# rush01();
# *skyscrapers_puzzle_solver();*

**jinlu’s group**

*‘jinlu’ + ‘alegesle’ + ‘moussade’*

*j.a.m*

01.03.2025 -> 03.03.2025

*may the moulinette be with us*

# roadmap();

## pré-ambule

Our task is to write a program that solves a 4x4 skyscraper puzzle.

- In the puzzle, we have a grid where each cell must be filled with a block numbered 1 to 4.
- Every row and column must contain each number exactly once.
- Around the grid are clues that tell us how many blocks are visible from that side.
- A block is visible only if it's taller than all the ones in front of it.
- Our goal is to arrange the numbers so that both the grid rules and the visibility clues are satisfied.

This roadmap breaks down the project into smaller, or bigger functions/programs (manageable steps). So we don't go crazy, we are literally building skyscrapers.

## brainstorming session notes

After meeting up for a quick huddle, we decided to first deepen our knowledge about matrices, skyscrapers puzzles and to try and find a strategy to solve the puzzle.

Team members *J and M* were familiar with Matrix as a formal logic concept, but also with Skyscrapers and mostly Sudoku. *M* had already a similar experience, trying to build a Sudoku solver a few years ago.

- During the exploration phase, we are mainly studying Soduko solvers to get inspired and find a strategy.
- We are overwhelmed by the complexity of the task, but ambitious and feel like having a solution on the tip of the lip.

## strategy agreed on post brainstorm

1. We are first starting by setting up the grid, using the #define N 4 preprocessor directive[1]
2. We will need a safety function, that will check if a number could be placed on a given cell without repeating on its row or column
3. We will need helper functions, ideally located each in a separate file (Merci Norminette). We could use one for the visibility of buildings, given the value outside of the grid, it would detect how many blocks are visible from one side of a row or column. Another one would be checking if the whole row / or column is matching the clues given outside of the grid.
4. To solve the puzzle, we have decided to use Backtracking, as many other groups suggested on Slack, and as some of our studied references suggested as well.
    a. Here, our function will start by the first cell. It will first start by checking the repetition safety via isSafe f(x). If allowed, the solver moves to the next cell in the grid, tries the next logical value, and so on. Until it hits a dead end,
    b. When a dead end is met, the solver moves one step back, to the last safe and sure value, then proceeds solving it again.

## research notes

You'll find down below, a compiled version of all notes taken during the research phase

1. #DEFINE

in C, define is a preprocessor directive defining macros.[2] it can be used to either **define constants**, **expressions** or **expressions with parameters**.

2. the safety function

We can either use Boolean, for false and true input parsing. but I think that regular integers returning values could work.
this function will ensure that no duplicates values (heights of building) exist in the same row or column

- protect the visibility constraints given by the hints outside of the grid

---

[1] Specifically, #define tells the preprocessor that everywhere the token M occurs, the text "4" should be used in its place.
https://stackoverflow.com/questions/5285155/utility-of-define-in-c

[2] macros refer to fragments of code, that we are giving a name. the library is going to be substituted by the preprocessor before actual compilation. in our library, macros can be used to create constants, inline functions or code shortcuts.
—-
for more in depth, ask chat abt Macros or look for Macros in sound design, same concept but simpler to understand as its visualized with sound.

- if all conditions are met, the function returns “true”

3. helper functions
   *visibility_fx*
- “*how many skyscrapers are visible from one side of a row or column*
  - *validates the clues outside of grid*
  - *in tandem with isSafe*

    <u>syntax</u>

    declaring the fx :

    **int visibility_fx(int line[GRID_SIZE]) { ….. arguments …. }**

    usage in row/column :

    **int row[] = {2, 3, 1, 4};** // Example row
    **printf("Visible buildings from the left: %d\n", visibility_fx(row))**

4. clue_checker
- Verifies if a row or column satisfies the external clues.
- Uses visibility_fx to determine the number of visible buildings.
- Checks both left/right or top/bottom clues depending on orientation.

**backtracking**

- We start at the first empty cell in the grid.
- We try putting a number (1, 2, 3, etc.).
- We use isSafe to check if that number follows the rules.
- If it follows the rules, we move to the next empty cell.
- If we reach a dead end (no number works), we erase the last number and try a different one.
  - i++: until return 1 or 0
    - Try something.
    - If it works, move forward.
    - If stuck, undo and try again.
    - Repeat until you solve the problem.

**steps followed on execution**

Step 1: Setup Project & Define Grid

- ☐ Define constants using #define N 4.
- ☐ Create 4x4 grid with all cells initialized to 0.

Step 2: Develop isSafe Function

- ☐ Write the isSafe function to check if a number is safe for placement (no row/column duplicates).
- ☐ Test the function with random values to verify its correctness.

Step 3: Helper Functions

- ☐ a. visibility_fx
    - ☐ Write visibility_fx function to calculate how many buildings are visible from one side.
    - ☐ Test with various rows and columns.
- ☐ b. clue_checker
    - ☐ Write clue_checker function to validate rows and columns against external clues.
    - ☐ Test with different row/column combinations and clues.

Step 4: Implement Backtracking

- ☐ Write the backtracking function (solveSkyscraper) to try placing numbers, check for safe placement, and backtrack if needed.
- ☐ Test the backtracking function with a sample puzzle.

Step 5: Final Integration

- ☐ Combine all functions (isSafe, visibility_fx, clue_checker, solveSkyscraper).
- ☐ Test with a full puzzle to ensure the solution is correct.
- ☐ Handle edge cases where no solution exists.

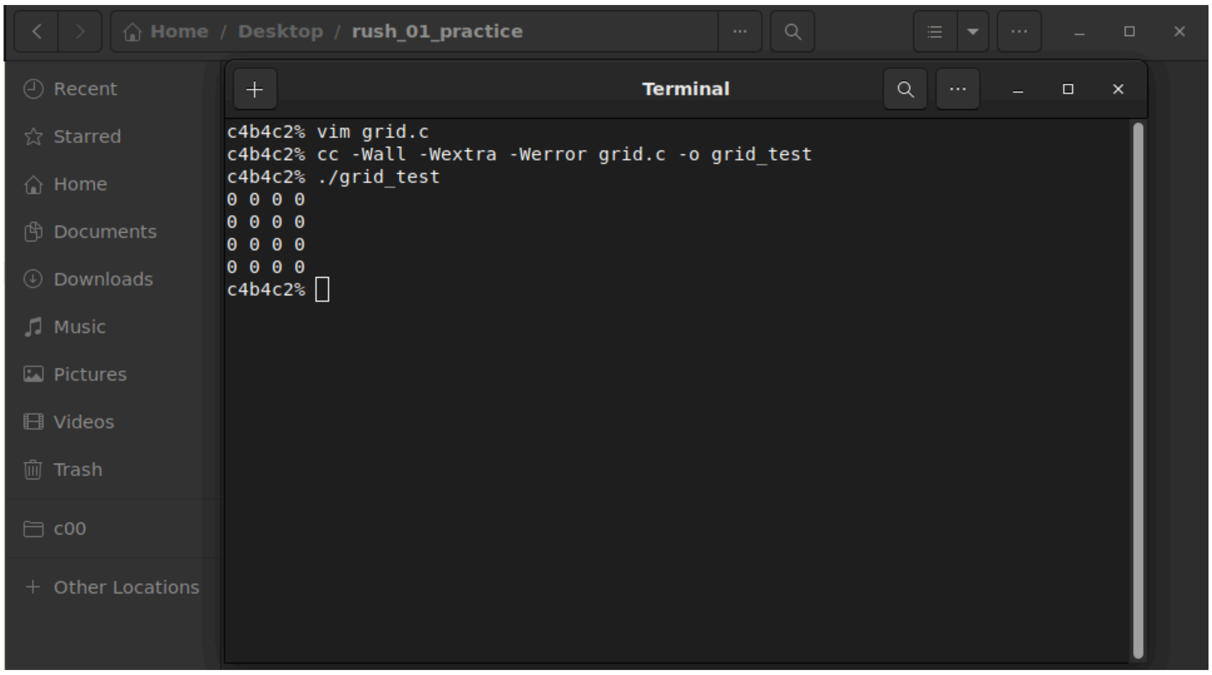
**[log + notes]**

13.55 : starting the set up. created the git repo[3]

14.04 : created and implemented a grid. testing out different arrays,

14.16 : still debugging, argument problem. maybe syntax?

14.27 : OK ! Grid prints perfectly, problem was typo in parameter grid size. zero values only….

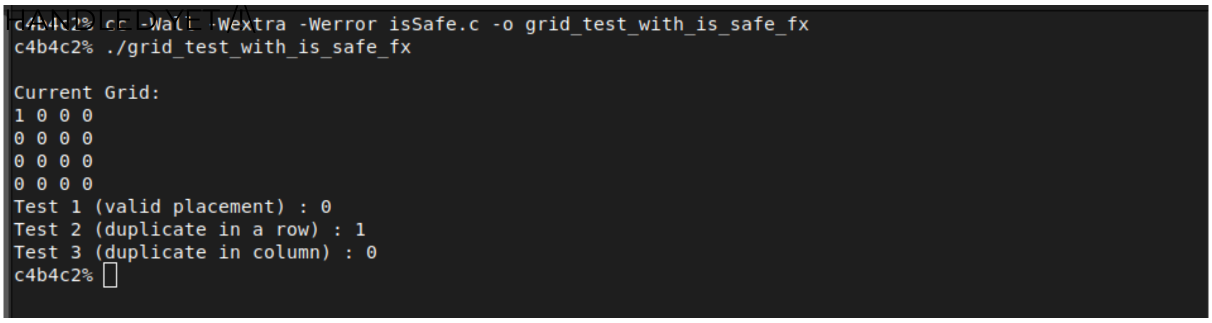


```
c4b4c2% vim grid.c
c4b4c2% cc -Wall -Wextra -Werror grid.c -o grid_test
c4b4c2% ./grid_test
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
c4b4c2%
```

14.29 : time for the isSafe function….  trying to understand how the logic could be implemented to verify the rows and columns repetition each time it is being called, also trying to understand fully how recursion works —------------ ETA start next step : > 15.00

15.14 : isSafe function parameters setting… trying to think of a formal logic of arguments to attribute (to signify rows and columns repetition)

15.40 : isSafe function runs smoothly. repetition on row checked : not possible, the same for columns. /!\ CRASH TEST NECESSARY STILL - NO EDGE CASES HANDLED YET /!\



```
c4b4c2% cc -Wall -Wextra -Werror isSafe.c -o grid_test_with_is_safe_fx
c4b4c2% ./grid_test_with_is_safe_fx

Current Grid:
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Test 1 (valid placement) : 0
Test 2 (duplicate in a row) : 1
Test 3 (duplicate in column) : 0
c4b4c2%
```

3

15.46 : main, define grid, printgrid, isSafe on one file success, break !!!! next need to work on visibility_fx(); ETE : one hour.

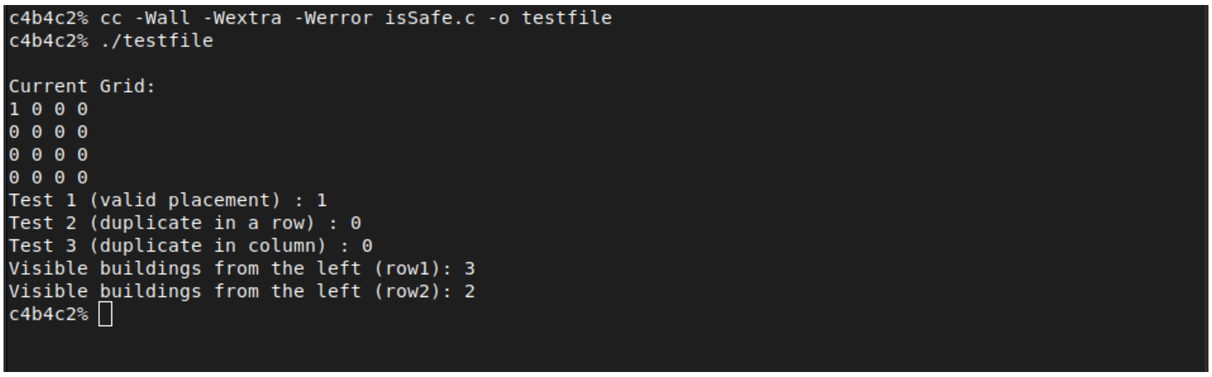
16.39 : back. starting visibility_fx, I'm doubtful on where to start. due to the length of code we are at now (70lines already) better to start on a new file. so that we will have two files for now : main.c (grid, is safe function and main for testing) and helper_functions.c

16.54 : UPDATE. we are going to have to use a recursive backtracking algorithm, there is no way out of it. seems doable, but totally new to me. will have to dig a bit more.

16.55 : 
```
int clues[4][GRID_SIZE] = {
    {2, 3, 1, 2},  // Top clues
    {1, 2, 3, 2},  // Bottom clues
    {3, 2, 1, 2},  // Left clues
    {2, 1, 3, 2}   // Right clues
};
```
this is how we are going to give in our hint values (outside of grid) indicating visibility of building from given p.o.v \\\ stored in 4x4 arrays with separate clues for each direction.

16.58 : starting visibility_fx, to count how many buildings are visible from one side of a row or col. it will start from one side, keep track of the tallest building seen so far. every time a new tallest building appears, the count will be increased, and we return the final count.

17.25 : SUCCESS. visibility fx is implemented and can count w/o error. taking a quick break b4 moving to backtracking...


```
c4b4c2% cc -Wall -Wextra -Werror isSafe.c -o testfile
c4b4c2% ./testfile

Current Grid:
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Test 1 (valid placement) : 1
Test 2 (duplicate in a row) : 0
Test 3 (duplicate in column) : 0
Visible buildings from the left (row1): 3
Visible buildings from the left (row2): 2
c4b4c2% 
```

18.18 : working on backtracking,  (cfr. 3 tab) but got a solution

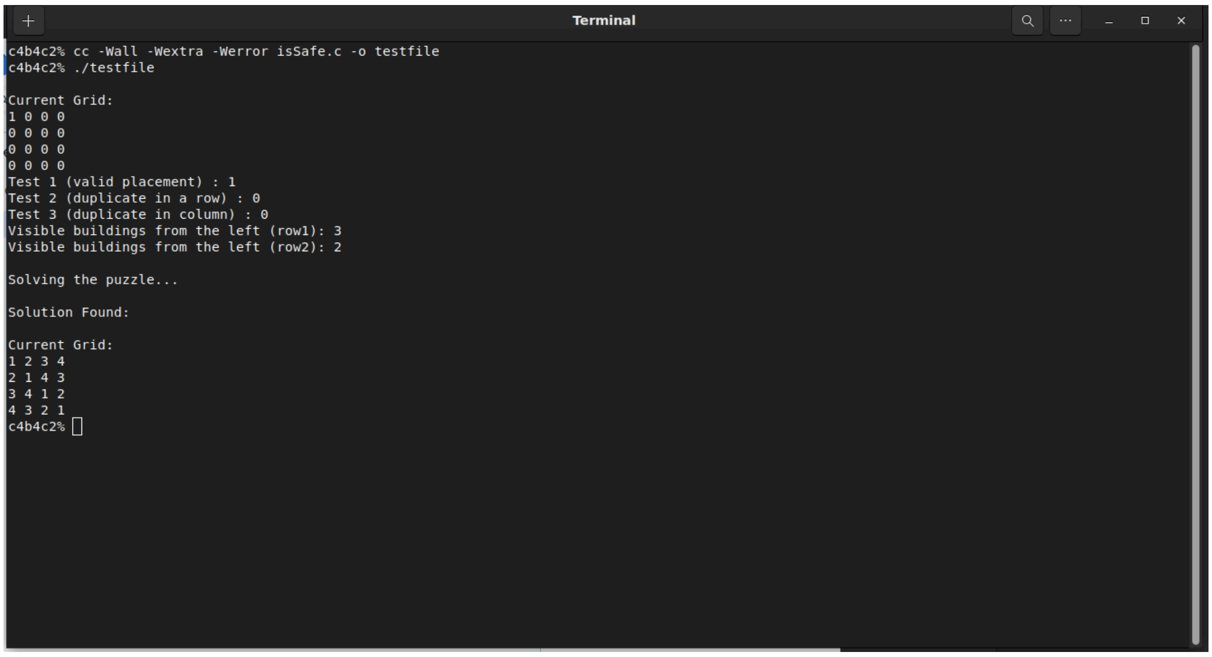
18.25 : actual topo of situation; imagine this like solving a maze where you dont know the correct path, and every time you hit a dead end, you go back to try a different path. it is a smart way to explore all possibilities without wasting time on wrong ones.

(...) concretely, this means that our recursive algo will start at the first empty cell in the grid, and before placing a number (1 to the GRID_SIZE) it will check if all rules and requirements are met (isSafe + clue_checker). if valid, move to the next cell. if not, it will try the next number.

if all the numbers have failed, the algo goes back and undoes the last move before trying a different number.

repeat until the whole grid is filled correctly.

19.27 : SUCCESSS !!!!!! WORKS PERFECTLY.

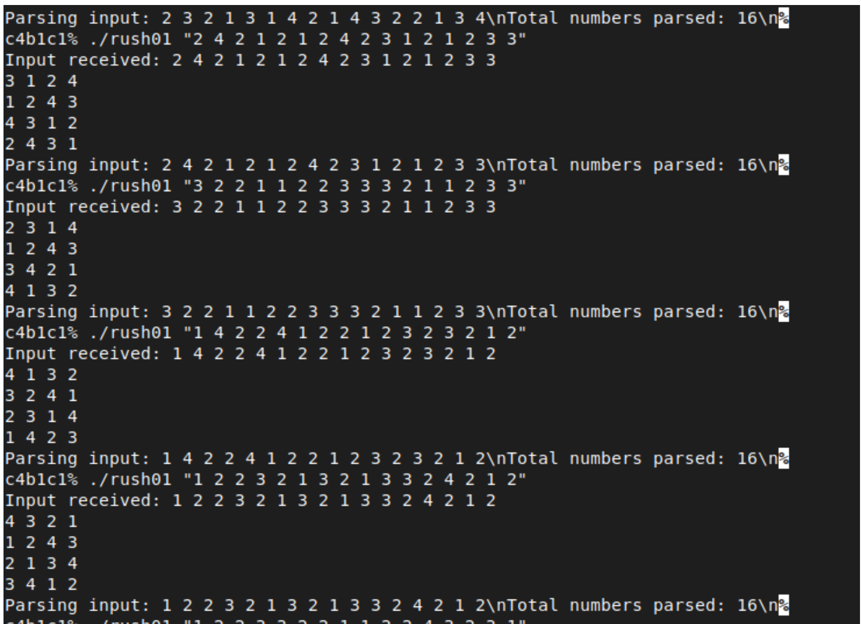


```
c4b4c2% cc -Wall -Wextra -Werror isSafe.c -o testfile
c4b4c2% ./testfile

Current Grid:
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Test 1 (valid placement) : 1
Test 2 (duplicate in a row) : 0
Test 3 (duplicate in column) : 0
Visible buildings from the left (row1): 3
Visible buildings from the left (row2): 2

Solving the puzzle...

Solution Found:

Current Grid:
1 2 3 4
2 1 4 3
3 4 1 2
4 3 2 1
c4b4c2% 
```

we have a list of bugs to work on tomorrow, you can find them in the third tab...

19.29 : code pushed on repo, now going AFK. will report back tomorrow morning

- 13.31 : worked remotely on the program yesterday, coul- 17.40 : had no moment to log. but will paste my notes later
  SUCCESSFUL MISSION



```
Parsing input: 2 3 2 1 3 1 4 2 1 4 3 2 2 1 3 4\nTotal numbers parsed: 16\n%
c4b1c1% ./rush01 "2 4 2 1 2 1 2 4 2 3 1 2 1 2 3 3"
Input received: 2 4 2 1 2 1 2 4 2 3 1 2 1 2 3 3
3 1 2 4
1 2 4 3
4 3 1 2
2 4 3 1
Parsing input: 2 4 2 1 2 1 2 4 2 3 1 2 1 2 3 3\nTotal numbers parsed: 16\n%
c4b1c1% ./rush01 "3 2 2 1 1 2 2 3 3 3 2 1 1 2 3 3"
Input received: 3 2 2 1 1 2 2 3 3 3 2 1 1 2 3 3
2 3 1 4
1 2 4 3
3 4 2 1
4 1 3 2
Parsing input: 3 2 2 1 1 2 2 3 3 3 2 1 1 2 3 3\nTotal numbers parsed: 16\n%
c4b1c1% ./rush01 "1 4 2 2 4 1 2 2 1 2 3 2 3 2 1 2"
Input received: 1 4 2 2 4 1 2 2 1 2 3 2 3 2 1 2
4 1 3 2
3 2 4 1
2 3 1 4
1 4 2 3
Parsing input: 1 4 2 2 4 1 2 2 1 2 3 2 3 2 1 2\nTotal numbers parsed: 16\n%
c4b1c1% ./rush01 "1 2 2 3 2 1 3 2 1 3 3 2 4 2 1 2"
Input received: 1 2 2 3 2 1 3 2 1 3 3 2 4 2 1 2
4 3 2 1
1 2 4 3
2 1 3 4
3 4 1 2
Parsing input: 1 2 2 3 2 1 3 2 1 3 3 2 4 2 1 2\nTotal numbers parsed: 16\n%
```

**last minute debugging notes // problems encountered after norminette**

- divided each function into a separate file,
- handling all edge cases and with comments on the side of code and a
- added a header file regrouping all function prototypes and macros needed
- replaced boolean with simple easy peasy integers
- converted define grid size into an inline const
- converted if loops to while loops
- replaced print f  with write functions (some ftputchar and nbr here and there)
- implemented atoi for the bonus points
- all functions are norminette and moulinette friendly
- readme file with docu.
- compiles perfectly with cc, but no output file - no need for a makeme file

## Solving skyscraper puzzles

- **Understanding the logic behind skyscrapers and exploring the matter**
    - Kolijn, L. (2022). *Generating and solving skyscraper puzzles using a SAT solver (Bachelor's thesis, Radboud University, pp. 1-12).* https://www.cs.ru.nl/bachelors-theses/2022/Laura_Kolijn___1025724___Generating_and_Solving_Skyscrapers_Puzzles_Using_a_SAT_Solver.pdf

## Backtracking Algorithm

- **Definition and Applications**:
    - Knuth, D. E. (2000). *Dancing Links*. In *Millennial Perspectives in Computer Science* (pp. 187-214). Palgrave Macmillan, London. https://en.wikipedia.org/wiki/Dancing_Links
    - Wikipedia contributors. (2023). *Backtracking*. In *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Backtracking
    - Programiz. (n.d.). *Backtracking Algorithm*. https://www.programiz.com/dsa/backtracking-algorithm

## Recursive Programming

- **Concept and Implementation**:
    - GeeksforGeeks. (n.d.). *Recursion in C*. https://www.geeksforgeeks.org/recursion-in-c/
    - TutorialsPoint. (n.d.). *C Recursion*. https://www.tutorialspoint.com/cprogramming/c_recursion.htm

## C Programming Concepts

- **Variables and Data Types**:
    - GeeksforGeeks. (n.d.). *Data Types in C*. https://www.geeksforgeeks.org/data-types-in-c/
    - cplusplus.com. (n.d.). *C Data Types*. https://cplusplus.com/doc/tutorial/variables/
- **Control Structures (if, else, while, return)**:
    - GeeksforGeeks. (n.d.). *Decision Making in C (if, if..else, Nested if, etc.).* https://www.geeksforgeeks.org/decision-making-c-c-else-nested-else/
    - GeeksforGeeks. (n.d.). *Loops in C and C++*. https://www.geeksforgeeks.org/loops-in-c-and-cpp/
- **Functions and Prototypes**:
    - TutorialsPoint. (n.d.). *C Function Prototypes*. https://www.tutorialspoint.com/cprogramming/c_function_prototypes.htm

- **Pointers**:
  - GeeksforGeeks. (n.d.). *Pointers in C.* https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/
  - cplusplus.com. (n.d.). *Pointers in C.* https://cplusplus.com/doc/tutorial/pointers/

- **Header Files and Macros**:
  - GeeksforGeeks. (n.d.). *Header Files in C.* https://www.geeksforgeeks.org/header-files-in-c-cpp-and-its-uses/
  - TutorialsPoint. (n.d.). *C Preprocessors (#define and Macros).* https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm

## Input Handling & Error Checking

- **Parsing User Input and Validation**:
  - GeeksforGeeks. (n.d.). *How to Validate Input in C.* https://www.geeksforgeeks.org/input-validation-in-c/
  - cplusplus.com. (n.d.). *C String to Integer Conversion (atoi(), strtol(), etc.).* https://cplusplus.com/reference/cstdlib/atoi/

## Practical Implementations

- **Step-by-Step Implementations of Backtracking in C**:
  - Stack Overflow. (2019). *Recursive Backtracking Algorithm in C to Solve a Sudoku.* https://stackoverflow.com/questions/56457867/recursive-backtracking-algorithm-in-c-to-solve-a-sudoku
  - Wikipedia contributors. (2023). *Dancing Links.* https://en.wikipedia.org/wiki/Dancing_Links
- **Sudoku in C**
  - GeekforGeeks, (n.d) *Sudoku in C* https://www.geeksforgeeks.org/sudoku-in-c/

## Youtube videos

- **Coding a Sudoku Solver in C**
  - Badcodinghabits, *Coding a Sudoku Solver in C - Part 1*
    - https://www.youtube.com/watch?v=9aMUyoYDI-0&
  - Badcodinghabits, *Coding a Sudoku Solver in C - Part 2*
    - https://www.youtube.com/watch?v=SfOEjjqtEyk&

https://www.w3schools.com/c/c_functions_recursion.php
https://stackoverflow.com/questions/18872553/skyscraper-puzzle-algorithm
https://github.com/norvig/pytudes/blob/main/ipynb/Sudoku.ipynb
https://www.conceptispuzzles.com/index.aspx?uri=puzzle/skyscrapers/techniques
https://www.krnsk0.dev/posts/skyscraper-puzzle-1/
https://www.geeksforgeeks.org/_generic-keyword-c/