rush01

Part 1: Understanding What You're Building

The Big Picture

You're creating a **puzzle solver** that uses a technique called **backtracking**. Think of it like solving a maze:

- Try a path
- If it leads to a dead end, go back and try a different path
- Keep doing this until you find the exit

In our case:

- The "maze" is placing numbers 1-4 in a 4x4 grid
- The "dead ends" are invalid placements that violate rules
- The "exit" is a complete, valid solution

Part 2: Breaking Down the Problem

What Makes a Valid Solution?

Your solution must satisfy **THREE types of rules**:

Rule 1: Row Uniqueness

Each row must contain 1, 2, 3, 4 exactly once.

```
✓ Valid:   [1, 2, 3, 4]
✗ Invalid: [1, 2, 2, 4] (two 2s!)
```

Rule 2: Column Uniqueness

Each column must contain 1, 2, 3, 4 exactly once.

```
✓ Valid column:
  1
  2
  3
  4

✗ Invalid column:
  1
  2
  2  (duplicate!)
  4
```

Rule 3: Visibility Constraints

The numbers represent building heights. From each direction, you must see a specific number of buildings.

How Visibility Works:

Imagine you're standing on the left side of this row: `[1, 3, 4, 2]`

- You see building 1 (height 1) ✓
- You see building 3 (height 3) - it's taller than 1 ✓
- You see building 4 (height 4) - it's taller than 3 ✓
- You DON'T see building 2 (height 2) - it's blocked by 4 ✗

Total visible from left: 3 buildings

Now from the right of the same row: `[1, 3, 4, 2]`

- You see building 2 (height 2) ✓
- You see building 4 (height 4) - taller than 2 ✓
- You DON'T see 3 or 1 (blocked by 4) ✗

Total visible from right: 2 buildings

Part 3: Your Development Strategy

Phase 1: Start with the Foundation (Day 1)

Step 1: Create Your Project Structure

Create these files:

```
main.c      - Entry point
utils.c     - Basic functions (ft_putchar, ft_putstr)
parser.c    - Read and validate input
grid.c      - Initialize and print grid
```

Step 2: Implement Basic I/O First

Why start here? You need to see output to debug later!

```c
// utils.c
void ft_putchar(char c)
{
    write(1, &c, 1);
}

void ft_putstr(char *str)
{
    int i = 0;
    while (str[i])
    {
        ft_putchar(str[i]);
        i++;
    }
}
```

Test immediately:

```c
int main(void)
{
    ft_putstr("Hello Rush01!\n");
    return (0);
}
```

Step 3: Parse the Input

Strategy: Build your parser in stages:

Stage A: Just count the arguments

```c
int main(int argc, char **argv)
{
    if (argc != 2)
    {
        ft_putstr("Error\n");
        return (1);
    }
    ft_putstr("Correct number of arguments!\n");
    return (0);
}
```

Test: `./rush01 "test"` should succeed, `./rush01` should print "Error"

Stage B: Validate the string format

Input format: `"4 3 2 1 1 2 2 2 4 3 2 1 1 2 2 2"`

- 16 digits
- 15 spaces
- Total: 31 characters

Strategy for parsing:

```c
int parse_input(char *str, int *clues)
{
    int i = 0;  // clues array index
    int j = 0;  // string index

    // Loop 16 times (one for each clue)
    while (i < 16)
    {
        // Check if current char is a valid digit (1-4)
        if (str[j] < '1' || str[j] > '4')
            return (0);  // Error: invalid digit

        // Convert char to int and store
        clues[i] = str[j] - '0';

        i++;
        j++;

        // After each digit (except last), expect a space
        if (i < 16)
        {
            if (str[j] != ' ')
                return (0);  // Error: missing space
            j++;
        }
    }

    // Make sure string ends here
    if (str[j] != '\0')
        return (0);  // Error: extra characters

    return (1);  // Success!
}
```

Pro Tip: You can also use `i * 2` indexing:

- Clue 0 is at position 0
- Clue 1 is at position 2 (skip the space)

- Clue 2 is at position 4
- Pattern: `clues[i] = str[i * 2] - '0';`

Test your parser:

```c
int main(int argc, char **argv)
{
    int clues[16];
    int i;

    if (argc != 2 || !parse_input(argv[1], clues))
    {
        ft_putstr("Error\n");
        return (1);
    }

    // Print parsed clues to verify
    i = 0;
    while (i < 16)
    {
        ft_putchar(clues[i] + '0');
        ft_putchar(' ');
        i++;
    }
    ft_putchar('\n');

    return (0);
}
```

Phase 2: Build the Grid System (Day 2)

Step 4: Grid Initialization

Strategy: Use a 2D array, initialize everything to 0 (empty)

```c
void init_grid(int grid[4][4])
{
    int i = 0;
    while (i < 4)
    {
        int j = 0;
        while (j < 4)
        {
            grid[i][j] = 0;
            j++;
        }
        i++;
    }
}
```

Step 5: Grid Printing

Critical: Get the format EXACTLY right!

- Space between numbers: `1 2 3 4`
- Newline after each row
- No trailing spaces

```
void print_grid(int grid[4][4])
{
    int i = 0;
    while (i < 4)
    {
        int j = 0;
        while (j < 4)
        {
            ft_putchar(grid[i][j] + '0');

            // Space after each number except the last in row
            if (j < 3)
                ft_putchar(' ');

            j++;
        }
        ft_putchar('\n');
        i++;
    }
}
```

Test with a hardcoded grid:

```
int main(void)
{
    int grid[4][4] = {
        {1, 2, 3, 4},
        {2, 3, 4, 1},
        {3, 4, 1, 2},
        {4, 1, 2, 3}
    };

    print_grid(grid);
    return (0);
}
```

Expected output:

```
1 2 3 4
2 3 4 1
3 4 1 2
4 1 2 3
```

Phase 3: Implement Validation (Day 3)

Step 6: Check Row Validity

Purpose: Before placing a number, check if it already exists in that row.

Strategy:

1. Loop through all columns in the given row
2. If you find the number, return 0 (invalid)
3. If you finish the loop without finding it, return 1 (valid)

```c
int is_valid_row(int grid[4][4], int row, int num)
{
    int col = 0;
    while (col < 4)
    {
        if (grid[row][col] == num)
            return (0);  // Found it, invalid!
        col++;
    }
    return (1);  // Didn't find it, valid!
}
```

Test it:

```c
int main(void)
{
    int grid[4][4] = {
        {1, 2, 0, 0},  // Row 0 has 1 and 2
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };
    // Should return 0 (invalid) - 1 is already in row 0
    if (is_valid_row(grid, 0, 1))
        ft_putstr("Valid\n");
    else
        ft_putstr("Invalid\n");

    // Should return 1 (valid) - 3 is not in row 0
    if (is_valid_row(grid, 0, 3))
        ft_putstr("Valid\n");
    else
        ft_putstr("Invalid\n");

    return (0);
}
```

Step 7: Check Column Validity

Same logic, but loop through rows instead:

```c
int is_valid_col(int grid[4][4], int col, int num)
{
    int row = 0;
    while (row < 4)
    {
        if (grid[row][col] == num)
            return (0);
        row++;
    }
    return (1);
}
```

Phase 4: Implement Visibility Counting (Day 4)

This is the **most important** part! Understanding this is key to solving the puzzle.

Step 8: Count Visible from Left

Mental Model: You're standing on the left side of a row, looking right.

Algorithm:

1. Start with `visible = 0` and `max_height = 0`
2. Go through each building from left to right
3. If a building is taller than `max_height`:

    - Increment `visible` (you can see it!)
    - Update `max_height` to this building's height
4. Return `visible`

```
int count_visible_left(int grid[4][4], int row)
{
    int col = 0;
    int visible = 0;
    int max_height = 0;

    while (col < 4)
    {
        // If this building is taller than anything we've seen
        if (grid[row][col] > max_height)
        {
            visible++;
            max_height = grid[row][col];
        }
        col++;
    }

    return (visible);
}
```

Example Walkthrough:

```
Row: [1, 3, 4, 2]

col=0: grid[row][0] = 1, max_height = 0
       1 > 0? YES → visible=1, max_height=1

col=1: grid[row][1] = 3, max_height = 1
       3 > 1? YES → visible=2, max_height=3

col=2: grid[row][2] = 4, max_height = 3
       4 > 3? YES → visible=3, max_height=4

col=3: grid[row][3] = 2, max_height = 4
       2 > 4? NO → visible stays 3

Result: 3 visible buildings
```

Step 9: Count Visible from Right

Same logic, but start from the right and go left:

```c
int count_visible_right(int grid[4][4], int row)
{
    int col = 3;  // Start from rightmost column
    int visible = 0;
    int max_height = 0;

    while (col >= 0)  // Go left
    {
        if (grid[row][col] > max_height)
        {
            visible++;
            max_height = grid[row][col];
        }
        col--;  // Move left
    }

    return (visible);
}
```

Step 10: Count Visible from Top and Bottom

Same pattern, but iterate through rows:

```c
int count_visible_top(int grid[4][4], int col)
{
    int row = 0;  // Start from top
    int visible = 0;
    int max_height = 0;

    while (row < 4)
    {
        if (grid[row][col] > max_height)
        {
            visible++;
            max_height = grid[row][col];
        }
        row++;
    }

    return (visible);
}

int count_visible_bottom(int grid[4][4], int col)
{
    int row = 3;  // Start from bottom
    int visible = 0;
    int max_height = 0;
    while (row >= 0)
    {
        if (grid[row][col] > max_height)
        {
            visible++;
            max_height = grid[row][col];
        }
        row--;
    }

    return (visible);
}
```

Test these functions thoroughly! They're critical.

Phase 5: Connect Visibility to Clues (Day 5)

Step 11: Understanding Clue Positions

Your input `"4 3 2 1 1 2 2 2 4 3 2 1 1 2 2 2"` becomes:

```
clues[0] = 4   ← Top of column 0
clues[1] = 3   ← Top of column 1
clues[2] = 2   ← Top of column 2
clues[3] = 1   ← Top of column 3
clues[4] = 1   ← Bottom of column 0
clues[5] = 2   ← Bottom of column 1
clues[6] = 2   ← Bottom of column 2
clues[7] = 2   ← Bottom of column 3
clues[8] = 4   ← Left of row 0
clues[9] = 3   ← Left of row 1
clues[10] = 2  ← Left of row 2
clues[11] = 1  ← Left of row 3
clues[12] = 1  ← Right of row 0
clues[13] = 2  ← Right of row 1
clues[14] = 2  ← Right of row 2
clues[15] = 2  ← Right of row 3
```

Step 12: Check Row Against Clues

```
int check_row_visibility(int grid[4][4], int row, int *clues)
{
    // Get the clues for this row
    int left_clue = clues[8 + row];
    int right_clue = clues[12 + row];
    // Count how many buildings are actually visible
    int left_visible = count_visible_left(grid, row);
    int right_visible = count_visible_right(grid, row);

    // Do they match the clues?
    if (left_visible != left_clue)
        return (0);  // No match!
    if (right_visible != right_clue)
        return (0);  // No match!

    return (1);  // All checks passed!
}
```

Step 13: Check Column Against Clues

```
int check_col_visibility(int grid[4][4], int col, int *clues)
{
    int top_clue = clues[col];
    int bottom_clue = clues[4 + col];

    int top_visible = count_visible_top(grid, col);
    int bottom_visible = count_visible_bottom(grid, col);

    if (top_visible != top_clue)
        return (0);
    if (bottom_visible != bottom_clue)
        return (0);

    return (1);
}
```

Phase 6: The Backtracking Solver (Day 6-7)

Step 14: Understanding Position Mapping

You'll solve the grid cell by cell, position 0 to 15:

```
Position:  0  1  2  3
           4  5  6  7
           8  9 10 11
          12 13 14 15
```

Convert position to row/col:

- `row = position / 4`
- `col = position % 4`

Examples:

- Position 0: row = 0/4 = 0, col = 0%4 = 0 → (0,0)
- Position 5: row = 5/4 = 1, col = 5%4 = 1 → (1,1)
- Position 11: row = 11/4 = 2, col = 11%4 = 3 → (2,3)

Step 15: The Solve Function - Your Brain

The backtracking algorithm in plain English:

```
1. Are we at position 16? If yes, we've filled all cells! Return success!

2. Figure out which row and column this position is

3. Try placing each number (1, 2, 3, 4):
   a. Check if this number is already in this row → Skip if yes
   b. Check if this number is already in this column → Skip if yes
   c. Place the number in the grid

   d. Did we just complete a row (last column)?
      → Check if row visibility matches clues
      → If not, remove number and try next

   e. Did we just complete a column (last row)?
      → Check if column visibility matches clues
      → If not, remove number and try next

   f. Try to solve the next position recursively
      → If it succeeds, great! Return success
      → If it fails, remove this number and try the next one

4. If we tried all numbers and none worked, return failure
```

Now in C:

```c
int solve(int grid[4][4], int *clues, int pos)
{
    int row, col, num;

    // Base case: solved!
    if (pos == 16)
        return (1);

    // Calculate row and column
    row = pos / 4;
    col = pos % 4;

    // Try each number 1-4
    num = 1;
    while (num <= 4)
    {
        // Is this number valid in this position?
        if (is_valid_row(grid, row, num) &&
            is_valid_col(grid, col, num))
        {
            // Place it
            grid[row][col] = num;

            // Just completed a row?
            if (col == 3)
            {
                if (!check_row_visibility(grid, row, clues))
                {
                    grid[row][col] = 0;  // Remove it
                    num++;
                    continue;  // Try next number
                }
            }

            // Just completed a column?
            if (row == 3)
            {
                if (!check_col_visibility(grid, col, clues))
                {
                    grid[row][col] = 0;  // Remove it
                    num++;
                    continue;  // Try next number
                }
            }

            // Try to solve the rest
            if (solve(grid, clues, pos + 1))
                return (1);  // Success! Pass it up

            // Failed, backtrack
            grid[row][col] = 0;
        }

        num++;
    }

    // Tried everything, no solution from here
    return (0);
}
```

Step 16: Put It All Together

```c
int main(int argc, char **argv)
{
    int grid[4][4];
    int clues[16];

    // Validate arguments
    if (argc != 2)
    {
        ft_putstr("Error\n");
        return (1);
    }

    // Parse input
    if (!parse_input(argv[1], clues))
    {
        ft_putstr("Error\n");
        return (1);
    }

    // Initialize grid
    init_grid(grid);

    // Solve it!
    if (solve(grid, clues, 0))
    {
        print_grid(grid);
    }
    else
    {
        ft_putstr("Error\n");
    }

    return (0);
}
```

Part 4: Testing Strategy

Test 1: The Provided Example

```
./rush01 "4 3 2 1 1 2 2 2 4 3 2 1 1 2 2 2"
```

Expected output:

```
1 2 3 4
2 3 4 1
3 4 1 2
4 1 2 3
```

Test 2: Input Validation

```
# No arguments
./rush01
# Expected: Error

# Too many arguments
./rush01 "test" "test"
# Expected: Error

# Invalid character
./rush01 "4 3 2 1 1 2 2 2 4 3 2 1 1 2 2 x"
# Expected: Error

# Invalid number
./rush01 "5 3 2 1 1 2 2 2 4 3 2 1 1 2 2 2"
# Expected: Error

# Too short
./rush01 "4 3 2 1"
# Expected: Error
```

Test 3: Edge Cases

```
# All ones (impossible)
./rush01 "1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1"
# Expected: Error (or a solution if one exists)

# All fours (impossible)
./rush01 "4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4"
# Expected: Error
```

Part 5: Debugging Guide

When Your Program Crashes

Problem: Segmentation Fault

Check:

1. Are you accessing `grid[row][col]` with row/col < 4?
2. Are you accessing `clues[i]` with i < 16?
3. Is your base case in `solve()` correct?

Debug technique:

```c
// Add at start of solve()
printf("Position: %d, Row: %d, Col: %d\n", pos, row, col);
```

Problem: Infinite Loop

Check:

1. Does your solve function return 1 when pos == 16?
2. Are you incrementing `num++` in the loop?
3. Are you returning 0 at the end of solve()?

Debug technique:

```c
// Add a counter
static int calls = 0;
calls++;
if (calls > 10000) {
    printf("Too many calls!\n");
    exit(1);
}
```

When Output is Wrong

Problem: Wrong Numbers

Check:

1. Is your visibility counting correct?
2. Test each counting function separately
3. Are you checking the right clue indices?

Debug technique:

```c
// Print the grid after each placement
void print_debug(int grid[4][4], int pos)
{
    printf("\n=== Position %d ===\n", pos);
    print_grid(grid);
}
```

Problem: Wrong Format

Check:

1. Spaces between numbers?
2. Newline after each row?
3. No extra spaces or newlines?

Part 6: Optimization Strategies

Strategy 1: Early Termination

Current: We check visibility only when row/col is complete

Better: Check if we're already violating visibility before completing

For example, if left clue is 2, but we already see 3 buildings at column 2, we can stop.

Strategy 2: Smart Clue Analysis

Observation: If a clue is 4, that row/column must be `[1, 2, 3, 4]`

Observation: If a clue is 1, the first building must be 4

You can pre-fill some cells based on clues before starting backtracking.

Strategy 3: Better Ordering

Try filling cells with more constraints first (corners, edges)

Part 7: Common Mistakes

Mistake 1: Not Resetting Cells

```
// WRONG
if (solve(grid, clues, pos + 1))
    return (1);
// Forgot to reset!
// CORRECT
if (solve(grid, clues, pos + 1))
    return (1);
grid[row][col] = 0;  // Reset before trying next number
```

Mistake 2: Wrong Clue Indexing

```
// WRONG - off by one!
left_clue = clues[9 + row];

// CORRECT
left_clue = clues[8 + row];
```

Mistake 3: Checking Visibility Too Early

```
// WRONG - row might not be complete!
grid[row][col] = num;
if (!check_row_visibility(grid, row, clues))
    return (0);
```

```
// CORRECT - only check when complete
if (col == 3 && !check_row_visibility(grid, row, clues))
{
    grid[row][col] = 0;
    continue;
}
```

Part 8: Bonus - Variable Grid Sizes

The Challenge

Support grids from 4x4 up to 9x9.

Required Changes

1. Dynamic Grid Size

```
// Instead of hardcoding 4
#define GRID_SIZE 4
```

```c
// Pass size as parameter
int solve(int **grid, int *clues, int pos, int size)
```

2. Dynamic Memory Allocation

```c
int **allocate_grid(int size)
{
    int **grid;
    int i;

    grid = malloc(sizeof(int *) * size);
    i = 0;
    while (i < size)
    {
        grid[i] = malloc(sizeof(int) * size);
        i++;
    }
    return (grid);
}

void free_grid(int **grid, int size)
{
    int i = 0;
    while (i < size)
    {
        free(grid[i]);
        i++;
    }
    free(grid);
}
```

3. Different Clue Count

- 4x4 grid: 16 clues
- 5x5 grid: 20 clues
- 6x6 grid: 24 clues
- Formula: `size * 4`

Part 9: Final Checklist

Before submission:

- [ ] Compiles with `-Wall -Wextra -Werror`
- [ ] No warnings

- [ ] Only uses `write`, `malloc`, `free`
- [ ] Handles invalid input correctly
- [ ] Prints "Error\n" for all error cases
- [ ] Output format is exact (test with `| cat -e`)
- [ ] Works with provided example
- [ ] No memory leaks (test with `valgrind`)
- [ ] Passes norminette (if required)

Summary

Your Development Timeline

Day 1: Input parsing and validation
Day 2: Grid system (init, print)
Day 3: Validation functions (row, col)
Day 4: Visibility counting (all 4 directions)
Day 5: Connect visibility to clues
Day 6-7: Backtracking solver
Day 8: Testing and debugging

Key Insights

1. **Backtracking is trial and error**: Try it, if it fails, undo it
2. **Check constraints early**: Don't wait until the end
3. **Test incrementally**: Don't write everything then test
4. **Visibility is counting**: Track max height as you go
5. **Position mapping is math**: `row = pos / 4`, `col = pos % 4`

The Core Algorithm

```
For each position:
    For each number 1-4:
        If number is valid:
            Place it
            Check visibility if row/col complete
            If constraints satisfied:
                Try to solve next position
                If success: return success
            Remove the number (backtrack)
    Return failure
```

Good luck! Remember: Understanding backtracking is 80% of the solution. The rest is just careful implementation.