Data Structures and Objects
CSIS 3700

Spring Semester 2025 — CRN 20796

Project 2 — Sudoku Solver
Due date: Monday, March 3, 2025

Goal

Develop and implement a stack-based hexadecimal Sudoku puzzle solver.

Important

You must follow the algorithms in this document. Do not use other algorithms. You must also use our stack implementation. Do not use the STL for anything in this project.

Details

You are most likely familiar with the Sudoku puzzle game. It consists of a 9-by-9 grid; initially, some of the positions are filled with numbers and others are blank.

When solved, each row must have all of the integers 1 – 9, each column must have all of the integers 1 – 9 and each of the nine 3-by-3 blocks must have all of the integers 1 – 9.

In this project, we will extend the concept to a 16-by-16 board with each row, column and the sixteen 4-by-4 blocks each having all of the hexadecimal digits 0 — F.

One method to solve a Sudoku puzzle is trial-and-error. If a valid guess can be made, make it and repeat. If no valid guess can be made, go back to the previous guess and change it; if no other guesses can be made, go back to the guess before that. Continue until either all boxes are filled or all guesses are exhausted (which shouldn't happen because that means there is no solution.)

Create a stack of integers which represent the location of cells being filled in. The top location on the stack represents the cell currently being filled in. Note that there will be at most 256 locations to track.

Note: If you're clever, you only need one integer to represent the location, not two.

Read the data from **`cin`**. Input consists of sixteen lines of sixteen characters. If a cell is filled in, its character will be a hexadecimal digit 0 – F. If it is blank, its character is a period.

Pro tip: You only need a single **`char`** variable for the input; no strings necessary.

Once the data is read, use the following algorithm to solve the puzzle.

Algorithm 1 The main Sudoku algorithm

Precondition: *board* contains an unsolved Sudoku puzzle
Postcondition: *board* contains a solved Sudoku puzzle, or no solution exists

```
procedure Solve
    Select the best empty cell and place its location on the stack

    while true do
        Let (i, j) be the location on top of the stack

        Select the next valid choice for board[i][j]
        if no such choice exists then
            Mark board[i][j] as not filled in
            Pop the stack
            if the stack is empty then
                Return; the puzzle has no solution
            end if
            continue
        end if

        Select the best empty cell and place its location on the stack
        if no such cell exists then
            break                                          ▷ Puzzle is now solved
        end if
    end while

    Output the solution
end procedure
```

Keeping track of choices

You should use bit manipulation to keep track of information for each cell. All of the information for one cell can be kept in 21 bits:

- One bit to indicate if the cell has been filled in
- Sixteen bits to keep track of which digits you are allowed to place in the cell
- Four bits to hold the current choice for the cell

You'll want to use the masking operations at various points in the program to turn bits on and turn them off. You'll also want to use the left shift operation to look at the valid choices for a cell.

Selecting the best empty cell

Hypothetically, you can pick any empty cell for your next choice. However, to minimize the work the computer performs in backtracking, there is a preferred cell. The best cell to choose has the fewest valid choices for its digit.

The following algorithm selects the best empty cell and places its location on the stack.

Algorithm 2 Finding the best empty cell

Precondition: *board* contains an unsolved Sudoku puzzle
Postcondition: the best location is pushed onto the stack
the best location is marked as filled in

```
procedure FindBest
    for each empty cell board[i][j] do          ▷ Initialize to allow all digits as choices
        Mark all digits as valid choices
    end for

    for each filled in cell board[i][j] do      ▷ Remove invalid choices
        for each unfilled cell in row i do
            Mark digit in board[i][j] as an invalid choice
        end for
        for each unfilled cell in column j do
            Mark digit in board[i][j] as an invalid choice
        end for
        for each unfilled cell in the 4x4 block containing board[i][j] do
            Mark digit in board[i][j] as an invalid choice
        end for
    end for

    Set low ← 17
    for each empty cell board[i][j] do
        Count 1-bits in valid choices for board[i][j]
        if count < low then
            low ← count
            i_best ← i
            j_best ← j
        end if
    end for

    if low = 17 then
        return false                            ▷ No empty cells remain
    end if

    Mark board[i_best][j_best] as filled in
    Push (i_best, j_best) onto the stack

    return true
end procedure
```

What to turn in

Turn in your source code and **`Makefile`**. If you are using an IDE, compress the folder containing the project and submit that.

Example 1

Input

```
b2.7...f6e...3.d
...9b.2.....f1.6
.a.c..7.4.8...e.
.8.3..1...fca.42
..e.....16a...3.
3........4.f8.6b
...d2....b..74..
.b840..d7.93e...
...f37.45..90dc.
..c8..a....b4...
49.5e.d........a
.d...6c1.....7..
91.e74...f..3.d.
.3...9.2.1..5.0.
f.4b.....3.d1...
5.d...fe9...2.b4
```

Output

```
Solution: b257 4a9f 6e01 c38d
          e409 b82c d53a f176
          1afc 6d73 4982 b5e0
          d863 5e10 b7fc a942

          07e2 8b49 16a5 dc3f
          3591 ace7 24df 806b
          cfad 2136 8be0 7459
          6b84 0f5d 7c93 e2a1

          2e1f 37b4 5a69 0dc8
          76c8 92a5 0d1b 4ef3
          4935 e0d8 f2c7 6b1a
          adb0 f6c1 384e 9725

          912e 740b af56 38dc
          837a d962 c1b4 5f0e
          f04b c58a e32d 1697
          5cd6 13fe 9078 2ab4
```

Example 2

Input

```
.23.5.d.0.e...f.
.af........6...b
6..8.af..9..42.5
.5......2a8f..0.
298d6..1...7e..0
f4.5.80.6e1b...9
..7b.d.f5..0....
3e.04....2..f.b.
.3.4..9....d8.62
....d..6a.2.b7..
b...f014.c9.3.ae
e..f8...7..319c4
.f..716c......2.
7.d6..5..f4.c..8
8...e........31.
.c...2.0.5.a.f4.
```

Output

```
Solution: 4237 56d9 0be1 a8fc
          0afe 23c8 4d56 719b
          61b8 0afe 397c 42d5
          d59c 14b7 2a8f 6e03

          298d 6ba1 f4c7 e530
          f4a5 3802 6e1b dc79
          c67b 9def 53a0 2481
          3e10 4c75 92d8 f6ba

          a354 c79b e1fd 8062
          98c1 de36 a024 b75f
          b762 f014 8c95 3dae
          ed0f 852a 76b3 19c4

          5f49 716c b83e 0a2d
          70d6 a953 1f42 cbe8
          8b2a ef4d c709 5316
          1ce3 b280 d56a 9f47
```

Example 3

Input

```
274c..b.......3.
ea..3.9.....6.c0
...9.8.cd..1..57
...16.....b9e8d2
a5.....e7..4.1.9
........1.a.5f..
f.6...3a58.b...4
8b.e..d4f.0....a
1....4.d07..9.bc
0...b.7246...5.1
..da.6.9........
b.c.0..39.....fe
c2054f.....a7...
38..9..1c.7.f...
d4.6.....3.e..15
.1.......2..ca46
```

Output

```
Solution: 274c adb5 e086 193f
          ea8d 319f 2547 6bc0
          60b9 e82c df31 a457
          5f31 6047 acb9 e8d2

          a523 8cfe 7d64 b109
          9d74 2b06 1eac 5f83
          fc60 193a 582b de74
          8b1e 57d4 f903 2c6a

          1352 f4ad 07e8 96bc
          09f8 be72 46cd 35a1
          4eda c619 3b5f 0728
          b6c7 0583 9a12 4dfe

          c205 4fe8 61da 739b
          38ab 9261 c475 f0ed
          d496 7ac0 b3fe 8215
          71ef d35b 8290 ca46
```