COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 3 Nonogram

Honor Code

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

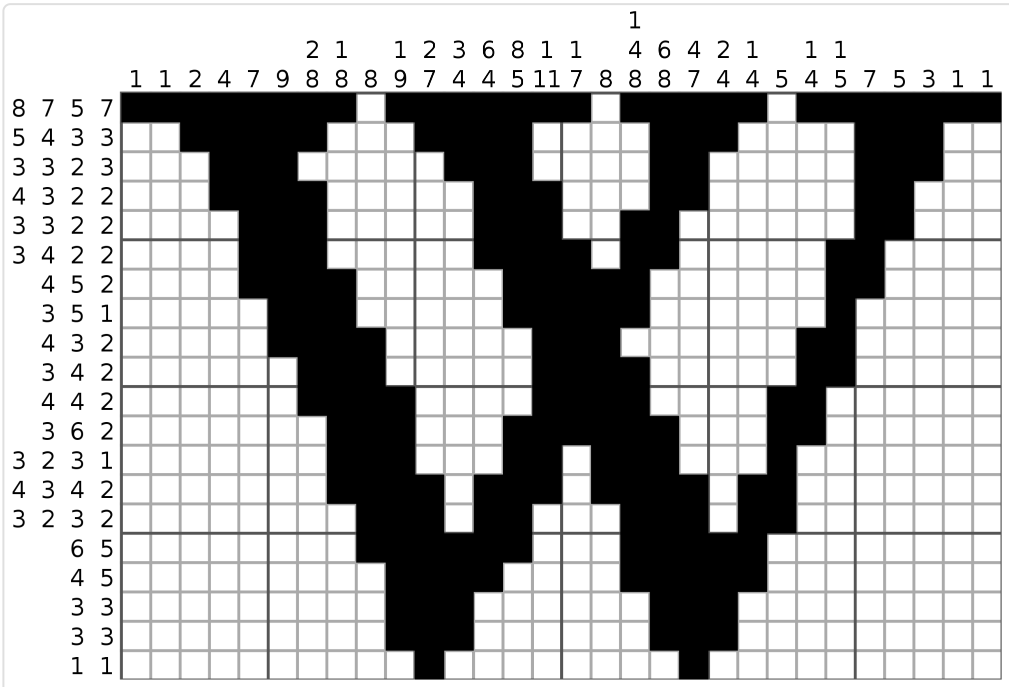
End of Honor Code

Objectives & Intended Learning Outcomes

The objective of this assignment is to provide you with practice on arrays, functions, and recursion. Upon completion of this assignment, you should be able to:

1. Define and use arrays to store data
2. Modularize your program in functions
3. Develop recursive functions to solve computational problems

End of Objectives & Intended Learning Outcomes



"Nonograms" are picture logic puzzles in which cells in a grid must be colored or left blank according to numbers at the side of the grid to reveal a hidden pixel art-like picture.

Source: https://en.wikipedia.org/wiki/Nonogram#/media/File:Nonogram_wiki.svg

Introduction

The goal of this assignment is to implement a text-based "Nonogram" game. You can check this Wikipedia link here for a historical overview of this genre of games.

Menu

Page maintained by

ZHONG, Shuhan
Email: szhongaj@connect.ust.hk
Last Modified: 10/20/2022 14:16:11

Homepage

Course Homepage

Nonogram is like "a combination of Minesweeper" and "Sudoku". Each cell on the board must be colored or left blank according to numbers at the side of the board (we call them "row constraints" and "column constraints") to reveal a hidden pixel art-like picture.

The numbers (i.e., "row constraints" and "column constraints") measures how many unbroken lines of filled-in squares there are in any given row or column. For example, a constraint of "4 8 3" would mean there are sets of four, eight, and three filled squares, in that order, with at least one blank square between successive sets. You may check the banner above to see how constraints work. That banner is a solved nonogram.

In this assignment, you are required to:

- Implement the game mechanisms outlined in Tasks 1 - 4.
- Implement a *solver*, which generate a Nonogram solution (or any, if there are multiple solutions) that satisfies all the constraints (Tasks 5 - 9).

End of Introduction

Description

Please read the FAQ section for some common clarifications. You should check that a day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by then.

Gameplay

During each round, the game interface displays a menu asking the user to select an action:

- `p` (print) - print the current board
- `m` (modify) - user set/unset (color/leave blank) a certain cell
- `c` (check) - check if the current board is a valid solution
- `s` (solve) - invoke Nonogram solver to solve the game
- `q` (quit) - quit the game and ends the program

Solver

You are required to write a solver that (most likely) uses recursion. Please refer to this paper about an overview of how the solver works. You don't need to know what is DFS or backtracking, just reading the process in **III. USING DFS TO SOLVE NONOGRAM PUZZLE** is good enough.

Don't worry if you are still confused about how to implement such an algorithm after reading the paper. We will give you detailed instructions on implementing some helper functions (Task 5 - 8), and we hope they will help you come up with an idea about how to design a solver with the help of them.

During grading process, we ensure the board we input has **only one** valid solution. So you don't need to worry the cases when (1) there is no solution, or (2) there are multiple solutions. When you are implementing the solver, you may choose either to (1) quit after finding one solution, or (2) quit after finding all solutions. However, we suggest you do the former, since if your solver takes too much time during execution, you may lose some scores.

Global variables

Since some variables that representing game states will be used in many functions, we make them global for simplicity. Here are a list of them:

Global Constants: (defined at the beginning)

- `const int MAX_ROW = 15, MAX_COL = 15, MAX_CONSTRAINT_NUM = 15` - maximum number of rows/columns/constraints of the game board in this assignment.
  For game board: (defined at the beginning)
- `int num_rows, num_cols` - number of rows / columns in the game board.
- `char board[MAX_ROW][MAX_COL]` - the game board represented by a 2D character array, `'.'` means the cell is left blank, `'X'` means colored.
- `int num_row_constraints[MAX_ROW], num_col_constraints[MAX_COL]` - number of constraints for each row / column. For example, if row 0 has constraint "4 3 5", then `num_row_constraints[0] = 3`. This actually records the length of `int row_constraints[][], col_constraints[][]` below.
- `int row_constraints[MAX_ROW][MAX_CONSTRAINT_NUM], col_constraints[MAX_COL][MAX_CONSTRAINT_NUM]` - row / column constraints. For each row / column, there are multiple numbers in its constraint, so we need an array to store it. The length of array is stored in `int num_row_constraints[], num_col_constraints[]` introduced above.

  For solver: (defined after Task 6)
- `int num_row_perms[MAX_ROW]` - number of valid permutations for each row.
- `char row_perms[MAX_ROW][MAX_PERM][MAX_COL]` - stores all valid permutations for each row. For example, `char row_perms[i][j][k]` represents for row `i`, in the `j`-th valid permutation, whether the cell at column `k` is colored or left blank.

Additional Remarks

- You are allowed to define your own helper functions, but be careful as ZINC won't be able to know of their existence. This shouldn't be a problem as long as the changes to the game state variables and task function return values are as expected.
- All the task functions are in global scope, and you are allowed to have the task functions call each other for easier implementation. Be careful when two or more functions call each other reciprocally, as they may enter an infinite loop.
- You are free (but not strictly required) to use recursion in any of the task functions. Some recursion hints are given in the tasks, but otherwise the final decision is up to you. Avoid accidental infinite recursion.
- You are only allowed to use one header file `#include <iostream>`.
- **You are strongly discouraged to modify the `main` function given in skeleton.** It will be replaced when we grade your program. **You are strictly forbidden to modify the signature of task functions**. However, you can overload them. When we grade your program, if we cannot find the exact function whose signature is what we described in Tasks section, you will have **no score** for that task. This situation is unlikely to occur if you never modified the `main` function.

End of Description

Tasks

This section describes the functions that you will need to implement. Please refer to the `main` function given in skeleton to see how they are invoked.

Part I: Initialize the board.

Task 1 - Implement the `get_input_board()` function

Description - This function asks user to input the board size and constraints, check if the constraints are valid, store the information in global variables and finally initialize the board.

Read constraints - The function will ask user to input constraints row by row, and then column by column. For each row / column, user need to input a list of numbers, separated by a space, to indicate the constraint for this row / column. Since the constraint can be of any length, the user will enter `-1` at the end, meaning that the constraint for this row / column is ended. If there is no constraint for this row / column, the user should simply enter -1. You may refer to the example below for details.

Validity checking - It is not easy to check if the constraints are valid, for example, you may think if the constraints result in no solution, they are invalid. However, in current stage we are unable to check such complex stuff. So in this function, you only need to check **if the row / column has enough space to possibly hold such constraints**. For example, if a row constraint is "4 3 5", since two adjacent groups require at least 1 blank cell between, we at least need 4 + 1 + 3 + 1 + 5 = 14 cells in this row. Hence, if `num_cols` is smaller than 14, this is invalid. You only need to check this situation.

```
void get_input_board();
```

Notes

- You may assume the user always input a valid integer for number of rows / columns, i.e., an integer between `1` and `MAX_ROW / MAX_COL`.
- You may assume the user always input positive integers for row / column constraints, except for the -1 used to flag input termination.
- You may assume the number of user input row/column constraints never exceeds the number of the row/columns, e.g., if the board size is 3 row by 4 col, the user will input ≤ 3 row constraints for each row and ≤ 4 column constraints.
- If the constraints input by user is not valid, this function will tell the user "Invalid row / column constraint, please try again.", and keep asking user to re-input the constraint.
- Some code snippets have been given for you in the skeleton. You may directly use them for output format.

Example: (user input is highlighted in red) Note that this is just an example, this board may have no solution.

```
Enter the number of rows: 3
Enter the number of columns: 4
Enter the number of constraints for row 0 (end with -1): -1
Enter the number of constraints for row 1 (end with -1): 1 1 -1
Enter the number of constraints for row 2 (end with -1): 1 1 1 -1
Invalid row constraint, please try again.
Enter the number of constraints for row 2 (end with -1): 1 3 -1
Invalid row constraint, please try again.
Enter the number of constraints for row 2 (end with -1): 2 1 -1
Enter the number of constraints for column 0 (end with -1): 1 1 1 -1
Invalid column constraint, please try again.
Enter the number of constraints for column 0 (end with -1): 1 2 -1
Invalid column constraint, please try again.
Enter the number of constraints for column 0 (end with -1): 2 -1
Enter the number of constraints for column 1 (end with -1): 1 -1
Enter the number of constraints for column 2 (end with -1): -1
Enter the number of constraints for column 3 (end with -1): 1 1 -1
[Then this function ends, program will print the board. Remaining parts
are ignored.]
```

Task 2 - Implement the `print_board()` function

Description - This function prints the board and the constraints for each row and column. Constraints should be printed at the bottom of each column (top-aligned) and on the left of each row (right-aligned). We will also output row / column index for each row / column. Row indices are numbered in 0, 1, 2, ..., while column indices are numbered in A, B, C, ... Please note that to distinguish row indices and row constraints, we will print an extra `|` between each row index and its constraints. You may refer to the example below for details.

```
void print_board();
```

Notes

- Each cell, or each number in constraint, or each index in the same row is separated by a space, as shown in example below.
- Each cell, or each number in constraint, or each **column** index occupies one character. However, each row index occupies two characters, and they should be right-aligned, as shown in example below.
- It's ok to have trailing spaces at the end of each line. We will ignore them during grading process.
- You may assume constraints will **never contain numbers larger than 9**. So they are always a single digit.
- You may also assume there will be **no more than `MAX_ROW=15` rows and `MAX_COL=15` columns**. The column indices are single characters ranging from `A` to `O`. However, there **can be more than 9 rows**, that's why we said each row index needs to take 2 characters.

Example: (user input is highlighted in red) Note that this is just an example, this board may have no solution.

```
Enter the number of rows: 5
Enter the number of columns: 5
Enter the number of constraints for row 0 (end with -1): 2 2 -1
Enter the number of constraints for row 1 (end with -1): 2 2 -1
Enter the number of constraints for row 2 (end with -1): -1
Enter the number of constraints for row 3 (end with -1): 1 1 -1
Enter the number of constraints for row 4 (end with -1): 3 -1
Enter the number of constraints for column 0 (end with -1): 2 1 -1
Enter the number of constraints for column 1 (end with -1): 2 1 -1
Enter the number of constraints for column 2 (end with -1): 1 -1
Enter the number of constraints for column 3 (end with -1): 2 1 -1
Enter the number of constraints for column 4 (end with -1): 2 1 -1
         A B C D E
2 2 |  0 . . . . .
2 2 |  1 . . . . .
    |  2 . . . . .
1 1 |  3 . . . . .
  3 |  4 . . . . .
         2 2 1 2 2
         1 1   1 1
[Then program will print menu, as shown in main function. Remaining parts
are ignored.]
```

Another example: when there are more than 9 rows. Here user inputs are ignored and only shows the board.

```
             A B C D E F G H I J K L
2 1 2 4 |  0 . . . . . . . . . . . .
    5 2 |  1 . . . . . . . . . . . .
1 1 3 1 |  2 . . . . . . . . . . . .
  2 2 3 |  3 . . . . . . . . . . . .
  4 1 3 |  4 . . . . . . . . . . . .
  1 1 8 |  5 . . . . . . . . . . . .
2 1 1 1 |  6 . . . . . . . . . . . .
1 2 2 3 |  7 . . . . . . . . . . . .
    5 4 |  8 . . . . . . . . . . . .
  2 1 1 |  9 . . . . . . . . . . . .
    2 6 | 10 . . . . . . . . . . . .
3 1 1 3 | 11 . . . . . . . . . . . .
             1 1 2 2 3 4 3 2 1 6 2 1
             5 2 2 1 6 1 3 1 1 2 7 2
             1 3 1 3   1 1 3 1 2 1 2
               2 1 1   2     1     1
                             1
```

Part II: User play on board.

Task 3 - Implement the `user_operate_board()` function

Description - This function will be called when user choose to set/unset a cell. It will:

1. Ask user to input which cell he/she wants to modify. The user will need to first enter the column index, followed by a space, and then the row index. For example, "A 2". You may always assume user will first enter a capital letter, followed by a space, and then an integer.
2. Check if the user input is a valid cell(i.e., it is within the board). If invalid, keep asking the user to input until a valid cell is received.
3. Set / unset (color / leave blank) the cell. If the cell was set before, then unset it, and vice versa.

```
void user_operate_board();
```

Example: (user input is highlighted in red, and original board is highlighted in blue.) Note that this is just an example, this board may have no solution.

```
         A B C D E
2 2 |  0 . . . . .
2 2 |  1 . . . . .
    |  2 . . . . .
1 1 |  3 . . . . .
  3 |  4 . . . . .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A -1
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): A 5
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): F 3
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): Z 100
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): B 2
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): B 2
[Then program will print menu. Remaining parts are ignored.]
```

Task 4 - Implement the `check_whole_board_valid()` function

Description - This function will be called after user finish filling the whole board. You need to check whether his/her solution is correct, i.e., satisfy all constraints. You don't need to print the result, it has been handled in `main()` function in the skeleton.

```
bool check_whole_board_valid();
```

Return value - `true`, if the solution is correct, `false`, otherwise.

Example: (user input is highlighted in red, and original board is highlighted in blue.)

```
          A B C D E
2 2 |   0 X X . X X
2 2 |   1 X X . X X
    |   2 . . . . .
1 1 |   3 X . . . X
  3 |   4 . X X X .
          2 2 1 2 2
          1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Congratulations! Your solution is correct!
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Ahh, your solution is incorrect, try again.
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 3
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Ahh, your solution is incorrect, try again.
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 3
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Congratulations! Your solution is correct!
```

Part III: A nonogram solver.

Please make sure you have read the paper in description section about how we are going to implement the solver.

Basically, the steps are:

1. For each row, choose one permutation and fill it on the board
2. Check if current board does not violate column constraints. (It cannot violate row constraints, since we used a valid row permutation)
   - If not violate, proceed on next row, until all rows are filled
   - If violate, go to step 1, choose another permutation

Now the problems are: (1) how to generate permutations for each row, and (2) how to check a board does not violate column constraints. Notice here the board is incomplete.

For generating permutations, we can first generate a left-aligned permutation as a starter. For example, if the row constraint is `{2, 1, 3}`, and there are 12 columns, then a left-aligned permutation is: `[X X . X . X X X . . . .]`, i.e., start filling cells from left most, and separate each group by only one empty cell.

Then, to generate other permutations based on the starter, we check if any group can be shifted to the right. In the example above, the last group can be shifted to the right by 1, 2, 3, or 4 cells (more shifts will hit the boundary), which will produce 4 other permutations. And after the last group is shifted, you may also check if the second group can be shifted. For example, when the last group is shifted by 2 cells, the second group can be shifted to the right by 1 or 2 cells. (more shifts will hit the last group) The same process should be done on the first group. In this way you are able to generate all permutations for a row.

Task 7 asks you to generate all permutations for a certain row, and **Task 6** asks you to check if a certain group can be shifted, as we described above. **Task 5** acts as a useful helper function, since it's convenient for us to only focus on *which column does each group start*, rather than how the row really looks like.

For checking whether current board violates column constraints, it is similar to what we have done in `check_whole_board_valid()`, except for

- We don't need to check the row constraints, since the row permutations we generated must not violate row constraints.
- We don't check the whole board. The board is not complete yet, we only need to check the rows that we have finished.

You will be asked to implement this function in **Task 8**.

Finally, you will be asked to implement the complete solver in **Task 9**. We expect you to make use of those previous functions that you have written.

Task 5 - Implement the `positions_to_row()` function

Description - This function coverts a **position vector** of a row into a **real row**, where

- A **position vector** is a 1D integer array, with known length. It stores the beginning indices for each colored group in the row. For example, if a row has 2 constraints, we expect there are 2 colored groups, and the position vector will record at which column do the 2 groups starts.
- A **real row** is a 1D character array. If a **position vector** is given, along with the row constraints, you will be able to figure out how the row looks like.
- For example, if `num_col = 8, position vector = {0, 3, 5},` row_constraint for this row is `{2, 1, 1}`, then the row should be: `[X X . X . X . .]`
  Explanation: there are 3 groups in this row, since there are 3 numbers in row constraint. Hence the position vector should have 3 elements. The first `0` means the 1st group starts at column 0, and since we know the first constraint is 2, so column 0 and column 1 must be colored. Similarly, the `3` in the position vector means the 2nd group starts at column 3, and since we know the second constraint is 1, so only column 3 is colored.

```
void positions_to_row(int row_idx, const int positions[], int num_pos, char result_row[]);
```

Parameters

- `int row_idx` - which row we are working on.
- `int positions[]` - the position array given.
- `int num_pos` - the length of the given position array.
- `char result_row[]` - you need to store your converted real row in this array. Note that you don't need to specify the length, it must have the length of `num_cols`.

Notes

- You may assume the parameters passed into this function are always valid *during grading process*. However, since this function is used as a helper function in later tasks, you need to be careful that you always pass in valid parameters when you invoke it in other functions.

Task 6 - Implement the `block_can_shift()` function

Description - This function checks if the given block on given row can be shifted to the right for one cell.

For example, if the position vector is `{0, 4, 6}`, `num_col = 8`, row_constraint for this row is `{2, 1, 1}`, then the row is: `[X X . . X . X .]`, and there are 3 blocks. Then,

- `block_can_shift(row_idx, 0, {0, 4, 6}) = true,`
- `block_can_shift(row_idx, 1, {0, 4, 6}) = false,` since it will hit 2nd block after shift
- `block_can_shift(row_idx, 2, {0, 4, 6}) = true,`

```
bool block_can_shift(int row_idx, int block_idx, const int positions[], int num_pos)
```

Parameters

- `int row_idx` - which row we are working on.
- `int block_idx` - which block we are going to shift.
- `int positions[]` - the position array given.
- `int num_pos` - the length of the given position array.

Return value - `true`, if the block can be shifted to the right for one cell; `false`, otherwise.

Notes

- You may assume the parameters passed into this function are always valid *during grading process*. However, since this function is used as a helper function in later tasks, you need to be careful that you always pass in valid parameters when you invoke it in other functions.

Task 7 - Implement the `get_row_perms()` function

Description - This function gets all valid permutations for a row, and store all results in global variables `row_perms` and `num_row_perms`. See description section for explanations about global variables used here.

Hint: you may use recursion. First generate a starter permutation, then try to shift each of the groups on that row, in order to produce other permutations. See instructions before Task 5 for more details.

```
void get_row_perms(int row_idx);
```

Parameters

- `int row_idx` - which row we will generate permutations for.

Notes

- You may assume the parameters passed into this function are always valid *during grading process*. However, since this function is used as a helper function in later tasks, you need to be careful that you always pass in valid parameters when you invoke it in other functions.

Task 8 - Implement the `check_rows_valid()` function

Description - This function checks if current state is valid, after we finish filling `num_complete_rows` rows. For example, if `num_complete_rows = 2`, it will only check if the first two rows (with index 0 and 1) do not break column constraints.

As we have discussed before, this function is similar to what we have done in `check_whole_board_valid()`, except for

- We don't need to check the row constraints, since the row permutations we generated must not violate row constraints.
- We don't check the whole board. The board is not complete yet, we only need to check the rows that we have finished.

```
bool check_rows_valid(int num_complete_rows);
```

Parameters

- `int num_complete_rows` - how many rows have we completed. We will only check those rows.

Return value - `true`, if the current board state is valid; `false`, otherwise.

Notes

- You may assume the parameters passed into this function are always valid *during grading process*. However, since this function is used as a helper function in later tasks, you need to be careful that you always pass in valid parameters when you invoke it in other functions.

Task 9 - Implement the `solve()` function

Description - This function will solve the board, and store the solution into global variable `board` directly. You may assume there are only one solution for the board given.

Hint: you may use recursion. Choose a permutation for first row, check if the current board is valid: if so, proceed on next row, otherwise, choose another permutation. End recursion when we have finished filling the last row. See instructions before Task 5 for more details.

```
void solve();
```

Notes

- You may assume the given game has only one solution *during grading process*. However, when you are testing your own board / constraints, the board may have 0 / 1 / more than 1 solutions.

Example: (user input is highlighted in red, and original board is highlighted in blue.)

```
         A B C D E
2 2 |  0 . . . . .
2 2 |  1 . . . . .
    |  2 . . . . .
1 1 |  3 . . . . .
  3 |  4 . . . . .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: s
Generating solution:
         A B C D E
2 2 |  0 X X . X X
2 2 |  1 X X . X X
    |  2 . . . . .
1 1 |  3 X . . . X
  3 |  4 . X X X .
         2 2 1 2 2
         1 1   1 1
```

End of Tasks

Resources & Sample I/O

- Skeleton code: download here
- Demo programs: Windows / MacOS Intel / MacOS Apple Silicon / Linux
- Sample program outputs: (user input is highlighted in red) - Download sample input, sample output.
  You may use

  ```
  g++ -o main main.cpp -Wall -std=c++11
  ./main < sample_input.txt > my_output.txt
  ```

  to let your program reads input from `sample_input.txt`, and output results to `my_output.txt`. Then you only need to compare `my_output.txt` with `sample_output.txt`. Here is an online file compare tool.

```
Enter the number of rows: 5
Enter the number of columns: 5
Enter the number of constraints for row 0 (end with -1): 2 2 2 -1
Invalid row constraint, please try again.
Enter the number of constraints for row 0 (end with -1): 2 2 1 -1
Invalid row constraint, please try again.
Enter the number of constraints for row 0 (end with -1): 2 1 1 -1
Invalid row constraint, please try again.
Enter the number of constraints for row 0 (end with -1): 2 2 -1
Enter the number of constraints for row 1 (end with -1): 2 2 -1
Enter the number of constraints for row 2 (end with -1): -1
Enter the number of constraints for row 3 (end with -1): 1 1 -1
Enter the number of constraints for row 4 (end with -1): 3 -1
Enter the number of constraints for column 0 (end with -1): 2 2 1 -1
Invalid column constraint, please try again.
Enter the number of constraints for column 0 (end with -1): 2 1 -1
Enter the number of constraints for column 1 (end with -1): 2 1 -1
Enter the number of constraints for column 2 (end with -1): 1 -1
Enter the number of constraints for column 3 (end with -1): 2 1 -1
Enter the number of constraints for column 4 (end with -1): 2 1 -1
         A B C D E
2 2 |  0 . . . . .
2 2 |  1 . . . . .
    |  2 . . . . .
1 1 |  3 . . . . .
  3 |  4 . . . . .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: p
         A B C D E
2 2 |  0 . . . . .
2 2 |  1 . . . . .
    |  2 . . . . .
1 1 |  3 . . . . .
  3 |  4 . . . . .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 2
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 8
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): F 7
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): F 0
Invalid row or column. Try again.
Enter the cell you want to modify (e.g. A 2): E 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: p
         A B C D E
2 2 |  0 X . . . X
2 2 |  1 . . . . .
    |  2 X . . . .
1 1 |  3 . . . . .
  3 |  4 . . . . .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Ahh, your solution is incorrect, try again.
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: s
Generating solution:
         A B C D E
2 2 |  0 X X . X X
2 2 |  1 X X . X X
    |  2 . . . . .
1 1 |  3 X . . . X
  3 |  4 . X X X .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Congratulations! Your solution is correct!
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): A 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Ahh, your solution is incorrect, try again.
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): E 0
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: m
Enter the cell you want to modify (e.g. A 2): E 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: c
Ahh, your solution is incorrect, try again.
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: s
Generating solution:
         A B C D E
2 2 |  0 X X . X X
2 2 |  1 X X . X X
    |  2 . . . . .
1 1 |  3 X . . . X
  3 |  4 . X X X .
         2 2 1 2 2
         1 1   1 1
===== Welcome to Nonogram Game =====
Please enter your choice:
Enter 'p' to print the current board.
Enter 'm' to modify a cell.
Enter 'c' to check your solution.
Enter 's' to invoke solver.
Enter 'q' to quit.
Your choice: q
Bye!
```

End of Resources & Sample I/O

Submission & Grading

Deadline: Sat, 1/10/22 HKT 23:59.
Compress the single source code file `main.cpp` by itself as `PA3.zip` for submission to the ZINC Autograding System. The ZINC submisssion portal is available.

Late Policy:
Please refer to the 3-day late budget policy.

You are strongly discouraged to modify the `main` function given in skeleton. It will be replaced when we grade your program. **You are strictly forbidden to modify the signature of task functions**. However, you can overload them. When we grade your program, if we cannot find the exact function whose signature is what we described in Tasks section, you will have **no score** for that task. This situation is unlikely to occur if you never modified the `main` function.

Before deadline, the test cases on ZINC will only contain a few cases. If you passed those tests, it only means your program can successfully run on ZINC and can pass those test cases. The result is by no means complete and the score are irrelavent to your actual score. In actual grading stage after deadline, we will use a **totally different set of test cases**, which is expected to be more complete and more strict, to test the correctness your program.
You are hence strongly suggested to test your program thoroughly by thinking about all possible cases.

Grading Scheme

The actual grading will use Unit Testing, i.e., we will test each functions individually, invoke the function you implemented, and check the return value/expected behavior of that function. The actual grading will also only be triggered, with the scores and test cases revealed, after the deadline. This hidden grading policy is for all the PAs in order to prevent reverse-engineering of the test cases, since the PAs are a significant part of the course assessment and course grade.

We will execute unit testing on the task functions individually, and some of the test cases may include invalid input values. In general, you can assume that:

- As long as we are testing solver part, i.e. Task 5 - Task 9, you can assume the parameters passed into those functions are always valid during unit testing. And you can also assume there is **only one final solution** for the board and constraints given.
- For other input parameters, if you are required to check the validity of parameters, we have explicitly told you what to check in Tasks section. You don't need to consider other cases.

Additional remark for solver: You are strongly discouraged to use another approach to implement solver instead of using the helper functions we described in Task 5 - 8. If you want to do so, you are also required to complete Task 5 - 8 to get scores for those tasks, even if your solver never uses them. Otherwise, even your solver is correct, you will only get the score for Task 9. Please note it's expected that the solver only worth a small part of scores, and Task 5 - 8 should worth (more likely) more scores.

End of Submission & Grading

Bonus

Nonogram Puzzles are considered as NP-Complete problems which cannot be completed in polynomial time. And the solution described in this assignment uses recursion and depth-frist search to traverse through all possibilities to find the answer. It has a high time complexity of `O(n^2*logn)`.
In Task 9 of this assignment, we also encourage you to optimize your code to achieve faster solution speed. Your programs will be evaluated on a separate set of randomly generated testcases that is expected to be **harder to solve**, and ranked according to the time spent. Up to 10 bonus points will be given according to the ranking.

- Your last submission on ZINC will be used for the ranking. No separate submission will be accepted.
- Late submission will be excluded from the ranking.
- Programs that produce wrong solutions in Task 9 will be excluded from the ranking.
- Programs that run slower than our sample solution will be excluded from the ranking.
- You can use `time path/to/your/program < path/to/your/input.txt` on Linux and Mac to know the time spent by your program.

End of Bonus

Frequently Asked Questions

Q: My code doesn't work, there is an error/bug, here is the code, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own and we should not finish the tasks for you. We are happy to help with explanations and advice, but we shall not directly debug the code for you.

Q: The demo program enters an infinite loop when given unexpected input (e.g. inputting a character when expecting an integer). Is this a bug?
A: This is just the behavior of `cin >> variable;` when given input that is not type-matched. You don't need to worry about such for PA3, you can assume the user always input the correct data type. For example, if we want an `int`, the user will always input an `int`, not other weird things. However, the user input can still be invalid, for example, you want an integer between 1800-9999, and the user may input 1700, or 12345, etc.

Q: What are the restrictions regarding modifying the header files, writing our own helper functions, including extra header files, etc.?
A: The only hard restriction is that you can only submit `main.cpp` to ZINC and can only use one header file `#include <iostream>`. Anything else that you do, while not strictly prohibited, **will be at your own risk regarding the PA3 grading result**. Please keep in mind that there is a grade penalty for all grade appeals that include modifications to your already submitted code (no matter how trivial the modification is).

Q: Am I allowed to use local function declarations (function declaration inside an existing function) for my helper functions?
A: You are strongly discouraged from doing so, as that "feature" is a leftover merely for backwards compatibility with C. In C++, it is superseded with class functions and lambda functions, which will be taught later in this course.

End of FAQ

Changelog

23:37 15/9/22

- Added description of `MAX_ROW, MAX_COL, MAX_CONSTRAINT_NUM`, please check it Here.

- Added notes to the assumption on user input row/col constraints in Task 1, please check the third note Here.
- Corrected misdescriptions on the number of rows/cols in Task 2, please check the 5th note Here.

16:51 16/9/22

- Corrected misdescriptions in the Global variables section:
  ~~`'*'` means colored.~~
  `'X'` means colored.

22:00 19/9/22

- Announcement of Bonus.
- The PA3 submission portal is available on ZINC.
- In Task 3:
  ~~Print the new board.~~
  Now, you are **NOT** supposed to print the new board every time the Task 3 function is called. This is to avoid you losing points in Task 3 because of wrongly implemented Task 2.
  Please download the updated demo executables Here.

End of Changelog