Next idx\_int

'3x3 grid ellenőrzése

boxStartRow\_int = row - ((row - 1) Mod 3)

boxStartCol\_int = col - ((col - 1) Mod 3)

For idx\_int = 0 To 2

For jdx\_int = 0 To 2

If grid(boxStartRow\_int + idx\_int, boxStartCol\_int + jdx\_int) = num Then

funIsSafe = False

Exit Function

End If

Next jdx\_int

Next idx\_int

funIsSafe = True

End Function

Sub SolveSudoku()

Dim grid As Variant

Dim i As Integer, j As Integer

Dim ws As Worksheet

Set ws = ThisWorkbook.Sheets("9x9 VBA")

ReDim grid(1 To 9, 1 To 9) As Integer

' Beolvasás tömbbe Q12:Y20

For i = 1 To 9

For j = 1 To 9

grid(i, j) = ws.Cells(11 + i, 2 + j).Value

Next j

Next i

' Megoldás

If funSolveSudoku(grid) Then

For i = 1 To 9

For j = 1 To 9

ws.Cells(11 + i, 16 + j).Value = grid(i, j)

Next j

Next i

MsgBox "A feladvány megoldása sikeres!", vbInformation

Else

MsgBox "A feladványt nem sikerült megoldani.", vbCritical

End If

End Sub

Function funSolveSudoku(grid As Variant) As Boolean

Dim row\_int As Integer

Dim col\_int As Integer

Dim num\_int As Integer

Dim isEmpty\_bool As Boolean

isEmpty\_bool = True

'Első üres cella keresése

For row\_int = 1 To 9

For col\_int = 1 To 9

If grid(row\_int, col\_int) = 0 Then

isEmpty\_bool = False

GoTo NextStep

End If

Next col\_int

Next row\_int

NextStep:

If isEmpty\_bool Then

funSolveSudoku = True

Exit Function

End If

'Összes variáció kipróbálása

For num\_int = 1 To 9

If funIsSafe(grid, row\_int, col\_int, num\_int) Then

grid(row\_int, col\_int) = num\_int

If funSolveSudoku(grid) Then

funSolveSudoku = True

Exit Function

End If

grid(row\_int, col\_int) = 0 ' Visszaállítás

End If

Next num\_int

funSolveSudoku = False

End Function

### A VBA-kód felépítése és működése

A kód két fő részből áll: a **funIsSafe** functionből, ami ellenőrzi, hogy egy adott szám érvényes-e egy adott pozícióban, és a **SolveSudoku** subból, ami maga a megoldó algoritmus. Lépésről lépésre haladva a kód működése a következő:

#### „funIsSafe” funcion

Sor ellenőrzése: A függvény végigiterál az adott sor összes elemén (*row*, fix oszlop változik) és ellenőrzi, hogy a *num* szám szerepel-e már ebben a sorban. Ha igen, *False* értékkel tér vissza, jelezve, hogy az adott szám nem helyezhető el biztonságosan.

Oszlop ellenőrzése: Hasonlóan az előzőhöz, itt az adott oszlop (col, fix sor változik) minden elemét ellenőrzi, hogy tartalmazza-e a *num* számot. Ha igen, a függvény *False*-sal tér vissza.

3x3 grid ellenőrzése: Kiszámítja a kis 3×3 doboz bal felső sarok cellájának (boxStartRow\_int és boxStartCol\_int) koordinátáit, amelybe a vizsgált cella esik. Ezután ellenőrzi, hogy a *num* szám szerepel-e ebben a 3×3 dobozban. Ha igen, *False* értékkel tér vissza.

Ha egyik ellenőrzés során sem talált problémát, akkor *True* értékkel tér vissza, jelezve, hogy a szám biztonságosan elhelyezhető az adott pozícióban.

#### „SolveSudoku” sub

Munkalap beállítása és rács inicializálása: Kiválasztja az aktuális munkalapot ("9x9 VBA") és létrehoz egy 9×9-es tömböt a Sudoku tábla tárolására.

Tábla beolvasása: Beolvassa a Sudoku táblát a munkalapról a grid tömbbe. A feladvány tábla a C12:K20 tartományban található a munkalapon.

Megoldás keresése: Meghívja a funSolveSudoku függvényt, amely rekurzívan próbálja meg kitölteni a táblát.

Eredmény kiírása: Ha a funSolveSudoku igazzal tér vissza (sikerült megoldani a feladványt), akkor a megoldást kiírja a munkalap Q12:Y20 tartományába, és információs üzenet jelenik meg. Ha hamissal tér vissza (nem sikerült megoldani), egy kritikus hibaüzenet jelenik meg.

#### „funSolveSudoku” function

Első üres cella keresése: Végigiterál a táblán, és keres egy üres cellát (aminek az értéke 0).

Ha nincs üres cella: Ha minden cella kitöltött (nincs több üres hely), akkor sikeresen megoldotta a Sudoku-t, True-val tér vissza.

Szám próbálgatása: Próbálja beilleszteni az 1-től 9-ig terjedő minden számot az üres cellába, amit az előző lépésben talált. Minden egyes szám esetén ellenőrzi a *funIsSafe* függvénnyel, hogy az adott szám érvényes-e.

Rekurzió: Ha egy számot sikeresen be lehet helyezni (azaz a *funIsSafe* igazat ad vissza), akkor rekurzívan meghívja önmagát a frissített táblával, hogy próbálja megoldani a következő cellát.

Backtracking: Ha egy adott szám beillesztése után nem sikerül megoldani a feladványt (a rekurzív hívás hamissal tér vissza), akkor visszaállítja az üres cella értékét 0-ra (visszavonja az előző lépést) és kipróbál egy másik számot.

A megoldás keresése addig folytatódik, amíg vagy meg nem találja a teljes megoldást, vagy bebizonyosodik, hogy a tábla nem oldható meg a jelenlegi állapotából kiindulva.

### A kód tesztelése

A kód tesztelése az Interneten, Google kereső segítségével fellelhető Sudoku feladványokkal történt.

A feladványok teszteléséhez a C12:K20 tartományban szükséges megadni a rendelkezésre álló számokat, majd megnyomni a „MEGOLDÁS!”- gombot.

A tesztelés során nem sikerült olyan feladványt találni, amelyet ne tudott volna megoldani az algoritmus.

Átlagos futásidő: 1-3 perc.

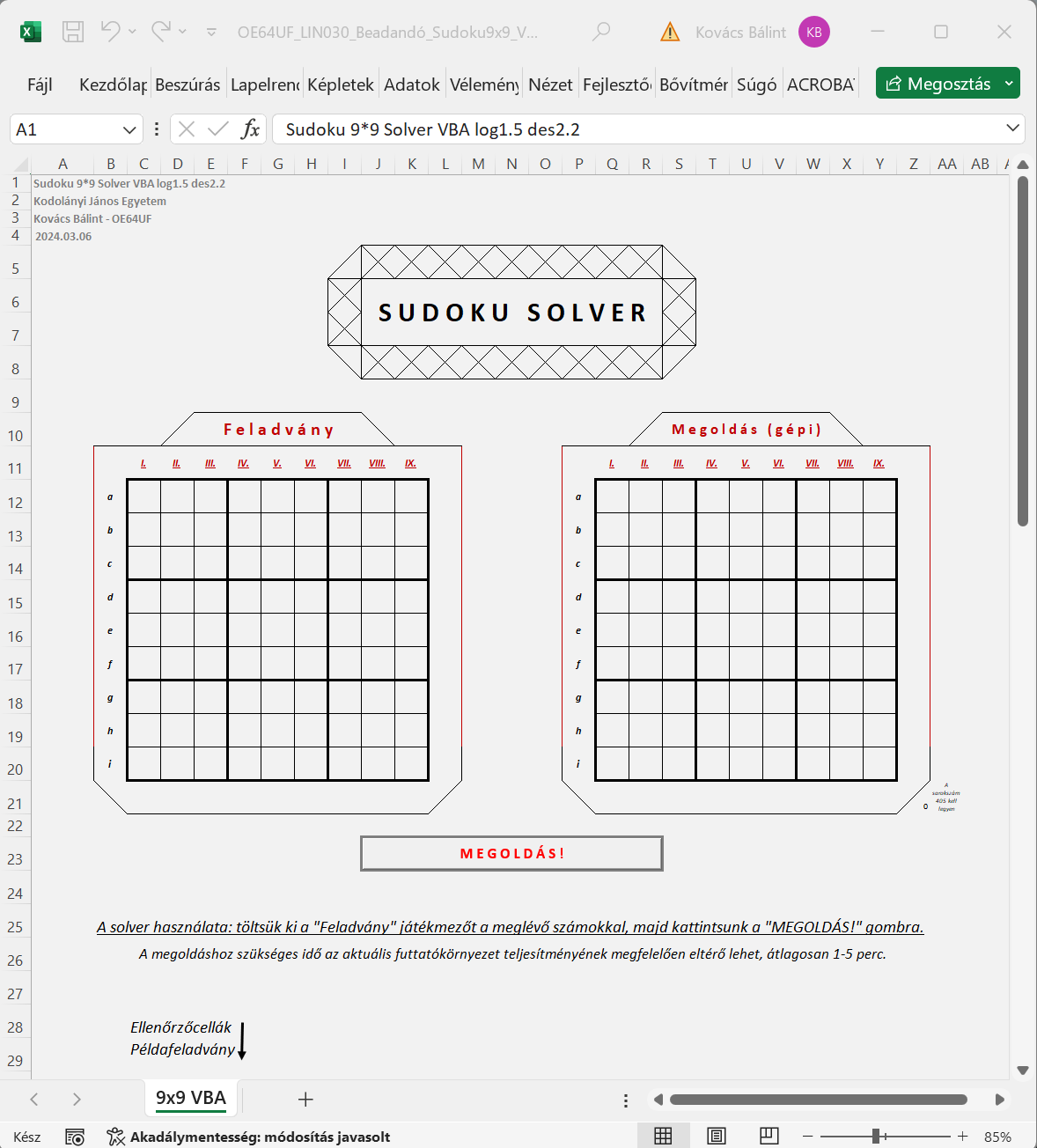
[*https://www.google.com/search?q=sudoku+with+answers*](https://www.google.com/search?q=sudoku+with+answers)

### Az Excel tábla kialakítása és a felhasználói élmény optimalizálása

Az elkészült algoritmus megfelelő minőségű keretbe (GUI-ba) foglalása sok esetben legalább olyan fontos (ha nem fontosabb), mint magának az algoritmusnak a kidolgozása.

A külalak az alábbi szempontok figyelembevételével került kialakításra:

* Felhasználóközpontúság: a tervezés során mindig a felhasználó igényeit és visszajelzéseit kell szem előtt tartani. A táblázatnak könnyen navigálhatónak és érthetőnek kell lennie minden típusú felhasználó számára.
* Konzisztencia: a felületelemek, mint a gombok, feliratok, keretek stílusa legyen konzisztens az egész táblázaton belül, hogy a felhasználók gyorsan megtanulhassák és hatékonyan használhassák azt.
* Egyszerűség: kerülni kell a felesleges bonyolultságot. A táblázat megjelenített részének csak a szükséges elemeket kell tartalmaznia, hogy a felhasználó könnyen elérje a kívánt funkciókat anélkül, hogy zavarba jönne.
* Adaptivitás: A táblázatnak alkalmazkodnia kell különböző képernyőméretekhez és felbontásokhoz, hogy a különböző eszközökön is megfelelően jelenjen meg.
* Színek és betűtípusok: A színek és betűtípusok megfelelő választása javíthatja a felhasználói élményt, segíthet a fontos elemek kiemelésében és javíthatja az olvashatóságot.



*Képernyőkép az elkészült felhasználói felületről*

#### Az Excel ablakának méretezése

A táblázat megnyitásakor célszerű a sallangoktól, segédcelláktól mentes nézettel indítani, megkönnyítve a felhasználónak a megfelelő területre való fókuszálást.

A méretezést dinamikusan is elvégezhetjük az aktuális képernyőfelbontás függvényében, azonban jelen esetben egy statikus, 600×660 pixeles méretre kényszerítjük az Excel ablakát. A felbontás megválasztása során evidenciának vesszük, hogy a programot nem fogják 800×600-as felbontástól kisebb értéken futtatni.

A méretezéshez szükséges VBA-kód:

Dim OpenWidth\_int As Integer

Dim OpenHeight\_int As Integer

Private Sub Workbook\_Open()

OpenWidth\_int = Application.Width

OpenHeight\_int = Application.Height

With Application

.WindowState = xlNormal

.Width = 600

.Height = 660

End With

End Sub

Az Excel bezárását követően vissza kell állítani az eredeti ablakméretet, különben az átméretezés a később megnyitandó Excel táblázatokra is kihat:

Private Sub Workbook\_BeforeClose(Cancel As Boolean)

With Application

.Width = OpenWidth\_int

.Height = OpenHeight\_int

End With

End Sub

#### A menüszalag bezárása

Esetünkben a menüszalag bezárásra kerül a táblázat megnyitásakor, ugyanis a felhasználónak nem kell használni azt a Sudoku Solver futtatása közben. Kilépéskor a menüszalagot újra meg kell nyitni, különben a menüszalag bezárása a később megnyitandó Excel táblázatokra is kihat:

Dim IsRibbonShowedBefore\_bool As Boolean

Private Sub Workbook\_Open()

If Application.CommandBars("Ribbon").Height >= 150 Then

IsRibbonShowedBefore\_bool = True

Application.CommandBars.ExecuteMso ("MinimizeRibbon")

Else

IsRibbonShowedBefore\_bool = False

End If

End Sub

Private Sub Workbook\_BeforeClose(Cancel As Boolean)

If IsRibbonShowedBefore\_bool = True Then

Application.CommandBars.ExecuteMso ("MinimizeRibbon")

End If

End Sub

#### A fókusz megadása

A megnyíló munkafüzetben kiválasztásra kerül a Sudoku Solvert tartalmazó munkalap, a nagyítás 85%, az aktív cella az „A1”:

Private Sub Workbook\_Open()

Sheets("9x9 VBA").Activate

ActiveWindow.Zoom = 85

With ThisWorkbook.Sheets("9x9 VBA")

.Activate

.Range("A1").Select

End With

End Sub

#### Az ellenőrző- és segédcellák

A „9x9 VBA” munkalap A35:X68 tartományában találhatóak az ellenőrző és segédcellák, amelyek segítségével követhetjük a program helyes működését.

#### Az „A1”-cella tartalma

Az „A1”-cellában egy kezdetleges verziókövetés kapott helyet. a „logX.Y” jelölés az algoritmus és az adat ki- és bemenet működési logikáját jelzi, ahol az „X” az algoritmus változásával növekszik, az „Y” a ki- és beviteli mezők pozíciójának, működésének változását jelzi. A „desX.Y” értékének módosításával a felhasználói felület változását kell lekövetni.

## Megoldás MS Excel Solver beépülő modul segítségével (Mayerhöffer Orsolya – IGJ96Q)

# Bevezetés

A dokumentum célja megismertetni a MS Excel Solver beépülő modul segítségével elkészített 4x4 es sudoku játék megoldási módszerét valamint ugyanezen módszerrel bemutatni 9x9-es játékon való próbálkozást is.

A 4x4 sudoku játék elsősorban gyerekeknek ajánlják és azoknak akik még csak ismerkednek a játékkal. Persze akkor is kiváló választás lehet ha épp csak egész kis időnk van, de szeretnénk játszani egyet. A játék szabályai hasonlóak az eredetihez. A lényeg, hogy minden sorban, oszlopban és 2x2-es mezőben 1-4-ig csak egyszer szerepelhetnek a számok. Egyszer viszont szerepelniük kell.

# Fejlesztési környezet

## Hardver

Dell Latitude 5530

RAM: 16 GB

Storage: 250 GB SSD

## Szoftver

MS Windows 11 Enterprise

Microsoft® Excel® for Microsoft 365 MSO

(Version 2402 Build 16.0.17328.20124) 64-bit

# Első 4x4 megoldási módszer

## Google keresés

Keresnem kellett egy online felületet, ahol néhány játék után megértettem a játék célját és szabályait.

## ChatGPT

Nem volt ötletem, hogy hogyan kezdjem el, így az alábbi kérdéssel fordultam a ChatGPT-hez:

A screenshot of a computer

Description automatically generated

* + - 1. ábra

A 3. pont alapján indítottam egy Google keresést, hogy megtudjam, hogyan telepíthető a bővítmény, pár kattintással használatba is vettem.

Létrehoztam a 4x4-es táblázatot, viszont a fenti válasz ennél tovább nem volt segítségemre, ezért más kérdésekkel próbálkoztam, hogy megkapjam a pontos válaszokat, de nem sikerült.

## Értekezletről készült felvétel

Többször megnéztem az óráról készült felvételből azt a részt, ahol a Solver-ről volt szó, ezután fogalmaztam meg a játék szabályait – ekkor még csak írásban -, mint megadandó feltételek:

* minden sor összege 10
* minden oszlop összege 10
* minden rács összege 10
* minden szám egyszer szerepelhet soronként, oszloponként és rácsonként
* 1<= n <=4
* minden szám egész szám legyen

SUM képlettel elvégeztem a számolásokat és beírtam a fenti feltételeket a Solver-be (lásd: sudoku\_v1.0.xlsx), lefutattam és az alábbi üzenetet adta vissza:

A screenshot of a computer

Description automatically generated

* + - 1. ábra

Minden feltétel teljesült, kivéve, hogy egy szám csak egyszer szerepelhet minden sorban, oszlopban és rácsban, az előre beírt számokat nem vette figyelembe. Ennél a pontnál ismét elakadtam.

## Brainstorming osztálytársakkal

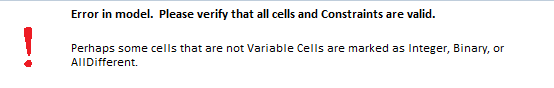
Hosszú keresés, youtube video és Solver működéséről szóló cikkek olvasása után felfedeztem, hogy a Solver-ben elérhető egy kimondottan olyan megkötés, mely biztosítja, hogy minden nézetben egyzsre forduljon elő egy szám a megadott tartományból:

A screenshot of a computer

Description automatically generated

3.ábra

Tehát beírtam egyenként minden sort, oszlopot és rácsot, mint „call reference” és „AllDifferent”-re állítottam be, valamint azt, hogy az összegük 10 kell, hogy legyen, a cél pedig a sorok, oszlopok, rácsok összege, azaz 120, így az alábbi hibaüzenetet kaptam.



4. ábra

Ekkor írtam osztálytársamnak, Ott Alexnek, aki mint kiderült, hasnoló megoldáson dolgozott 9x9-es táblán és így derült ki számomra, hogy a fenti megkötés csak akkor fut le, ha a a tábla üres, nincs előre beírt szám, így azokat kitörölve a Solver ki tudta tölteni megfelelően a táblát. (sudoku\_v1.1.xlsx)

Ahhoz, hogy csökkentsem a futási időt úgy, hogy a hiba lehetősége ne legyen több, a nézetek összegit külön mezőben számoltam, így csak ezen tartományok beírására volt szükséges a Solver-ben.

A screenshot of a graph

Description automatically generated

1. ábra (sudoku\_v1.2.xlsx)

# Második megoldási módszer

Szerettem volna egy olyan megoldást is, ami képes kiegészíteni a már meglévő számokat a szabályok szerint.

## MIAU

Ez volt a meghatározó segítség a végső megoldásomhoz, mivel már korábban elkezdtem azon gondolkozni, ha segédtáblázatokon megszámolom „COUNTIF” képlettel, hogy nézetenként hányszor szerepelnek 1 és 4 között a számok, akkor azt valahogy át lehet írni a megfelelő megkötéssé és így biztosan különbözni fognak minden nézetben a számok.

A screenshot of a computer

Description automatically generated

6. ábra

Nem akartam IF-eket használni, túl bonyolultnak találtam.

Tovább kerestem, így találtam meg egy, a [MIAU felületen](https://miau.my-x.hu/miau2009/index.php3) publikált megoldást, ami a szórás segítségével határozza meg a megkötéseket:



Így a változó mezőket egy külön táblázatba vezettem:

A green square with black text

Description automatically generated

7. ábra

A három nézetre készült segédtáblázatokra különböző módokon szórást számoltam (6. ábra), miszerint az átlagtól való eltérésük abban az esetben egyezik az átlaggal, ha mindenhol egyszer szerepel egy szám. Minden más esetben nem nulla. Tehát ezzel a logikával van megkövetelve, hogy a segédtáblázat (6. ábra) értékei mindenhol 1-es legyen 0 cél értékre minimalizálva, mivel az soha nem lesz teljesen 0, viszont egész számként várt eredmény esetében ez lényegtelen. Ezt pedig további néhány megkötés biztosítja az eddig használtak közül, mint pl. minden nézet összege 10 és kisebb vagy egyenlő, mint 4 és egyenlő vagy nagyobb, mint 1.

Bár meg volt a megoldás, mégis sokáig eltartott, mire azt helyesen be tudtam írni a Solver-be, végül a sikert a különböző nézetek szórás eredményeinek összeadása hozta meg, amit egy külön mezőben számoltam két féle módon is:

A screenshot of a calculator

Description automatically generated

8. ábra (sudoku\_v2.0.xlsx)

A screenshot of a computer

Description automatically generated

9. ábra (sudoku\_v2.0.xlsx)

Nehéséget okozott rájönnöm arra, hogy valójában a cél érték soha nem lesz pontosan nulla, így azt csak minimalizálni lehet a helyes megoldás eléréséhez.

# Harmadik megoldási módszer

Itt a célom az, hogy a Solver ne csak előre beírt számok, ahogy a 8. és 9. ábrán a piros színű számok mutatják, egy feladat esetén találjon megoldást, hanem úgy is, ha bármilyen megoldható feladat alapján adom meg számokat. Így bővítettem a változó cellák tartományát pont annyi cellára, mint amennyi a játék is, azaz 4x4-re és a játék celláit hivatkoztam a változó cellákra, majd a játék táblába beírtam számokat, úgy, hoyg az megoldható legyen, valamint a Solver megkötései közt módsítottam azt a tartományt a változó cellák szerint, ahova egész számot várok és így is sikerült megoldásra jutnom:

A green and white sheet of paper with green text

Description automatically generated

10. ábra (sudoku\_v2.1)

A hátrány az, hogy minden új játék előtt újra kell hivatkozni a játék mezőit a változó mezőkre, majd kézzel beírni a feladvány számait a játék táblázatába.

# 9x9 megoldási módszer

A negyedik pontban leírt módszer alapján 9x9-es táblára is elkészítettem a rendszert (11. ábra), viszont sikertelen megoldás született.

A screenshot of a computer

Description automatically generated

11. ábra

Ezután megpróbáltam további megkötésként megadni a változó mezőkbe várt számok összegét hozzáadva a nem haszált változó mezőkbe írt egyesek összegével, viszont így sem tudott megfelelő megoldás születni.

A screenshot of a computer

Description automatically generated

12. ábra

## ChatGPT

Kíváncsi voltam, hogy 9x9-es táblára van-e megoldása a ChatGPT-nek Excel Solver-rel:

A screenshot of a computer

Description automatically generated

# Konklúzió

A bemutatott módszerek által jól látható, hogy az Excel Solver beépülő modul nem a legjobb, csak kis térben, 4x4-es táblán működik biztosan, nehezebb, 9x9-es tábla feladatot már nem tud megoldani megfelelően.

## Sudoku Excel Solver megoldás 4x4 és 6x6 táblázat alapján (Csötönyi Viktória - J908Q3)

### Bevezetés

Sudoku népszerű logikai játék, amely a játékost gondolkodásra sarkall. A feladványokat megtalálhatjuk az újságok lapjain, a rejtvénykönyvekben vagy manapság különböző mobil applikációkban szintén.

Elsődlegesen segítséget kértem a ChatGpt-től, hogy magyarázza el a szabályokat (ChatGPT3.5).

Az alábbira kértem: ,,Kérlek magyarázd el a sudoku szabályait!”

Ezt a leírást kaptam:

,,A Sudoku egy logikai játék, amely egy 9x9-es rácsra épül, amelyet további 9 kisebb 3x3-as rácsra osztanak fel. A játék célja, hogy minden sorban, oszlopban és 3x3-as blokkban szerepeljenek a számok 1-től 9-ig úgy, hogy egyetlen szám sem ismétlődik meg ugyanabban a sorban, oszlopban vagy blokkban.

A Sudoku szabályai a következők:

1. A rácsot 9x9-es négyzetre osztják, amelyeket tovább osztanak 3x3-as blokkokra.
2. A rácsnak 9 sorban és 9 oszlopban kell lennie.
3. A rács néhány cellája már kitöltött számmal kezdődhet. Ezeket a számokat "adatoknak" nevezik.
4. A játékosok feladata a rács többi celláját kitölteni a szabályoknak megfelelően.
5. A játékosnak úgy kell kitöltenie a rácsot, hogy minden sorban, oszlopban és 3x3-as blokkban mindegyik szám (1-től 9-ig) csak egyszer szerepelhet.
6. A játék megoldása mindig egyértelmű és egyedülálló. Ez azt jelenti, hogy csak egy helyes megoldás létezik a játékban.

A Sudoku egy nagyon népszerű logikai játék, amelynek sok változata létezik, de az alapvető szabályok általában változatlanok maradnak. A játék megoldása logikai gondolkodást és problémamegoldó képességet igényel.”

A szabályok ismerete után először 4x4 táblázatot hoztam létre, majd a sikeres megoldás után áttértem a 6x6 táblázatra.

### Célkitűzés

Sudoku megoldása Excel Solver segítségével, először 4x4, majd 6x6 sudoku táblázattal.

### Excel Solver

### 3.1 Excel solver 4x4 táblázat

Sablon 4x4-es sudokut elkészítettem az Excelben (Web1.). Ezt követően a Solver működésének néztem utána, hogy megértsem a program alapját. A Miau oldalon megtaláltam a ,,SUDOKU-algoritmusok összevetése, avagy SOLVER vs. egyedi kód” című publikációt, amelynek segítségével megértettem a Solver alapjait (Pitlik L., Pitlik M., web2).

A kijelölés könnyebbsége miatt az üres cellákat összesítettem egy másik táblázatban, ezt a táblázatot neveztem el Solver változó cellának. 4x4 táblázatnál két rész táblázatot hoztam létre, amelynek a feltétele, hogy minden egyes sorban, illetve oszlopba az adott szám egyszer szerepeljen. Ezt a már fentebb említett „Darabha” függvénnyel határoztam meg és a táblázatok végén a rész táblázatok szórását határoztam meg „szórás” függvénnyel. Azonban a célérték meghatározásnál kértem segítséget Kovács János csoporttársunktól, mivel az általam meghatározott célérték (az oszlopok és a sorok maximális összege összesen (40)) nem működött. Azt a megoldást ajánlotta, hogy a két táblázat szórás összegét adjam meg a célértéknek, amelynek a célja az, hogy a sorok és oszlopoknak a szórásainak az összege 0 legyen. Ezáltal értettem meg teljesen a célértéket és ezekkel a feltételekkel a Solver helyesen oldotta meg a feladványt.

Solver feltételek:

* A solver változó cella =>1,
* A solver változó cella =<4,
* A solver változó cella = egész,
* Az oszlopok összege =10
* A sorok összege = 10
* A blokkok összege = 10

### 3.2 Sudoku Solver 6x6 táblázat

Mivel kisebb részben a Solver helyesen megtudta oldani a feladatot, ezért létrehoztam 6x6 táblázatot, amelyet 3x2 blokkokra osztottam. (Azonban ezt a táblázatot egy másik Excelben csináltam, ugyanis, ha 4x4 táblázattal szerepelt egyben külön fülön a Solver futásánál lefagyott a program). Internetről, pedig egy sablon sudokut választottam alap megoldásra (Web 3.). Ezt a táblázatot először kézzel kitöltöttem a Solver megoldás helyességének ellenőrzése miatt.

Továbbá egy külön Solver változó táblázatot hoztam létre. Illetve az oszlopok és sorok elemét itt is „Darabha” függvénnyel számoltam és ennél a táblázatnál már a 3x2 blokkhoz is létrehoztam egy ilyen táblázatot, ahol a bizonyos számok csak egyszer szerepelhetnek, az adott blokkban.

Szintén a szórások összegét adtam meg a célértéknek.

Az alábbi feltételeket adtam meg a solverben:

* A solver változó cella =>1,
* A solver változó cella =<6,
* A solver változó cella = egész,
* Az oszlopok összege =21
* A sorok összege = 21
* A blokkok összege = 21

### 3.3. Eredmény 6x6 sudokunál

Solver a megadott feltételeket teljesítette, azonban vannak olyan megoldások, amelyeket a sudoku szabályainak nem felelnek meg. Egy blokkban több azonos szám is szerepel, mégis az általam megadott feltételek teljesülnek. Többszöri próbálkozásra viszont a sudoku szabályainak megfelelő helyes megfejtést is megadta. Ebből kifolyólag a megadott feltételekkel több megoldás is van.

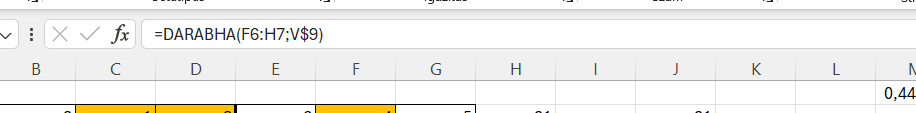
Emiatt megadtam plusz feltételnek, hogy a sor, oszlop és blokkok szórásának 0-ával kell egyenlőnek lennie, viszont a Solver hibát jelzett ki (1. ábra).

A képen szöveg, képernyőkép, szoftver, szám látható

Automatikusan generált leírás

1.ábra

Ekkor néztem át újból a hibajelzéseket és vettem észre azt, hogy képletben a $ jelek hiányoznak, így automatikusan elcsúszott a meghatározottnak vélt blokkom (2. ábra).



2.ábra

A hibát javítottam és újból a plusz feltételekkel lefuttattam a solvert, amely már csak egy helyes megoldást adott (3.ábra).

A képen szöveg, képernyőkép, sor, szám látható

Automatikusan generált leírás

3. ábra

Ennél a pontnál gondolkodtam el, hogy hogyan lehetne a feltételeket lecsökkenteni és egyszerűbben megadni, hogy sudoku szabályainak megfeleljenek. Ekkor kitöröltem ,,a sorok, oszlopok és blokkok legyenek egyenlőek 0-ával” utasítást és helyette megadtam, hogy az ,,oszolopok, sorok és blokkok szórásaik legyenek egyenlőek 0-ával” (4.ábra.). Solver megoldásig 2. sorig jutott el, utána lefagyott vagy nem végezte el helyesen a műveleteket.

A képen szöveg, képernyőkép, képernyő, szoftver látható

Automatikusan generált leírás

4.ábra

### Összegzés

Az Excel Solver megadott feltételekkel és a helyes Excel képlet használatával megoldotta megfelelően a 4x4 sudoku táblázatot. Azonban 6x6 táblázatnál lehetett megfigyelni, hogy egy adott feltételeknek több helyes megoldása is van, amely működött a 4x4-nél nem biztos, hogy megfelelően működik a 6x6-nál. OAM alapján 10-ből egyszer sikerült a helyes megoldást megjeleníteni, azonban kétszer hozott ki ugyanolyan megoldást,amely sudoku szabályainak nem megfelelő.

A leszűkített 6x6 verzió(2) viszont nem működött jól, mivel a szoftver többször lefagyott és a megoldásnál csak az első két sort oldotta meg helyesen.

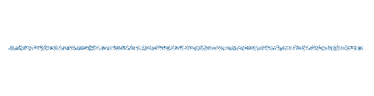
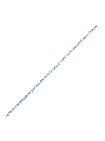
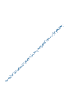
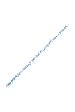
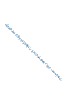
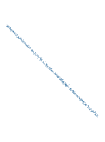
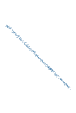
## Generálás MS Excel Solver beépülő modul segítségével (Ott Alex András – IZFQB5)

# Bevezetés

1.ábra

„A 9 × 9-es játékmező összes sorsaiba és oszlopaiba úgy kell beírni a számokat 1–9-ig, hogy minden szám csak egyszer szerepelhet egy sorban, illetve egy oszlopban. A teljes négyzetrács további 3 × 3-as négyzeteiben is minden szám csakis egyszer szerepelhet.”



A játék elején a táblázat néhány mezője már ki van töltve, a többi mezőt pedig neked kell logikus gondolkodásod segítségével kitöltened. A Sudoku egy szórakoztató és kihívást jelentő játék, amely fejleszti a logikai készséget és a koncentrációt.” (Forrás: URL = <https://sudoku.hu/sudoku-szabalya/> ) A Sudoku megoldása nagyon sok képességet vesz igénybe, a logikai gondolkodást mondhatnánk a legfőbb szükségletnek, de ez közel sem elég, (hiszen akár térben[[1]](#footnote-1) is lehet gondolkodni).



A 9×9-es Sudoku tehát 3×3 csempékből épül fel (vö. 1. ábra). 9 csempe alkot egy Sudoku-t. A megoldásához nagyon fontos a szabályok ismerete, ami több részből áll, de csak 1 egyszerű szabály hierarchikusan felépítve.



1. Ábra:Felépítés magyarázat
2. Számok 1-9 ig ismétlődnek.
3. Nem ismétlődhetnek a számok:
   1. sorban,
   2. oszlopban
   3. 3×3 csempéken belül sem

A Szabályok ennyire egyszerűek és pont ez teszi a játékot nehézzé hiszen a szabályt egy sms-ben is le lehet írni. A négyzetes és nem négyzetes Sudoku rejtvények ugyanazon alapelveken alapulnak, de a rejtvényrács szerkezetében és a játékmenetben eltérések mutatkoznak. A négyzetes Sudoku rejtvények könnyebbek és ideálisak a kezdők számára, míg a nem négyzetes Sudoku rejtvények nagyobb kihívást jelentenek a tapasztaltabb játékosoknak. Mivel nehezebb egy nem négyzetes sudoku:

1. Kevesebb a biztos cella, míg az elterjedt sudoku 9×9 csempéből áll addig könnyen átlátható 3×3 csempére tudjuk ezt felbontani az egyszerűbb szerkezetért.
2. Míg az eredeti sudokuban 81 csempe van addig egy kisebb nem négyzetes sudokuban 1-től n-ig terjednek a számok a terület csökkentésével a biztosan beírható számok száma csökken. A méretarány változással a lehetséges megoldások száma exponenciálisan növekszik.
3. Az alrács-struktúra nem add akkora támpontot.

Például ha csak a rendelkezésre álló cellákat vesszük figyelembe egy 9×9 sudokuban összesen 3,56 × 10^16 lehetséges végkifejlete van ezzel szemben például egy 8×10 lehetséges megoldásainak számát nehéz meghatározni, mivel több lehetséges megoldása is van, attól függően, hogy hány kiinduló adatot adunk meg. Azonban becslések szerint a lehetséges megoldások száma 10^64 és 10^100 között lehet, attól függően, hogy a 8×10 sudokuban mennyi a kiinduló adat.

Számomra a játék sokkal többet jelent, mint egy egyszerű „játék”. Gyerekkorom meghatározó időtöltése volt. A régi offline játékok, mint az Amőba, voltak könnyen elérhetőek, hiszen csak egy rácsos lapra volt hozzá szükség. Mind a kettő szabályzata hasonlóan egyszerű, könnyen fogyasztható és bárki által fél percen belül könnyen megtanulható.

A mesterséges intelligencia-alapú játékstratégia-felismerés kihívása azonban éppen az, hogy NEM szabad a szabályokat elmagyarázni = forráskódban, paraméterekben rögzíteni! Ez azért fontos, mert a Solver-alapú SUDOKU esetén egyelőre abból indul ki a szerző, hogy ezek a szabályok számára ismertek és így a Solver számára is megadásra kerülnek – kényszerűen egyelőre!

A Sudoku Solvert pont ez tette számomra hatalmas kihívássá és érdekes kihívássá, hiszen rengeteg időt töltöttem vele és most lehetőségem van nekem magamnak „tökéletesíteni”. A tökéletesítés itt és most azt jelenti, hogy a játékos spontán/intuitív szintjéről annak a lépésnek a kikerülésével kell egy automatikus megoldást jelentő matematikai keretet alkotni, hogy játékosként előtte meg kellett volna álmodni az univerzális = mindenkor sikert jelentő megoldási szabályrendszert (vö. szabályok ismeretének át NEM adása az MI számára). Vagyis tehát több rétege létezik a játékosi tudásnak:

1. Spontán találgatás (vö. quasi véletlenszerű próbálkozás = vak tyúk is talál szemet)
2. Intuitív ráérzés/logikus gondolkodás nem feltétlenül tudatosított komplex szabályrendszer, de tudatos részletszabályok alapján (okos találgatás = brute force a kritikus pontokon)
3. Tudatos, mindenkor egy előre (fejben, de akár papíron is) adott szabályrendszer követése (manuális robot)
4. A 3. szint szabályainak átírása forráskódba (automatizált szabály-elvű SUDOKU-megoldó szoftver)
5. Solver-alapú SUDOKU (= a szabályok beépítésre kerülnek a solver-logikába, DE a játékosnak NEM szükséges még a 2. szintet sem elérnie, vagyis nem kell tudnia megoldania egyedül, emberként a SUDOKU-t, pl. figyelem-zavaros személy esetén = definitív képtelenség, de a solver-paraméterezés kisebb koncentrációs szintű egyedi lépéseire még meg lehet a biológiai képesség)
6. MI-alapú automatizált felismerése a szabályok ismerete nélküli, csak a játékállapotokat és ezek értékét/értelmét jelentő inputoknak (vö. ha a játékállapotok ÉRTÉKE csak annyiban adott, hogy hibás lépés, ill. legitim lépés, ez már a szabályok indirekt becsorgatása a gépi tanulási folyamatba?!)

A feladat megoldása ennél viszont körökkel bonyolultabbnak bizonyult mert nem olyan egyszerű, mint azt bárki is hinné, hiszen egy számítógép csak arra képes, amit mi megadunk neki (vö. 1-6 lista fentebb az eltérő kihívásokról = az eltérő mit-is-adunk-meg stratégiákról). Kérdés: Vajon az MI jelenlegi tudása alapján nem képes önállóan megoldani egy SUDOKU-t emberi beavatkozás nélkül? Vagyis elérhető a 6. szint? De mire ez a dokumentum más által olvasásra kerül, lehet ez a kijelentés már elavulttá válik… A solver-alapúság valahol a normál szabály-elvűség és a klasszikus MI határán mozog:

* A solver pótolhatja a brute force-t
* A solver lehet genetikus/evolúciós algoritmussal támogatott
* A solver lehet egyenletrendszer-alapú

# Feladat-értelmezés

Először is meg kellett értenem a feladat lényegét = konvertálnom kellett a már eleve értett szabályokat a solver-paraméterezés (vö. speciális nyelv) szabályainak felhasználásával, vagyis el kellet magyaráznom a solver-paramétertér (nyelvi lehetőségek) segítségével mit jelent egy-egy szómágikus szabályrészlet solver-ül:

Több eshetőséget is meg kell vizsgáljunk, hogy meg tudjuk oldani. Megpróbáltam a „problémát” kisebb részletekre bontani, hogy könnyebben megérthető legyen, de az alap „probléma” a következő:

Szükségünk van egy programra, ami megvizsgálja, hogy az általunk beírt számok előre meghatározott szabályoknak, hogyan felelhet meg és ha megfelelhet akkor pedig, hogyan.

Jobbra lévő kép (2. ábra) egy általam írt próbálkozás a lehetséges végső megoldás irányába tett kísérletre.

A grid of squares with numbers

Description automatically generatedAz Excel-képlet csak egyetlen dolgot vizsgál, ami nem más: mint,hogy az általunk megadott 9x9 es (nagy) csempéken mely számok (hányszor) fordulnak elő.

Hiszen egy-egy kockakövön (táblázati pozícióban, cellában) 1-9 ig fordulhat elő szám. Az adott 3×3-as (kis) csempén 9 számjegy fordulhat elő és mindegyik „unique” = nincs belőle még egy. A második általunk vizsgált szempont ,hogy az általunk beírt számok (1;…;9) sorban és oszlopban sem ismétlődnek, hiszen sorban és oszlopban is egyedi számokat kell tartalmaznia minden logikai egységnek (vö. nagy csempe, ill. játéktér).

A program kód itt csak azt vizsgálja,hogy a megadott 1x1 kockaköveken milyen egyedi számok jelennek meg , így egy mellékelt beviteli táblázatban ha megadunk egy számot ebben a programban (2. ábra) az összes lehetőséget megvizsgálja és kitörli az adott számot a lehetséges opcióink közül..

2. Ábra: Egy cellára jutó lehetséges opciók

Beírunk egy számot (3.ábra) ez most az egyes szám lesz és beírtuk a kisegítő 9×9 1 csempélyébe. az ábrán jól látható, hogy az összes lehetséges helyről ahova az egyes számot beírhattuk volna , kikerült ennek a lehetősége.

A grid of squares with numbers

Description automatically generatedEzzel a módszerrel megoldható egy sudoku viszont csak kézi(manuális) bevitellel működik. solver vagy MI bevonásával a függvények semmire nem jók mert egyik se tudja az eredményeit kezelni. Ezzel is többnek nevezhetjük az emberi logikát hiszen mint egy autómata tesztelésnél is rengeteg hibát megtalálhat, de olyan adatot ami számára értelmezhető és helyes nem működik így csak manuális teszteléssel lehet megtalálni. Ez ebben az esetben sincsen másként.

Ez volt az elsődleges ötletem,hogyan tudnám megvalósítani azt,hogy egy Sudoku megoldót készítsünk az Excel solverrel. De a folyamatos manuális bevitelt fel kell cserélnem biztonságos csempékre és abból a programnak önállóan kell megoldaia.

3.ábra Működés közben

# A screenshot of a computer Description automatically generated Első próba

Először a „problémát” kisebb részre bontottam és a következő részletet a „mystyc 3x3” megoldó részletből, és a megoldáshoz vezető út már itt félre lett általam definiálva. Hiszen ez a program nem oldja meg az általam definiált „problémát” csak létrehozni (generálni) egy lehetséges végkifejletet.

4.ábra Paraméterezés (saját programkód))

Itt viszont ezzel még nem voltam tisztában. Hogy az általam megadott paraméterek nagyon leszűkítik a lehetséges megoldások számát. Mindeközben a 3×3 sudokunak 512 lehetséges megoldása van.

A képlet (forrás: Sudoku Solver (version 1).xlsm, Mistery 3x3 munkalap, 4.ábra ) itt még elég egyszerű volt. Adott egy 3 négyzet (cella) széles és 3 négyzet (cella) magas mágikus kocka, amiben egy szám egyszer szerepelhet 1től 9ig. A 15-ös szám a (kis) csempe szélén segített mert minden irányban meg kell lennie a 15nek ez azt eredményezi, hogy a csempe egyedi számokat fog használni, de ezzel erősen korlátozottak a megoldási lehetőségeink.

# Második próba

Éreztem, hogy az első rész egy ígéretes irány, hiszen már sikerült egy-egy lehetséges megoldást generálnom. Ami egy ígéretes irány.

Az alternatív megoldások mindegyikét legenerálni pl. úgy lehet elkezdeni, hogy kikötjük, pl. a bal felső cella ne legyen az az érték, ami már az éppen adott sikeres megoldásban szerepel és újra futtatjuk a Solver-t.

A grid of numbers

Description automatically generatedMásodjára lefuttattam a solvert és sikeresen generált 3x3 csempét 9 részre klónozni, hogy megoldást találhassak az esetleges 9x9 es Sudoku-ra. Itt a kihívás más megfogalmazásban az: hogyan lehet a fentebb előállítottalternatív (kis) csempéken egy nagy 9\*9-es helyesen kitöltött játéktérként értelmezni, lévén a nagy játéktér minden csempéje helyes kell, hogy legyen, s ezek vizuálisan ellenőrizhetően AKÁR ismétlődhetnek is, hacsak az nem egy spontán felismert új szabály, hogy legalább 9 alternatív kis (helyes) csempe létezik, s így ezek egy nagy 9\*9-es játéktéren SOHA nem ismétlődhetnek. Matematikusok számára szép kihívás annak bizonyítása, hogy létezik-e olyan SUDOKU egyáltalán, ahol minden kis csempe más, vagy nem minden kis csempe más? stb. (Ez nem a BPROF képzés szintje, de egy MSC (informatika?, ill. BSC matematika?) szint esetén már vélhetően ez a kihívás is kezelendő/kezelhető…)

5. ábra: 3×3 csempe 9×

A screenshot of a computer error

Description automatically generatedSajnos ez a gondolat egyelőre még nem vezetett megoldásra, mert itt plusz 2 szabályt (1. nem lehetnek a sorokban se azonos számok, 2. nem lehet az oszlopokban se azonos számok) kellett bevezetni, aminek előzetesen még nem voltam tisztában, hogy sajnos ezen kinézete a táblázatnak ezt nem fogja engedni, mert az Excel Solver nem tud egymástól elhatárolt mezőket egybefüggően kezelni egy azonos szabályra, az összefűzés következténben pedig összeomlik (a kipróbált megoldás esetében legalább is). Sajnos az Excel log- vagy a windows log-ból nem sikerült azt kinyernem, hogy mi okozhatta a hibát, a Windowsnak írt hibaüzenetem a hiba konkrét leírására, pedig semmilyen válaszüzenetet nem eredményezett eddig. (vö. mellékletben képernyőképekkel alátámasztott kommunikáció)

Így a feladatban megpróbáltam a legtöbb tapasztalatot leszűrni intuitív módon,hogy mi lehet az, amivel tudnék fejldőni ebben a fázisban. (Na ez az az intuitivitás, amire egyelőre az MI a szó szoros értelmében nem képes: hiába látja be egy MI-alapú keretrendszer, hogy valami nem sikerült eddig, nem ijed meg, nem kapcsol túlélési fokozatba, nem álmodja meg a megoldást…)

Ami nem más lett, mint hogy a feladatot nem tagolhatom szét, (mert a Solver ezt közvetlenül nem tudja kezelni).

A következő részben? pedig a sor és oszlop nem lehet azonos számok. A képen (x. ábra) láthatjuk,hogy a megadott kritériumoknak megfelel, hiszen a sor, oszlop és átlók is stimmelnek a 15-ös összegszabályra és a 9x1 es kockaköves részek sorra és oszlopra is 45-öt adnak (45-ös összegszabály), ami nem más, mint 1-9 ig a számok összeadva.

* 15 (A sorok és oszlopok összeadása 45 mivel 1től 9ig ha összeadjuk a számokat 45, magyarul 45/3)
* 45 (1-9 ig a számok összértéke, mivel más szám nem fordulhat elő a 3×3 csempében)
* Sor 45 (Mivel a sorban 1-9 ig vannak a számok vontkozik rá a 45ös szabály)
* Oszlop 45 (Mivel a oszlopban 1-9 ig vannak a számok vontkozik rá a 45ös szabály)

# A grid of numbers with a green box Description automatically generatedHarmadik próba

Előző 2 próbából tanulva, már egybe a mozaik 9db 3×3-as csempe. Afeladat ami az lett volna,hogy a 3×3 alldifferent és a sor és oszlopok értékének 45 nek kellene lennie a solver szerint jó megoldást kapott a 15 értékkel, ami megfelel a a kritériumoknak.És a solver értelmezésében a feladatot sikeresen elvégezte páros számok összeadásával páratlan számot kapott és a kért 45 helyett 15 ör, erre többedmagammal se találtam semmilyen magyarázatot, de a program alig pár ezred másodperc alatt erre a következtetésre jut Így mivel elég sok kérdés maradt ezzel kapcsolatban bennem, úgy éreztem, hogy az Excel-nek nem adtam elég feltételt, hogy meg tudja oldani a problémát. Így ezt a próbálkozást tökéletesnek ítéltem meg arra vonatkozóan, hogy merre nem haladhatok tovább!

# Negyedik próba

Ezt a kísérletet érzem a legnagyobb tanulságomnak, mert az volt az a rész, ami a lehető leges-legmesszebb juttatott a valós Sudoku megoldás felé és ennek ellenére a legkomolyabb zsákutcámnak bizonyult.

A Kód nem alkalmas arra, hogy megoldjon Sudoku-t, de tökéletesen alkalmas arra, hogy egy-egy Sudoku-t létrehozzon:

Így egy fél sikernek éreztem, hiszen lehet nem vagyok képes megoldani A Problémát, de képes vagyok egy új tesztesetet létrehozni(8.Ábra). A szabályok amit a programnak adtam egyszerűek voltak.

Minden 3x3 AllDIfferent

Minden Sor AllDIfferent

Minden Oszlop AllDIfferent

Ez pedig a következő képet (8. ábra) eredményezte,hogy a sor számláló ami a különálló 3x3 csempéket adja össze 45 lesz, ezzel szemben a legelső próbához képest már közel se 15 az eredmény leggyakrabban előforduló száma.Jelen Sudoku, amit létrehoztam, közel se tudom, mennyi időbe telhetett mert 1 óra után és 1millió „subproblem” után én magára hagytam a gépet A screenshot of a crossword puzzle

Description automatically generatedés lefeküdtem aludni.

8. ábra negyedik próba

De a program sikeresen tudott generálni Sudoku-t, tény és való, hogy ezzel szemben nem tud megoldani még egyet sem és abban az esetben, ha rögzítem a helyeket, hogy vegye bele a megoldásba akkor lefagy. Mivel az AllDifferent kikötés miatt, nem lehetséges számokat rögzíteni.

# Ötödik próba

A screenshot of a computer

Description automatically generatedEzen próba alkalmával a megoldást más irányból próbáltam megközelíteni, a Reddit segítségét vettem igénybe, ahol egy általam írt kódot tettem közzé és több ember segítségét kértem, hogy létrehozzunk egy VBS (VBS = Visual Basic Script) kódot az általam írt Python kódból. Ezt a kódot a végén egy már publikált szerzőtől küldték el számomra (https://tutorialhorizon.com/excel/vba-excel-sudoku-solver/)szolgáltatta számomra, hogy segítse a jelenlegi Sudoku Solver előre mozdulását a siker irányába, amivel a régi programot előre tudtam mozdítani mivel, sikerült egy VBA (VBA = Visual Basic Application) macro-t létrehoznom és ezzel az általam generált Sudoku-kat, amit generáltam, le tudom tesztelni, hogy helyesek-e? Sajnos ez a teszteset „csak” abban segített, hogy rájöjjek a „Negyedik próba” ban hol hibáztam és hogyan tudnám javítani. De ezzel a Solver nem mutatott sokkal nagyobb javulást, de az idő, ami ahhoz kell, hogy sikerüljön megoldást találni, nagyban csökkent, már 1 óra után mikor elindítjuk(19:52 indítás első ellenőrzés 20:52) a generálást (Solver) akkor már az utolsó

9. ábra ötödik próba

részeket tökéletesíti. Abban az esetben, ha sikerül a teljes folyamatot végig futtatni, természetesen dokumentálom tovább.

A screenshot of a computer

Description automatically generatedA két kép között további 1 óra telt el (22:02) és láthatólag a megoldás közelébe nem sikerül jutnia a Solvernak.

10. ötödik próba

# Sudoku megoldása az Excel Solver segítségével: Konklúzió

A dokumentáció a Sudoku megoldására tesz kísérletét az excel solverrel. Öt próbálkozáson ment keresztül az adott folyamat ahol szükségességét éreztem,hogy mindegyik értékes tanulságból konklúziót vonjak:

Első próba:

Cél: 3×3-as csempén belül 1től 9ig számok generálása ismétlődés nélkül

Megállapítás: A paraméterek túlságosan leszűkítik a lehetséges megoldásokat.

Második próba:

Cél: Alternatív 3×3 csempék generálása és összeillesztése egy 9x9-es "nagy" csempévé

Kihívás: A sorokban és oszlopokban ismétlődő számok kizárása

Probléma: Az excel solver nem tudja kezelni ha a csempék szét vannak bontva.

Harmadik próba:

Cél: A "nagy" csempében lévő sorok és oszlopok 45-ös összegének elérése

Megállapítás: A páros számok összeadása páratlan eredményt ad (15), ami nem a várt 45.

Következtetés: Az excelnek nem adtunk elegendő feltételt a probléma megoldásához.

Negyedik próba:

Cél: Sudoku generálása

Megállapítás: A program képes sudoku-t generálni, de nem képes azt megoldani.

Tanulság: A program alkalmas tesztesetek létrehozására.

Ötödik próba:

Cél: A Solver javítása külső kódok bevonásával

Eredmény: A VBA makró segített a hibák azonosításában és a generálási idő csökkentésében.

Megállapítás: A Solver továbbra sem hatékony a megoldás megtalálásában.

A saját egyéni konklúzióm, hogy a jelenlegi próbák nem hoznak áttörést a probléma megoldására, így mindenféleképpen más megközelítés vagy perspektíva váltás szükséges.

# Hatodik próba

’Sudokut nem csak 9×9 csempén lehetséges játszani.’ Ez volt az első lehetőség ami eszembe jutott mikor arra gondoltam, hogy más megközelítésre van szükségem a probléma megoldásához.

De nem tudtam, hogy honnan kellene ezt megközelítenem.

Elsődlegesen a 4×4 sudokura gondoltam, mivel az összetettségét figyelembe véve jó lehetőség és próbának tökéletes, hogy új konklúziót vonhassak le.

4×4 sudoku szabályai:

1. Minden oldala 4 csempe hosszúságú
2. A számok 1től 9ig lehetnek
3. Sorban nem lehet szám ismétlődés
4. Oszlopban nem lehet szám ismétlődés

11. Ábra Hatodik próba


11.ábra Hatodik próba

Hogy könnyebben megértsem minden cella sarkát megjelöltem, hogy tudjam melyik részhez tartozik. Ezzel még magam se voltam tisztában, de a feladatot félre értelmeztem. Mivel a végeredményben a kitűzött szabályokat nem sikerült megvalósítanom, de sikerült egy sokkal izgalmasabb megoldásra rálelnem.

A 4×4 sudokun belül sikerült 3×3 sudokut generálnom méghozzá ismétlődés nélkülit(a különböző színek jelölik mely számok mely 3×3 sudokuhoz tartoznak a 11. ábrában)

# Hetedik próba

Előző próbában kitűzött célokat tovább gondolva, sikerült egy 4×4 generátor készítenem.

A screenshot of a computer

Description automatically generated

12.ábra Hetedik próba

Ebben a próbában sikerült az általam kitűzött célokat elérnem, miszerint

4×4 sudoku szabályai:

1. Minden oldala 4 csempe hosszúságú
2. A számok 1től 9ig lehetnek
3. Sorban nem lehet szám ismétlődés
4. Oszlopban nem lehet szám ismétlődés

# Nyolcadik Próba

A grid of numbers on a grid

Description automatically generatedEbben a próbában újabb feltételeket határoztam meg az eddigi meglévő feltételek mellé.

13.ábra Nyolcadik próba

Az új feltételek:

* Minden 3×3 csempe 45-nek kell lennije (hiszen a számok 1től 9ig összeadva)
* A 9×9 csempének 405-nek kell lennije(hiszen 9\*45=405)
* 9×9 csempén belül a számokra igaz a következő egyenlet ([legyen jelölve a szám most X-el ( 9>= X >=1) ]

Régi feltételek:

* Minden sorban csak 1től 9ig szerepelnek a számok ismétlődés nélkül
* Minden sornak 45 az értéke (hiszen a számok 1től 9ig összeadva)
* Minden oszlopban csak 1től 9ig szerepelnek a számok ismétlődés nélkül
* Minden oszlopnak 45 az értéke (hiszen a számok 1től 9ig összeadva)
* 3×3 csempében a számok 1től 9ig ismétlődés nélkül

# Sudoku megoldás Python programnyelven (Bodicsi Viktor József – EJIOIM)

## Bevezetés:

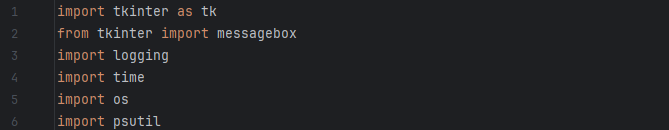
**Sudoku szabálya:**A sudoku egy logikán és kombinatorikán alapuló kiváló számelhelyezéses rejtvény. A cél a 9×9-es tábla kitöltése a számokkal úgy, hogy minden egyes oszlop, sor és mindegyik 3×3-as mező, (a főrácsot felépítő alrácsok) 1-től 9-ig tartalmazzák a számokat. A kiinduló helyzetben egy részlegesen kitöltött táblát kapunk, amelynek általában egyetlen megoldása van. A kitöltött feladvány mindig egyfajta latin négyzet, egy megkötéssel az egyes területek tartalmára vonatkozóan. Például egy sudokun belül ugyanaz az egész szám nem jelenhet meg kétszer ugyanabban a sorban, oszlopban vagy a 9 db 3×3-as altábla belsejébe.

**A programról pár szóban:**A Sudoku solver Pythonban írt alkalmazás, amely arra szolgál, hogy a delikvens egy 9x9-es Sudoku feladványt készít és egy grafikus felhasználói felülettel ellátott alkalmazás megoldja a feladványt. Az alkalmazás célja, hogy segítse a felhasználókat a Sudoku feladványok gyors megoldásában.A program a backtracking algoritmusra épül, amely rekurzívan próbálja megoldani a Sudoku feladványt. A felhasználó egy grafikus felhasználói felületen adhatja meg a Sudoku feladványt, majd megoldhatja azt a "Megoldás" gomb segítségével.

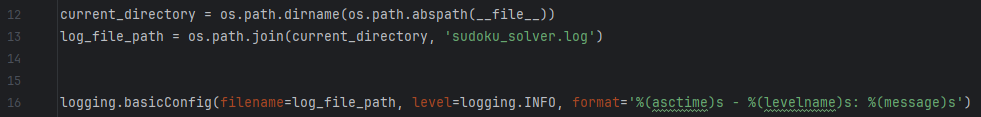
**Idegen, illetve szakszavak a leírásban:**

**Backtracking:** A visszalépéses keresés (angolul: backtracking) egy általános algoritmus bizonyos számítási problémák, különösen a korlátkielégítési probléma (más néven kényszerkielégítési probléma) összes (vagy legalább néhány) megoldásának megtalálására. Az algoritmus fokozatosan felépíti a megoldás útját (részmegoldások), és elhagyja azokat az útvonalakat ("visszalépések"), amelyekről megállapítja, hogy nem adnak érvényes megoldást. **Rekurzív:** A rekurziót leggyakrabban a visszalépéses keresés (angolul backtracking) jellegű algoritmusokban szokták használni. A rekurzív algoritmusok ciklussá formálása „automatikusan” elvégezhető verem bevezetésével.  
**GUI:** A grafikus felhasználói felület vagy grafikus felhasználói interfész (angolul graphical user interface, röviden GUI) a számítástechnikában olyan, a számítógép és ember közti kapcsolatot megvalósító elemek összessége, melyek a monitor képernyőjén szöveges és rajzos elemek együtteseként jelennek meg.  
**Tuple:** A tuple típus egy, vagy több adattag tárolására szolgáló generikus objektum. Leginkább arra találták ki, hogy ha kettő, vagy három adattagot össze kell fogni, akkor ne kelljen azok tárolására kódba egy új osztályt bevezetni.  
**Iterál:** Az informatikában iterációnak nevezzük valamely eljárás ismétlődő végrehajtását. Az iteráció esetén a programozónak saját kezűleg kell gondoskodni a leállásról, különben az iteráció végtelen ciklussá válhat. Igaz, egyes esetekben ez kívánatos lehet, ilyen például egy játékprogram fő ciklusa.  
**Argumentum:** Egy olyan érték vagy kifejezés, amelyet egy függvénynek vagy eljárásnak átadnak, hogy azt a függvény vagy eljárás felhasználja a működése során. **Mátrix:** A mátrix vízszintes vonalban elhelyezkedő elemei sorokat, függőleges vonalban elhelyezkedő elemei oszlopokat alkotnak. Egy m sorból és n oszlopból álló mátrixot m-szer n mátrixnak nevezzük (m × n), az m és n pozitív egész számok a mátrix dimenziói (kiterjedései). A mátrix dimenzióit mindig először a sorok számával, majd az oszlopok számával adjuk meg.  
**Lebegőpontos szám:** A számítástechnikában a lebegőpontos számábrázolás lehetővé teszi a valós számok kezelését véges tárhely esetében, széles skálát fedve le a számhalmazon belül.

## Program felépítése:

1. 

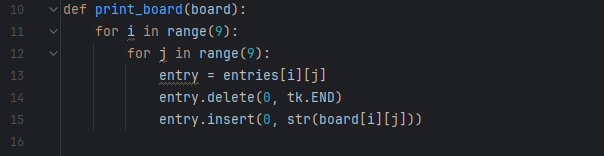
A **„tkinter”** modul egy grafikus felhasználói felület készítésére szolgáló eszközkészlet a Pythonban. Importáljuk a **„tkinter”** modult, majd a **„messageboxot”** a **„tkinter”** modulból. A **„messagebox”** segítségével üzenetdobozokat lehet megjeleníteni a felhasználó számára.   
**„import logging”**: A **logging** modul, lehetővé teszi a naplózás különböző szintjeinek (pl. hibakeresés, figyelmeztetés, információ stb.) kezelését és naplózását.  
**„import time”**: A Python beépített **„time”** modulja, teszi lehetővé az idővel kapcsolatos műveletek végrehajtását. A mi esetünkben most arra lesz szükség, hogy mérjük mennyi idő alatt oldotta meg a program a sudoku feladványt.   
**„import os”**: Ezzel a kódsorral betöltjük a Python beépített **„os”** modulját, amely lehetővé teszi az operációs rendszerrel való kommunikációt, például fájlkezelést, könyvtárkezelést vagy környezeti változók elérését.  
**„import psutil”**: Ezzel a kódsorral betöltjük a psutil modult, amely lehetővé teszi az operációs rendszerrel kapcsolatos információk lekérdezését és kezelését, például a rendszeres erőforrások (CPU, memória stb.) felhasználásának ellenőrzését.



Ezekkel a kódsorokkal beállítjuk a programnak a naplózást.  
**„current\_directory = os.path.dirname(os.path.abspath(\_\_file\_\_))”**: Ezzel kódsorral meghatározzuk a program aktuális munkakönyvtárát. Az **„os.path.abspath(\_\_file\_\_)”** visszaadja a jelenlegi futtatható fájl teljes elérési útját, míg az **„os.path.dirname()”** függvénnyel kinyerjük ennek a fájlnak a könyvtárát. Ezt az elérési utat tároljuk el a **„current\_directory”** változóban.  
**„log\_file\_path = os.path.join(current\_directory, 'sudoku\_solver.log')”**: Ezzel a kódsorral létrehozzuk a naplófájl elérési útját. Az **„os.path.join()”** függvény segítségével összefűzzük a jelenlegi munka könyvtárat (**„current\_directory”**) a naplófájl nevével (**„sudoku\_solver.log”**). Ezt az elérési utat tároljuk el a **„log\_file\_path”** változóban.  
**„logging.basicConfig(filename=log\_file\_path, level=logging.INFO, format='%(asctime)s - %(levelname)s: %(message)s')”**: Ezzel konfiguráljuk a naplózást a program számára. A **„filename”** argumentum megadja a naplófájl elérési útját, a **„level”** argumentum meghatározza a naplózási szintet (itt **„logging.INFO”** -t állítunk be, ami az információs üzeneteket jelenti), a **„format”** argumentum meghatározza a naplózási üzenet formátumát, amely tartalmazza az időbélyeget **„(%(asctime)s)”**, a naplózási szintet **„(%(levelname)s)”** és magát az üzenetet **„(%(message)s)”**. A **„basicConfig()”** függvényt használjuk a naplózás alapvető beállításainak meghatározására.



Ezek a kódsorok a **„psutil”** modul segítségével lekérdezik a rendszer erőforrásainak kezdeti felhasználását, és az alábbiakat jelentik:  
**„start\_cpu\_usage = psutil.cpu\_percent()”**: Ezzel a kódsorral lekérjük a CPU kihasználtságát százalékban a **„psutil.cpu\_percent()”** függvény segítségével. Ez az érték jelzi, hogy mennyi százalékban van kihasználva a számítógép CPU-ja az adott pillanatban.  
**„start\_ram\_usage = psutil.virtual\_memory().used / (1024 \*\* 3)”**: **„psutil.virtual\_memory().used”** Ezzel a kódrészlettel meghívjuk a psutil modul virtual\_memory() függvényét, ami visszaadja a virtuális memória (RAM) állapotát. A used attribútummal lekérjük a használt memória mennyiségét byte-ban. **„/ (1024 \*\* 3)”** Ezzel a részlettel pedig a byte-ból gigabyte-ba konvertálja az eredményt. Mivel 1 gigabyte = 1024 megabyte, ezért a byte-ból gigabyte-ba való átváltáshoz a byte-ot elosztjuk 1024^3-mal (1024 \* 1024 \* 1024), ami 1 gigabyte.

1. ****

Ezzel a függvénnyel egy Sudoku tábla állapotát jelenítjük meg. A függvény egy 9x9-es rácsot járunk végig két egymásba ágyazott **„for”** ciklussal. Minden ciklus 0-tól 8-ig fut, mivel egy 9x9-es tábláról van szó. Az **entry.delete(0, tk.END)** rész törli az adott cella jelenlegi tartalmát, majd az **entry.insert(0, str(board[i][j]))** rész behelyezi az adott cellába a megfelelő értéket a **’board’** mátrixból.

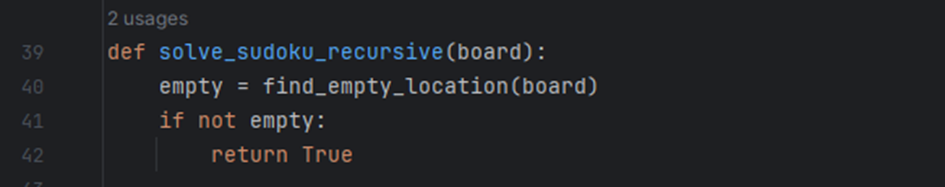


„**def solve\_sudoku():”**: Ez a sor megkezdi a **„solve sudoku”** függvény definiálását, amely a feladvány megoldására szolgál.

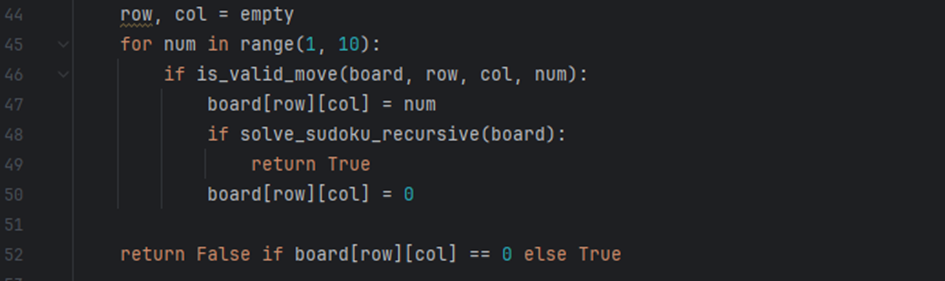
A **„try”** blokk azon kódrészleteket tartalmazza, amelyeket a Python megpróbál végrehajtani. Ha valamilyen hiba lép fel ezekben a blokkokban, akkor az **„except”** blokkok kezelik ezeket a hibákat.  
**„start\_time = time.time()”**: Elindítjuk az időmérést a Sudoku feladvány megoldásának kezdete előtt.  
„**board = []** és **for i in range(9):”**:Tulajdonképp itt hozzuk létre a Sudoku táblát.Először egy üres lista **„board”** létrehozása, amelyben tárolni fogjuk a Sudoku táblát. Két egymásba ágyazott **for** ciklussal bejárjuk a 9x9-es rácsot, létrehozva egy 9x9-es mátrixot. A belső ciklusban minden egyes cella értékét beolvassuk a GUI-ban lévő beviteli mezőkből, és hozzáadjuk a „**row”** nevű sorhoz.  
**„if entry\_text.strip() == '' or not entry\_text.strip().isdigit():”**:Az aktuális beviteli mező értékét beolvassuk és eltároljuk az **„entry\_text”** változóban. Ellenőrizzük, hogy az **„entry\_text”** üres-e (**” ”**) vagy nem szám-e **„not entry\_text.strip().isdigit()”**. Ha igen, akkor a cella értékét 0-ra állítjuk, mivel a Sudoku-ban az üres cellákat általában 0-val jelöljük.  
**„if solve\_sudoku\_recursive(board):”**:Meghívjuk a **„solve\_sudoku\_recursive függvényt”** a **„board”** mátrixszal, hogy megpróbáljuk megoldani a Sudoku-t.  
Ha a függvény visszatérési értéke igaz (sikerült megoldani a Sudoku-t), akkor a **„print\_board”** függvény segítségével kinyomtatjuk a megoldást a konzolra.  
**„elapsed\_time = time.time() - start\_time”**: Számolja az eltelt időt a feladvány megoldásához.  
**„end\_cpu\_usage = psutil.cpu\_percent()”**: Lekérdezzük a CPU kihasználtságát a **„psutil.cpu\_percent()”** függvény segítségével az adott pillanatban.  
**„end\_ram\_usage = psutil.virtual\_memory().used / (1024 \*\* 3)”**: Lekérdezzük a memória (RAM) kihasználtságát az adott pillanatban a **„psutil.virtual\_memory().used”** függvény segítségével.  
**„cpu\_usage = end\_cpu\_usage - start\_cpu\_usage”**: Ezzel a sor kiszámítjuk a CPU felhasználás változását a Sudoku feladvány megoldása során. Ehhez kivonjuk a megoldás előtti CPU kihasználtságát „**start\_cpu\_usage”** a megoldás utáni CPU kihasználtságból **„end\_cpu\_usage”**.   
**„ram\_usage = end\_ram\_usage - start\_ram\_usage”**: Ezzel a sorral kiszámítjuk a RAM felhasználás változását a Sudoku feladvány megoldása során. Ehhez kivonjuk a megoldás előtti RAM kihasználtságát **„start\_ram\_usage”** a megoldás utáni RAM kihasználtságból **„end\_ram\_usage”**.  
**„logging.info('Sudoku feladvány megoldva! Megoldás ideje: %.2f másodperc, CPU használat: %.2f%%, RAM használat: %.2f GB', elapsed\_time, cpu\_usage, ram\_usage)”**:

Naplózzuk az információt a megoldott feladványról és az eltelt időről, illetve a felhasznált erőforrásról. A „**%2f**” egy formázó irányelv, ami azt jelenti, hogy egy lebegőpontos számot kell beilleszteni a szövegbe, két tizedesjeggyel. Az „**elapsed\_time**” változót azért illesztjük be ide, hogy megjelenítsük az eltelt időt a megoldásig, a **„cpu\_usage”** illetve a **„ram\_usage”** változókkal pedig megjelenítjük a felhasznált erőforrást a feladvány megoldásáig.

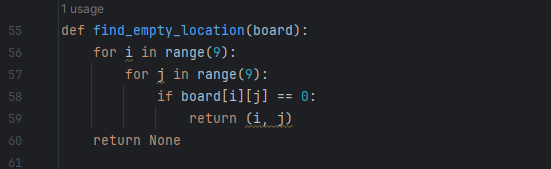
**„messagebox.showinfo("Megoldva", "A Sudoku feladvány megoldva! Megoldás ideje: {:.2f} másodperc".format(elapsed\_time))”**: Egy felugró ablakban tájékoztatást adunk a felhasználónak a megoldás sikerességéről és az eltelt időről.  
**„else:”**: Viszont ha nem sikerül bármilyen okból kifolyólag megoldani a Sudoku feladványt akkor egy felugró ablakban hibaüzenet fog megjelenni. A hibaüzenet megjelenítéséről a **„messagebox.showinfo(„Nincs megoldás” , „Nincs megoldás, hibás a Sudoku feladvány!”)”** gondoskodik.   
**„logging.warning('Nincs megoldás, hibás a Sudoku feladvány!')”**: Ezzel a sorral pedig naplózzuk a figyelmeztetést a megoldás sikertelenségéről.  
**„except Exception as e:”:** Ha bármilyen hiba történik a függvény végrehajtása során, az **„except”** blokk kezeli ezeket a hibákat.

****

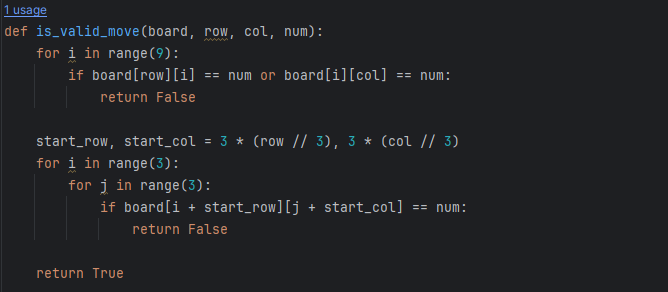
A program ezen részével vizsgáljuk meg, hogy a feladványunkban maradt-e üres cella. Amennyiben nem maradt üres cella akkor visszatér igazzal, hogy sikerült megoldani a Sudoku feladványt.  
**„def solve\_sudoku\_recursive(board):”**: Ez a sor határozza meg a **„solve\_sudoku\_recursive”** függvényt, amely a Sudoku feladvány rekurzív megoldását végzi el.  
**„empty = find\_empty\_location(board)”**: Meghívjuk a **„find\_empty\_location”** függvényt a **„board”** mátrixszal, hogy megkeressük az első üres cellát a Sudoku táblán. Az eredményt az **„empty”** változóban tároljuk.  
**„if not empty:”**: Itt ellenőrizzük, hogy találtunk-e üres cellát a feladványunkban, amennyiben nem (tehát az **„empty”** változó üres), akkor a Sudoku feladványt sikerült megoldani. Ebben az esetben **„True”** értékkel visszatérünk és jelezve ezzel, hogy a megoldás sikeres volt.

1. 

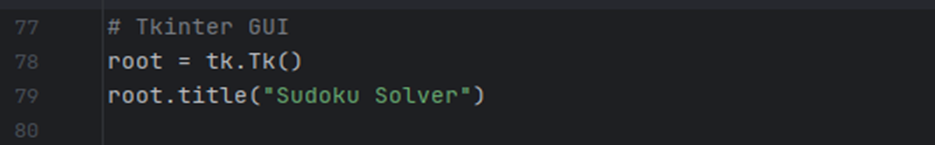
Ez a rész a rekurzív megoldás lényegét tartalmazza. Ismétli az üres cellában a lehetséges számokat (1-től 9-ig), és megpróbálja behelyezni azokat. Ha egy szám megfelelőnek bizonyul, akkor rekurzívan hívja önmagát, hogy megoldja a Sudoku feladványt a frissített táblával.  
**„row, col = empty”**: Kinyeri az üres cella sor- és oszlopinformációit, amelyeket korábban a **„find\_empty\_location”** függvény talált meg.  
**„for num in range(1, 10):”**: A **„num”** változó 1-től 9-ig ismétel, reprezentálva az összes lehetséges számot, amit az üres cellába lehet behelyezni.  
**„if is\_valid\_move(board, row, col, num):”**: Ezzel a sorral ellenőrizzük, hogy a szám ami az üres cellába került helyes-e.   
**„if solve\_sudoku\_recursive(board):”**: Amennyiben a szám helyes a cellába akkor a függvény meghívja önmagát a frissített **„board”** mátrixszal.  
**„return True”**: Ezután ha a rekurzív hívás sikeres volt, azaz az összes többi üres cellát sikerült kitölteni akkor a függvény **„True”** értékkel tér vissza.   
**„board[row][col] = 0”**: Amennyiben nem volt sikeres a rekurzív hívás, azaz nem tudta megoldani a Sudoku-t az adott lépésnél akkor visszaállítja a cella értékét és továbblép a következő szám vizsgálatára.   
**„return False if board[row][col] == 0 else True”**: Ha nem talál megoldást az adott állapotban, akkor a függvény visszatér egy **„False”** értékkel.

1. ****

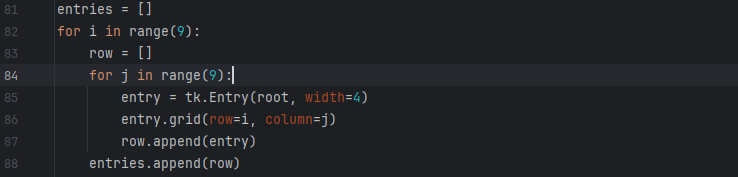
Ez a rész a Sudoku táblán keres üres cellát, vagyis olyan cellát, amelynek az értéke 0. Ha talál ilyen cellát, akkor visszatér annak a sor- és oszlopinformációval, amelyben az üres cella található. Ha a táblán nincs üres cella, akkor **„None”** értékkel tér vissza.  
**„for i in range(9):** és **for j in range(9):”**: Az i változó 0-tól 8-ig, a j változó pedig 0-tól 8-ig iterál a 9x9-es tábla celláin. Ezután megvizsgálja, hogy az adott cella értéke 0-e. Ha igen, akkor visszatér az adott cella sor- és oszlopinformációival a (i, j) tuple formájában.  
**„if board[i][j] == 0:”**: Amennyiben talál egy olyan cellát, amelynek az értéke 0, akkor a függvény visszatér az adott cella sor- és oszlopinformációival.  
**„return None”**: Ha végig járta az összes cellát, és nem talált üres cellát, akkor a függvény **„None”** értékkel tér vissza, jelezve, hogy nincs üres cella a táblán.

1. 

Ez a függvény ellenőrzi, hogy egy adott szám (num) helyes-e az adott cellában (row, col) a Sudoku táblán.   
**„for i in range(9):”**: Az i változó 0-tól 8-ig iterál, ami a tábla méretének megfelelő.  
**„if board[row][i] == num or board[i][col] == num:”**: Az **„if”** feltétel ellenőrzi, hogy az adott szám **„num”** már előfordult-e az adott sorban **„board[row][i] == num”** vagy az adott oszlopba n **„board[i][col] == num”.**   
**„return False”**: Amennyiben az adott szám már szerepel egy másik cellában akkor azonnal egy **„False”** értékkel tér vissza, jelezve, hogy ez az adott szám nem helyes az adott cellában.  
**„start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)”**: Ez a rész a 3x3-as részterületet vizsgálja meg a Sudoku táblán belül, ahol az adott cella található. Először kiszámítja a 3x3-as részterület bal felső sarkának sor- és oszlopinformációit (**start\_row** és **start\_col**) a **„row”** és **„col”** változók felhasználásával. Ezután a **„for”** ciklusokkal végig iterál a 3x3-as részterületen, és ellenőrzi, hogy az adott szám már előfordult-e a részterületen belül. Ha az adott szám már előfordult a részterületen belül, a függvény azonnal **„False”** értékkel tér vissza.  
**„return True”**: Amennyiben minden lépés helyes, azaz nem találtunk olyan esetet, hogy az adott szám már előfordult volna az adott sorban vagy oszlopban vagy egy 3x3-as részterületen akkor a függvény **„True”** értékkel tér vissza, jelezve azt, hogy az adott szám helyes az adott cellában.

1. 

**„root = tk.Tk()”**: Létrehoz egy Tkinter objektumot, amely az ablakot ábrázolja. Ezt az objektumot a **„root”** változóhoz rendeljük, amelyet majd az ablak kezelésére használjuk később.   
**„root.title("Sudoku Solver")”**: Ezzel a sorral beállítjuk az ablakunk címét a **„Sudoku Solver”** szövegre.

1. ****

Itt az **„entries”** listát fogjuk létrehozni, ami tartalmazni fogja az összes beviteli mezőt, amelyek a sudoku táblát ábrázolja a grafikus felhasználói felületen. A felhasználó számokat írhat be ezekbe a mezőkbe a sudoku feladvány megoldásához.  
**„entries = []”**: Egy üres lista létrehozása, amelyben tárolni fogjuk a **„tk.Entry”** objektumokat, vagyis azokat a mezőket, amelyek az egyes cellákat ábrázoljuk majd a Sudoku táblán.  
**„for i in range(9): és for j in range(9):”**: Itt hozzuk létre a beviteli mezőket. Két egymásba ágyazott **„for”** ciklussal létrehozunk egy 9x9-es rácsot, amely ábrázolja a Sudoku táblánkat.   
Minden egyes iteráció során létrehozunk egy új **„tk.Entry”** objektumot az aktuális cella számára a **„root”** ablakban.   
Beállítjuk a beviteli mező szélességét 4 karakterre a **„width=4”** -el.   
Az **„entry.grid(row=i, column=j)”** metódussal a beviteli mezőt a megfelelő sorba illetve oszlopba helyezzük el a **„root”** ablakban.  
**„row.append(entry)** és **entries.append(row)”**: Ezekkel a sorokkal adjuk hozzá a beviteli mezőket. Minden egyes beviteli mezőt hozzáadjuk az aktuális sorhoz a **„row.append(entry)”** sorral. Az „**entries.append(row)”** sorral pedig hozzáadjuk az aktuális sort az **„entries”** listához, amely tartalmazni fogja az egész Sudoku táblát.

1. 

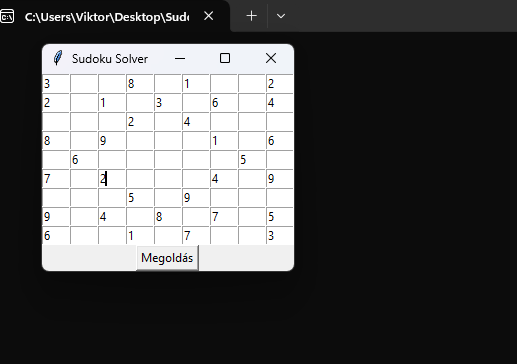
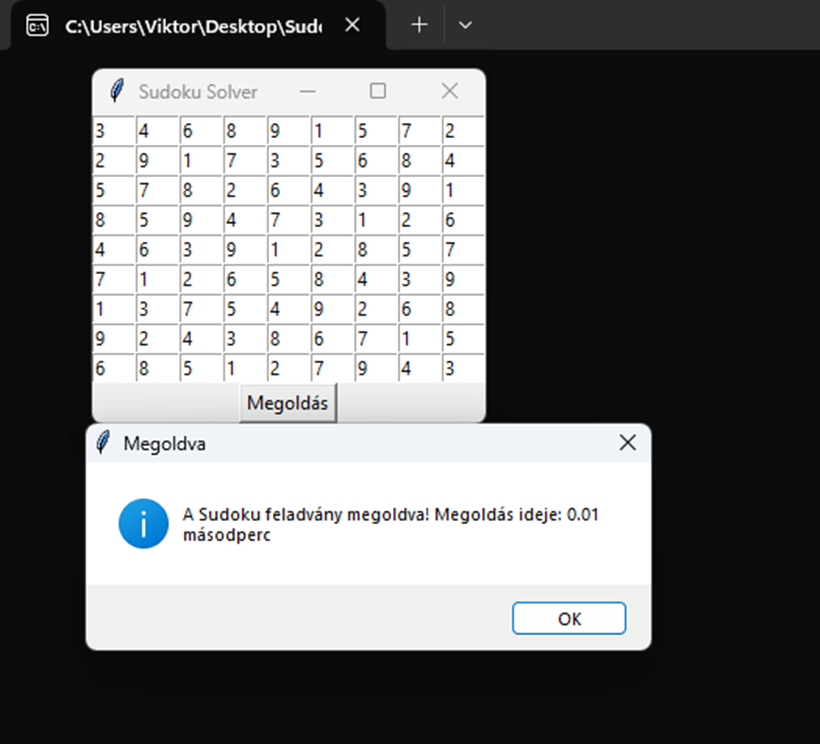
**solve\_button = tk.Button(root, text="Megoldás", command=solve\_sudoku)**: Ezzel a sorral fogjuk létrehozni a gombunkat. Létrehozunk egy új **„tk.Button”** objektumot, amely a **"Megoldás"** felirattal rendelkezik. A gombra kattintva a **„solve\_sudoku”** függvény lesz meghívva, mivel ezt adjuk meg a **„command”** argumentumban.  
**„solve\_button.grid(row=9, columnspan=9)”**: Ezzel a paranccsal pedig elhelyezzük a gombunkat. A **„grid(row=9, columnspan=9)”** metódussal elhelyezzük a gombot a rácsban. A gomb a 9. sorban lesz elhelyezve. A **„columnspan=9”** argumentummal az oszlopszélességet a teljes rács szélességére állítjuk, hogy a gomb teljesen kitöltse a rácsot.

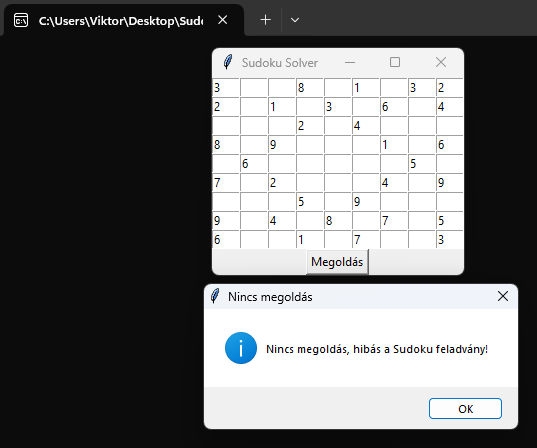
1. 

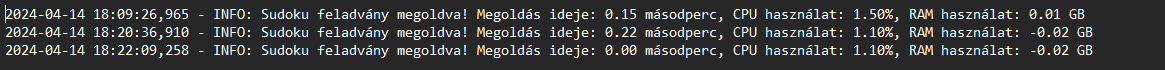
**root.mainloop()**: Ez a függvény addig fut, amíg az ablak bezárására nem kerül sor, és folyamatosan figyeli és kezeli a felhasználói interakciókat.

## A program működés közben:

A program indítás után egy 9x9-es sudoku tábla fogadja a delikvenst.

* Itt tudjuk megadni a kezdő számokat a programnak.   
    
   
* Töltsük ki a kezdő számainkkal a sudoku táblánkat:   
    
  
* Amint megvagyunk a kezdő számok kitöltésével nincs más dolgunk, mint a Megoldás gombra kattintani.
* Amennyiben helyesen adtuk meg a feladványt a programnak akkor visszakapjuk a megoldást, illetve kapunk egy felugró ablakot a megoldás sikerességéről.   
    
  
* Viszont, ha egy hibás feladványt adunk a programunknak akkor egy felugró ablak jelzi nekünk, hogy „Nincs megoldás, hibás a Sudoku feladvány!”



* A program minden esetben készít egy log fájlt miután lefuttattuk a programunkat. Ennek az elérési útja: \Sudoku külön futtatható program\\_internal\sudoku\_solver.txt
* Itt pedig a log fájlból láthatunk egy részletet:

## Felhasznált források:

<https://hu.wikipedia.org/wiki/Visszal%C3%A9p%C3%A9ses_keres%C3%A9s>  
<https://hu.wikipedia.org/wiki/Rekurzi%C3%B3>  
<https://hu.wikipedia.org/wiki/Grafikus_felhaszn%C3%A1l%C3%B3i_fel%C3%BClet>  
<https://csharptutorial.hu/docs/hellovilag-hellocsharp/8-nyelvi-szolgaltatasok/a-tuple-tipus/>  
<https://hu.wikipedia.org/wiki/Iter%C3%A1ci%C3%B3>  
<https://www.informatika-programozas.hu/informatika_java_programozas_gyakorlat_2_matrix.html>  
<https://hu.wikipedia.org/wiki/Lebeg%C5%91pontos_sz%C3%A1m%C3%A1br%C3%A1zol%C3%A1s>

# Sudoku megoldó program Python nyelven, PyCharm környezetben Oláh Adrián – CW6W8K

## A Sudoku szabályai, felépítése:

A Sudoku egy logikai játék, amelyben egy 9x9-es rácsot kell kitölteni úgy, hogy minden sorban, oszlopban és 3x3-as négyzetben mindegyik szám 1-től 9-ig pontosan egyszer szerepeljen. A kezdeti elrendezés néhány számmal indul, ezek a "hintek". A játék célja az összes üres cella kipótlása a szabályoknak megfelelően, anélkül hogy ugyanazt a számot többször használnánk ugyanabban a sorban, oszlopban vagy 3x3-as négyzetben. A helyes megoldás egyértelmű és egyedülálló. A játék népszerűsége a logikai kihívásából és az egyszerű szabályokból ered, amelyek könnyen érthetők, de egyre nehezebbé teszik a feladványokat.

A programról néhány szóban:

A Sudoku megoldó applikáció fő célja hogy, segítse a felhasználókat egy Sudoku feladvány gyors megoldásában, az applikáció a rekurzív megoldásra épül, röviden és tömören összefoglalva, a megadott lehetőségek közül, minden cellában, minden elérhető lehetőséget megpróbál, míg nem talál helyes megoldást, vagy nem ütközik hibába. Az applikáció a megoldó képlet mellett kapott néhány kiegészitő funkciót is. Az egyik ilyen funkció az új tábla kérésének lehetősége, így a felhasználónak nem szükséges a program bezárására és újbóli futtatására, egyszerűen egy gombnyomásra új táblát nyithat. Emellett két megfigyelő kiegészítést is kapott, az egyik a folyamatos figyelés, ami monitorozza a pillanatnyi erőforrás felhasználást. Illetve a másik megfigyelő kiegészítés, ami a Megoldás gombra való kattintástól egészen a megoldás lefutása végéig figyeli az eltelt időt, a RAM és CPU erőforrások felhasználását.

## 2. A Program felépítése (GUI, Sudoku megoldás, hibakeresés)

import pygame  
import sys  
import time  
import psutil  
import tkinter as tk  
from tkinter import messagebox  
import random

Első körben szükségünk lessz néhány addonra hogy, a programunk működőképes legyen.  
“pygame” maga a felület amin a programunk futni fog.  
“sys” segítséget nyújt a rendszerrel kapcsolatos műveletekhez  
“psutil” Az erőforrás monitorozásáért felelős addonunk.  
“tkinter” A tkinter fog nekünk segíteni abban hogy, a program lefutása végén egy előugró ablakban láthassuk majd a program statisztikáit.  
“import” random véletlenszerű számok importálásáért felelős.

# Az üres tábla kezdeti állapota  
board = [  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0]  
]

Az üres táblánk kezdeti állapota, ami annyit tesz hogy, a 0-ák az üres mezőket jelölik.

# A tábla érvényességének ellenőrzése  
def valid\_board(board):  
 for i in range(9):  
 for j in range(9):  
 if board[i][j] != 0:  
 if not is\_valid(board, board[i][j], (i, j)):  
 return False  
 return True

Itt a táblánk érvényességét ellenőrizzük a Sudoku szabályainak megfelelően. A kódsorunk az összes elérhető cellát megvizsgálja, amennyiben egy cella nem üres, abban az esetben ellenőrzi hogy, az ott lévő szám helyesen van-e elhelyezve a táblán. Amennyiben egy szám nem a Sudoku szabályainak megfelelően van elhelyezve a táblánkon, abban az esetben a kód False étékkel tér vissza, minden más esetben az érték True.

# Fő funkció: a sudoku megoldása  
def solve\_sudoku(grid):  
 find = find\_empty(grid)  
 if not find:  
 return True  
 else:  
 row, col = find  
  
 for num in range(1, 10):  
 if is\_valid(grid, num, (row, col)):  
 grid[row][col] = num  
  
 if solve\_sudoku(grid):  
 return True  
  
 grid[row][col] = 0  
  
 return False

Ez a kód egy algoritmust valósít meg a Sudoku rejtvény megoldására. Az első lépés az üres cellák keresése és azok sorban való kitöltése. Az algoritmus próbálkozik egy számmal, viszont amennyiben ez nem vezet megoldáshoz, visszalép, és egy másik számmal próbálkozik, mígnem teljes megoldást talál vagy kimerül az összes lehetőség.

# Az üres helyek keresése a táblán  
def find\_empty(grid):  
 for i in range(len(grid)):  
 for j in range(len(grid[0])):  
 if grid[i][j] == 0:  
 return (i, j)  
 return None

Ez a kód egy egyszerű függvény, ami arra szolgál, hogy megtaláljuk azokat a helyeket a táblán, ahol nincs szám. Vagyis körbejárja a táblát, és ha talál egy olyan helyet, ahol nincs szám, akkor vissza adja az üres mező koordinátáit. Vsziont, amennyiben minden helyen van már szám, akkor semmit sem ad vissza.

# Adott szám ellenőrzése egy adott helyen  
def is\_valid(grid, num, pos):  
 # Sor ellenőrzése  
 for i in range(len(grid[0])):  
 if grid[pos[0]][i] == num and pos[1] != i:  
 return False

Itt egy egyszerű ellenőrzést hajtunk végre egy adott számról egy adott helyen a Sudoku táblánkon. A függvény megvizsgálja az adott sor minden celláját és ha talál egy azonos számot, amely nem ugyanabban a cellában van, akkor visszatér hamis értékkel.

# Oszlop ellenőrzése  
for i in range(len(grid)):  
 if grid[i][pos[1]] == num and pos[0] != i:  
 return False

Itt vizsgáljuk meg, hogy az adott szám megfelelő-e az oszlopban az adott cella pozíciójához képest a Sudoku táblánkon. Ha talál azonos számot az oszlopban, kivéve az aktuális cellát, akkor hamis értékkel tér vissza.

# Négyzeten belüli ellenőrzés  
box\_x = pos[1] // 3  
box\_y = pos[0] // 3  
  
for i in range(box\_y \* 3, box\_y \* 3 + 3):  
 for j in range(box\_x \* 3, box\_x \* 3 + 3):  
 if grid[i][j] == num and (i, j) != pos:  
 return False  
  
return True

Ez a rész azért felelős, hogy ellenőrizze, hogy az adott szám megfelelő-e a kis 3x3-as négyzetben az adott cella pozíciójához képest a Sudoku táblán. Először meghatározza, hogy az adott cella melyik kis 3x3-as négyzetben található, majd végigmegy ennek a négyzetnek a celláin, hogy ellenőrizze, hogy az azonos számú másik cella nem található-e ugyanabban a kis négyzetben. Ha talál ilyen számot, akkor hamis értékkel tér vissza, máskülönben valós értékkel.

Vízszintes vonalak  
for i in range(10):  
 if i % 3 == 0:  
 pygame.draw.line(window, (0, 0, 0), (50, 50 + 50 \* i), (500, 50 + 50 \* i), 4)  
 else:  
 pygame.draw.line(window, (0, 0, 0), (50, 50 + 50 \* i), (500, 50 + 50 \* i), 2)  
  
# Függőleges vonalak  
for i in range(10):  
 if i % 3 == 0:  
 pygame.draw.line(window, (0, 0, 0), (50 + 50 \* i, 50), (50 + 50 \* i, 500), 4)  
 else:  
 pygame.draw.line(window, (0, 0, 0), (50 + 50 \* i, 50), (50 + 50 \* i, 500), 2)  
  
# Számok kirajzolása  
font = pygame.font.Font(None, 36)  
for i in range(len(grid)):  
 for j in range(len(grid[0])):  
 if grid[i][j] != 0:  
 text\_surface = font.render(str(grid[i][j]), True, (0, 0, 0))  
 text\_rect = text\_surface.get\_rect(center=(j \* 50 + 75, i \* 50 + 75))  
 window.blit(text\_surface, text\_rect)

Ezt a kódot a táblánk kirajzolására használjuk a “pygame” addon segítségével. Első körben függőleges és vízszintes vonalakat rajzolunk, majd a tábla számainak megfelelő pozíciókba való elhelyezését végzi el a tábla számait tartalmazó ,,rács” alapján.

# Jelölés a kiválasztott mezőben  
if selected\_row is not None and selected\_col is not None and 0 <= selected\_row < 9 and 0 <= selected\_col < 9:  
 pygame.draw.rect(window, (255, 0, 0), (50 + selected\_col \* 50, 50 + selected\_row \* 50, 50, 50), 3)

Ez a rész egy piros keretet rajzol a kiválasztott cella köré miután rákattintottunk. Amennyiben a megfelelő tartományon belül vannak.

# Megoldás gomb  
pygame.draw.rect(window, (0, 255, 0), solve\_button)  
text\_surface = font.render("Megoldás", True, (0, 0, 0))  
text\_rect = text\_surface.get\_rect(center=solve\_button.center)  
window.blit(text\_surface, text\_rect)

Ezzel a kóddal egy Megoldás feliratú gombot hozunk részre a táblánk alján, zöld háttérrel és fekete “Megoldás” felirattal, a gomb közepére helyezve.

# Új gomb  
pygame.draw.rect(window, (255, 255, 0), new\_button)  
text\_surface = font.render("Új", True, (0, 0, 0))  
text\_rect = text\_surface.get\_rect(center=new\_button.center)  
window.blit(text\_surface, text\_rect)

Ezzel a kóddal egy Új feliratú gombot hozunk létre a táblánk alján a Megoldás gombtól jobbra, sárga háttérrel és fekete “Új” felirattal.

# App RAM és App CPU kiírása  
text\_surface = font.render(f"App RAM: {app\_ram:.2f} MB", True, (0, 0, 0))  
text\_rect = text\_surface.get\_rect(topright=(750, 50))  
window.blit(text\_surface, text\_rect)  
  
text\_surface = font.render(f"App CPU: {app\_cpu:.2f}%", True, (0, 0, 0))  
text\_rect = text\_surface.get\_rect(topright=(750, 100))  
window.blit(text\_surface, text\_rect)

Ez a rész megjeleníti az alkalmazás aktuális RAM és CPU felhasználását a képernyő jobb felső sarkában. Az aktuális RAM-felhasználásot a "App RAM" felirattal, míg a CPU-felhasználást az "App CPU" felirattal jeleníti meg. Az értékek megjelenítésekor mindkét esetben a megfelelő egységgel jelöljük, Ram-MB Cpu-%.

# Leírás  
description\_font = pygame.font.Font(None, 24)  
description\_text = "Kérem írja be a Sudoku feladványt, majd kattintson a Megoldás gombra!"  
description\_surface = description\_font.render(description\_text, True, (0, 0, 0))  
description\_rect = description\_surface.get\_rect(midright=(680, solve\_button.bottom + 10))  
window.blit(description\_surface, description\_rect)  
  
pygame.display.update()

Ezzel a kóddal egy feliratot hozunk létre a táblánk alján a következő szöveggel “Kérem íra be a Sudoku feladványt, majd kattintson a Megoldás gombra!””. A szöveget a solve, azaz Megoldás gomb a kiindulási pontja, azaz onnan adjuk meg a szöveg pontos pozícióját.

## 3. Fő program (Definíciók, Live Monitoring, Érzékelők,Logikai változók)

A különböző kiegészítő kódok után, elérkeztünk a program indítása utána müveletekhez, itt a kód hosszúsága miatt próbáltam több részre lebontani a kódot hogy, könnyebben emészthető legyen.

def main():  
 pygame.init()  
 window = pygame.display.set\_mode((800, 600))  
 pygame.display.set\_caption("Sudoku Solver")

Első részünk, itt inicializáljuk a pygame addonunkat első körben, majd egy 800x600 pixelből álló ablakot hozunk létre “Sudoku Solver” címmel.

solve\_button = pygame.Rect(150, 520, 200, 50) # Megoldás gomb definiálása  
new\_button = pygame.Rect(450, 520, 200, 50) # Új gomb definiálása

Itt két gombot definiálunk pygameben. Az első gomb a "Megoldás" gomb, amelynek a pozíciója (150, 520) és a mérete 200x50 pixel. A második gomb az "Új" gomb, amelynek a pozíciója (450, 520) aminek a mérete szintén 200x50 pixel.

draw\_board(window, board, selected\_row, selected\_col, solve\_button, new\_button, 0, 0

Ezzel a rész egy rajzoló függvényt hívunk, amimegjeleníti a Sudoku táblánkat és annak összes elemét a megadott ablakban. A függvény a következőket tartalmazza: a Sudoku tábla, a jelenleg kiválasztott sor és oszlop, a "Megoldás" és "Új" gombok, valamint a RAM és CPU kiírásának helye az ablakon. A RAM és CPU értékei ennél a kódnál nincsennek használatban.

# Az App CPU és RAM frissítése 0.5 másodpercenként  
update\_interval = 0.5  
last\_update\_time = time.time()

Itt a CPU és RAM frissitési intervallumát állítjuk be hogy, mindíg az aktuális értékeket kapjuk a CPU és RAM felhasználásáról, a frissitési időnk 0.5 másodperc, ami a last update time = time.time() inditásától indul.

new\_board\_requested = False # Új tábla igénylésének jelzője

A kódunk ezen része figyeli hogy a felhasználó szeretne-e új táblát igényelni. Amíg a felhasználó nem igényel új táblát addig az értékük False, azaz hamis.

while True:  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()

A kódunk végtelenített ciklusa az események megfigyelésére a pygame addonunkban. A kód figyeli hogy a felhasználó bezárta-e az alkalmazás ablakát. Amennyiben igen, abban az esetben a kód bezárja a pygame addont és kilép a rendszerből.

if event.type == pygame.MOUSEBUTTONDOWN:  
 mouse\_pos = pygame.mouse.get\_pos()  
 selected\_col = (mouse\_pos[0] - 50) // 50  
 selected\_row = (mouse\_pos[1] - 50) // 50

Ezzel a kóddal figyeljük hogy, a felhasználó kattintott-e az egérrel, amennyiben igen. Abban az esetben a kódunk lekéri az egérmutató pozícióját az alkalmáson belül a Sudoku táblánkon való elhelyezkedése alapján. Így lehetővé téve hogy a felhasználó ki tudja választani az adott cellákat a Sudoku táblánkon belül.

if solve\_button.collidepoint(mouse\_pos):  
 if valid\_board(board):  
 start\_time = time.time() # Az időmérés kezdete  
 solve\_sudoku(board)  
 end\_time = time.time() # Az időmérés vége

A kódunk következő része figyeli hogy, a felhasználó a Megoldás gombra kattintott-e amennyiben igen, a kód elindít egy időmérést ami a solve\_sudoku(board) azaz, a Sudoku feladvány megoldásáig tart. Ez a live monitoringhoz szükséges, ugyanis a programunktól a Sudoku feladvány megoldása után adni fog nekünk néhány értéket.

# App és CPU RAM kiírása  
process = psutil.Process()  
app\_ram = process.memory\_info().rss / (1024 \* 1024)  
app\_cpu = process.cpu\_percent()

Itt figyeljük a RAM és CPU pillanatnyi értékét a psutil addon segítségével. A RAM felhasználását az alkalmazás futási információból számolja ki MB-ban megadva, míg a CPU-t az alkalmazás futási processzor terheléséből kapjuk meg százalékos arányban.

## 4.Eredmények megjelnítése, tkinter(felugró ablakok), szám bevitel billentyűparanccsal,Live monitoring GUI, új tábla igénylése üres mezőkkel

# Eredmények kiírása a képernyőre  
total\_time = end\_time - start\_time # Az összes eltelt idő  
message = f"Megoldás ideje: {total\_time:.2f} másodperc\n" \  
 f"App CPU felhasználás: {app\_cpu:.2f}%\n" \  
 f"App RAM felhasználás: {app\_ram:.2f} MB"  
tk.Tk().withdraw() # Csak az üzenetdoboz mutatása, nem egy teljes ablak  
messagebox.showinfo("Eredmények", message)

A kódunk kiszámolja a Sudoku feladvány megoldási idejét, illetve a CPU és RAM felhasznált erőforrásainak mennyiségét, majd a program lefutása után egy felugró ablakban tájékoztatja a felhasználót az adatokról.

else:  
 tk.Tk().withdraw() # Csak az üzenetdoboz mutatása, nem egy teljes ablak  
 messagebox.showinfo("Hiba", "Hibás feladvány!")

Ebben az esetben viszon, nem kapunk adatokat a megoldási időről és felhasznált erőforrásokról. Ez a kód ugyanis akkor lép érvényben, ha a megoldani próbált feladványunkhibás. Ebben az esetben egy másik felugró ablak fogadja a felhasználót, ami“Hibás felavány!” kiírásával figyelmezteti a felhasználót hogy, a megoldani próbált feladvány hibás.

elif new\_button.collidepoint(mouse\_pos):  
 new\_board\_requested = True # Új tábla igénylésének jelzője

Amennyiben megoldottuk a feladványt, vagy hibás feladványt adtunk meg. Lehetőségünk van új táblát igényelni, ez a kód figyeli hogy, az egérmutató az “Új” gombon belül van-e, történt-e kattintás. Amennyiben történt, abban az esetben az alkalmazás létrehoz nekünk egy új táblát.

elif 0 <= selected\_row < 9 and 0 <= selected\_col < 9:  
 draw\_board(window, board, selected\_row, selected\_col, solve\_button, new\_button, 0, 0)

Ez a rész ellenőrzi, hogy a kiválasztott sor és oszlop értékei megfelelő tartományon belül vannak-e. Amennyiben igen, akkor meghívja a draw\_board függvényt a jelenlegi ablakkal, a Sudoku táblával, a kiválasztott sorral és oszloppal, valamint a "Megoldás" és "Új" gombokkal. Így kapunk egy új táblát.

if event.type == pygame.KEYDOWN:  
 if selected\_row is not None and selected\_col is not None:  
 if event.unicode.isdigit() and 1 <= int(event.unicode) <= 9:  
 board[selected\_row][selected\_col] = int(event.unicode)  
 draw\_board(window, board, selected\_row, selected\_col, solve\_button, new\_button, 0, 0)

A kód következő része figyeli hogy, történt-e billentyű leütés az alkalmazásban. Amennyiben történt és az érték 1 és 9 közt van, a megfelelő cellában. Abban az esetben az alkalmazás beírja a kiválasztott számot. Így a felhasználó könnyedén be tudja írni a megoldanó Sudoku feladványt.

# Ha az új tábla igénylője igaz, akkor azonnal generálódjon új tábla  
if new\_board\_requested:  
 generate\_new\_board()  
 draw\_board(window, board, selected\_row, selected\_col, solve\_button, new\_button, 0, 0)  
 new\_board\_requested = False # Visszaállítjuk az új tábla igénylő jelzőt

Ez a kód felelős az új tábla igényléséért, itt figyeljük hogy történt-e új tábla igénylés. Amennyiben igen, abban az esetben végig fut a kódunk és kapunk egy új táblát, majd a kód végén ismét visszaáll False azaz hamis értékse, magyarán nem igényel ismételten új táblát.

# App CPU és RAM frissítése 0.5 másodpercenként  
if time.time() - last\_update\_time >= update\_interval:  
 process = psutil.Process()  
 app\_ram = process.memory\_info().rss / (1024 \* 1024)  
 app\_cpu = process.cpu\_percent()  
 draw\_board(window, board, selected\_row, selected\_col, solve\_button, new\_button, app\_ram, app\_cpu)  
 last\_update\_time = time.time()

Itt történik a futó programunkon belül az erőforrások figyelése, ami ugye 0.5 másodpercenként történik, amennyiben változás történt az értékekben, abban az esetben a kód frissíti a látható értékeket. Így folyamatos képet kapunk az applikációnk folyamatos erőforrás felhasználásáról.

# Új tábla generálása  
def generate\_new\_board():  
 global board  
 # Minden mezőt üresre állítunk  
 board = [[0 for \_ in range(9)] for \_ in range(9)]

A kódunk következő része szintén az új tábla genereálására szolgál, itt viszont maga a Sudoku játéktérről beszélünk, a kód az új táblánk minden mezőjét 0-ára azaz üresre állítja. Így lehetőségünk van egy új feladványt megadni.

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

Itt ellenőrizzük hogy a fájlt közvetlenül futtatják-e, esetleg egy másik fájlba importálták. Amennyiben a fájlt közvetlenül futtatják, abban az esetben meghívja a main függvényt, ami elindítja az alkalmazásunkat.

## 5. Felhasznált források

<https://chat.openai.com/>

<https://www.youtube.com/watch?v=PZJ5mjQyxR8>

https://www.geeksforgeeks.org/building-and-visualizing-sudoku-game-using-pygame/

# Az alkalmazott módszerek összehasonlítása OAM elemzés segítségével

Több különböző Sudoku solver (pl. Go program és Excel VBA) futtatási eredményeit hasonítjuk össze. A sikeres OAM elemzéshez először definiálni kell az objektumokat és attribútumokat, majd ezek alapján elemzést végezni.

## Objektumok:

Go program

Excel VBA

Excel Solver

## Attribútumok:

Algoritmus típusa (backtrack, Solver)

Megoldási terület mérete („4×4”; „6×6”; „9×9")

Feladat mérete (különböző nehézségű Sudoku feladványok)

Futási idő sec-ben

Memória használat byte-ban (az erőforrás-igény)

Sikerült-e megoldani (megoldás sikeressége)

**Az összehasonlítás részletei és eredménye a mellékelt Excel-táblázatban érhető el („Sudoku solver OAM excel.xlsx”).**

# Tantárgyi kapcsolatok

## LIN029 - Operációs rendszerek

Az operációs rendszerek kezelik az erőforrásokat (memória, processzoridő), amelyek létfontosságúak a Sudoku solver futtatásához. A folyamatok és szálak kezelése, amit az operációs rendszerek tantárgy tárgyal, közvetlenül befolyásolja az Excel programok, így a Sudoku solver teljesítményét és stabilitását. Emellett az operációs rendszerek hibakezelési mechanizmusai alapvetőek a Sudoku megoldóprogramok zökkenőmentes működéséhez. Röviden, az "Operációs rendszerek" elvei nélkülözhetetlenek a hatékony és megbízható Sudoku megoldó programok fejlesztéséhez Excel környezetben.

## LIN026 - Matematikai alapok

Az Excelben készült Sudoku megoldóprogram és a "Matematikai alapok" tantárgy közötti kapcsolat abban rejlik, hogy a megoldó algoritmusok tervezésekor és implementálásakor alapvető matematikai logikát és algoritmusokat, mint például kombinatorikát, logikai műveleteket és esetlegesen gráfelméletet használunk. A tantárgy által nyújtott matematikai gondolkodásmód és problémamegoldó készségek közvetlenül segítik a fejlesztőt a Sudoku rejtvények hatékony megoldási stratégiáinak kialakításában.

## LSK001 - Kultúra, sport, munkahelyi jóllét 1.

A Sudoku megfejtése javítja a logikai gondolkodást és a koncentrációt, amelyek hozzájárulnak az általános mentális egészséghez és jólléthez, ezzel összhangban állva a tantárgy által képviselt értékekkel. Ez a kapcsolat kiemeli, hogy a szellemi kihívást jelentő tevékenységek, mint a Sudoku megoldása, pozitív hatással lehetnek a munkahelyi és személyes jóllétre.

## LKN490 - Európai civilizáció és identitás

A Sudoku, mint logikai játék, és annak megoldására szolgáló programok fejlesztése tükrözheti az európai kultúrában gyökerező értékeket, mint a tudományos gondolkodás, az innováció és az intellektuális kihívások iránti elkötelezettség. Ez a kapcsolat bemutatja, hogyan hozhatnak össze különböző területeket a kulturális hagyományok és a modern technológiai megoldások.

## LIN028 - Hálózatok és számítógép architektúrák

Az Excelben készült Sudoku megoldóprogram és a tantárgy közötti kapcsolat abban rejlik, hogy a program fejlesztése és futtatása során kihasználja a számítógép architektúráját, mint a processzor teljesítményét és a memória kezelését.

## LIN027 - Adatszerkezetek és algoritmusok

A program alapját képező logikai algoritmusok, mint a backtracking és a megoldási stratégiák, közvetlenül kapcsolódnak az adatszerkezetek és algoritmusok tanulmányozásához, bemutatva, hogy ezek a fogalmak hogyan alkalmazhatók valós problémák megoldására.

## LNE178 - A jog szerepe a modern társadalmakban

A program fejlesztése során felmerülő jogi kérdések, mint a szoftver licenszelése, a felhasználási feltételek meghatározása és a szellemi alkotások jogi védelme, mind fontos elemek, amelyek bemutatják a jog szerepét a technológia és innováció világában.

# Összegzés

A választás a Sudoku megoldó program kialakításakor a különböző programnyelvek között függ a felhasználó vagy fejlesztő preferenciáitól, a rendelkezésre álló időtől, és attól, hogy mennyire bonyolult a megoldandó Sudoku rejtvény. Míg az Excel Solver egy gyors és egyszerű megoldást kínál kevésbé összetett feladványok esetén minimalizált fejlesztési erőfeszítéssel, addig a backtrack algoritmus lehetővé teszi mélyrehatóbb testreszabást és általában jobb teljesítményt nyújt bonyolultabb rejtvényeknél, cserébe a nagyobb fejlesztési idő és a szükséges programozási ismeretek miatt.

* Go program: Általánosságban kevesebb erőforrást használ, és gyorsabban fut, de vannak esetek, amikor nem sikerül megoldania a feladványt.
* Excel VBA: Nagyobb erőforrásigényű, és lassabb futási idejű, de magas sikerességi rátát mutat.

# Summary

The choice between different programming languages when designing a Sudoku solver program depends on the user or developer's preferences, the available time, and the complexity of the Sudoku puzzle to be solved. While Excel Solver offers a quick and simple solution for less complex puzzles with minimized development effort, the backtrack algorithm allows for more in-depth customization and generally provides better performance for more complex puzzles, in exchange for greater development time and the need for programming knowledge.

* Go program: Generally uses fewer resources and runs faster, but there are cases where it fails to solve the puzzle.
* Excel VBA: More resource-intensive and has a slower runtime, but shows a high success rate.

# Rövidítésjegyzék, hivatkozásjegyzék, irodalomjegyzék

## Csötönyi Viktória\_J908Q3:

Források:

Web1.:<https://letsdopuzzles.com/dl/4x4-sudoku-puzzles-set-3>

Web 2..:https://miau.my-x.hu/miau2009/index.php3?x=e0&string=sudoku

Pitlik László és Pitlik Marcell, „SUDOKU-algoritmusok összevetése, avagy SOLVER vs. egyedi kód” (2019)

Web 2.:<https://letsdopuzzles.com/dl/6x6-sudoku-puzzles-1>

# Mellékletek

1. https://hu.wikipedia.org/w/index.php?title=Sz%C3%BAdoku&section=3#Hagyom%C3%A1nyos\_sz%C3%BAdokuk [↑](#footnote-ref-1)