ECE 220

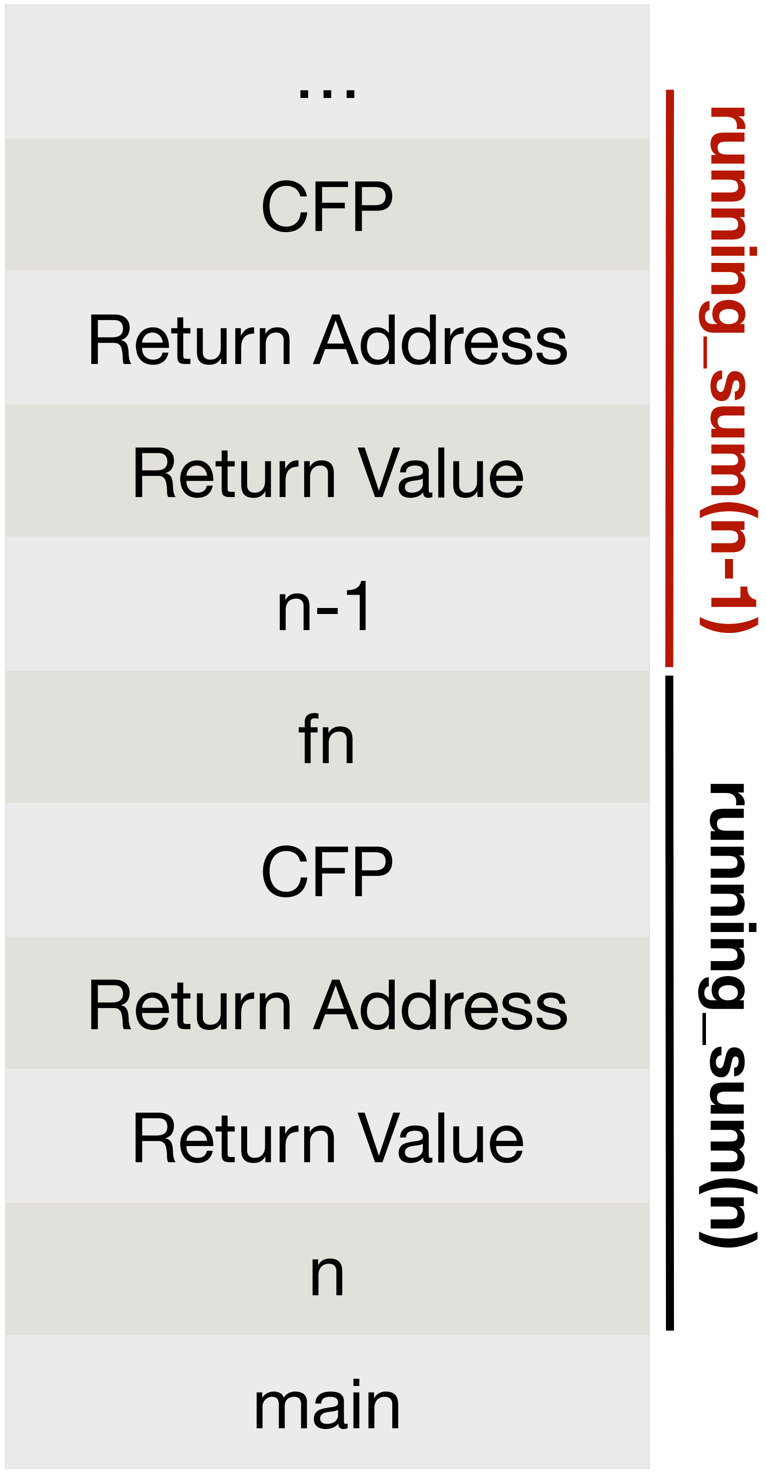
Lecture x000D - 02/29

Recap

- Formal introduction to recursion
    - Factorial
    - Binary search
    - Towers of Hanoi
    - LC3 implementation
- Today: More recursion & problem solving
    - N - Queens problem
    - Maze solving
    - Exercise(s)

Quick review

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 4;
    running_sum(4);
}
```



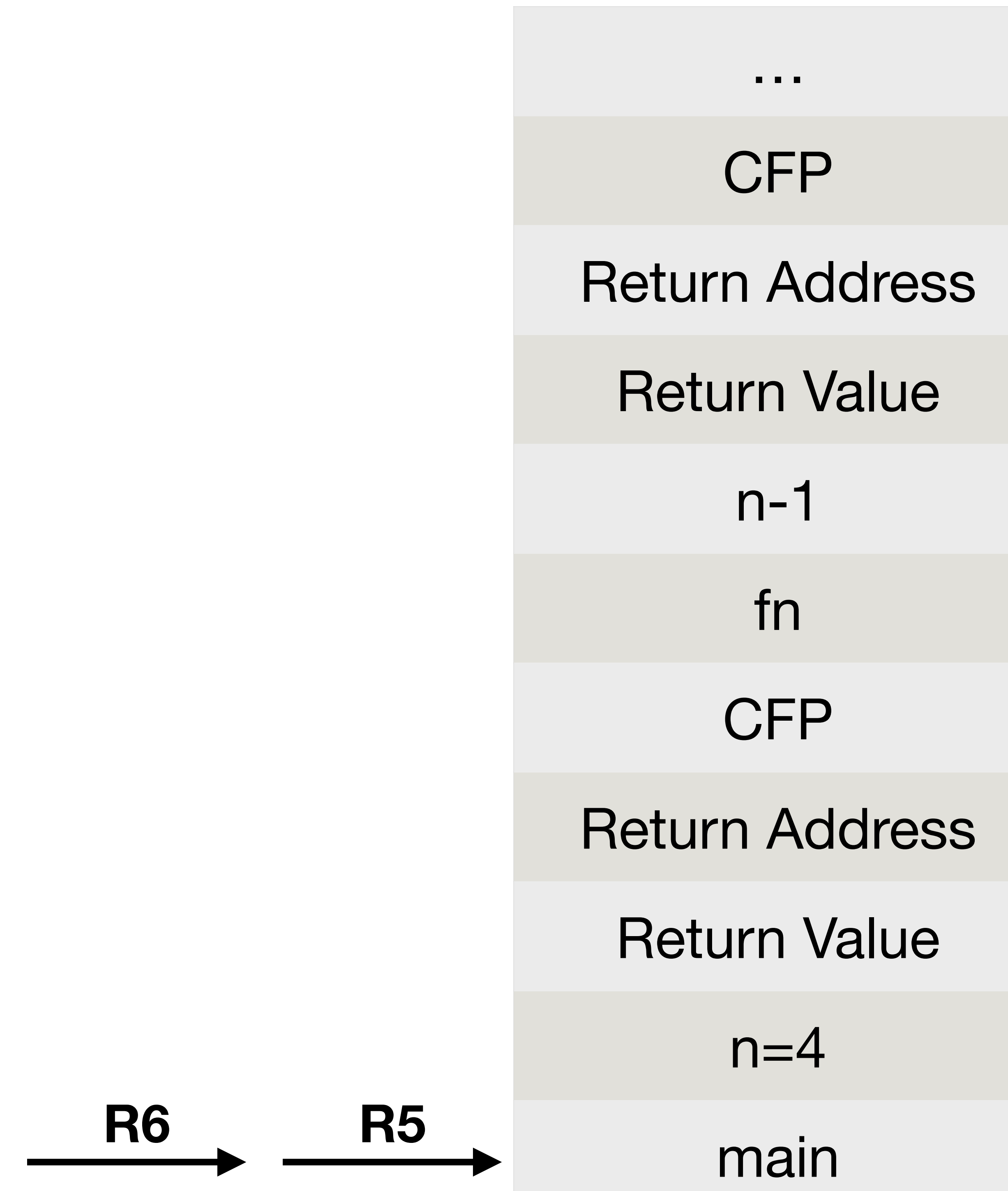
Gitlab C2L3 steps

Review

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
;Caller set-up for Running(n)
STR R0, R5, #0 ; R5 points to main's first local
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1 ;return address
STR R5, R6, #0 ;CFP

;update frame pointer & make space for local variable
ADD R5, R6, #-1 ;step 6 on Gitlab
ADD R6, R6, #-1 ;step 7 on Gitlab

;function logic
;base case (n==1)
LDR R1, R5, #4
ADD R2, R1, #-1
BRz BASE_CASE
```



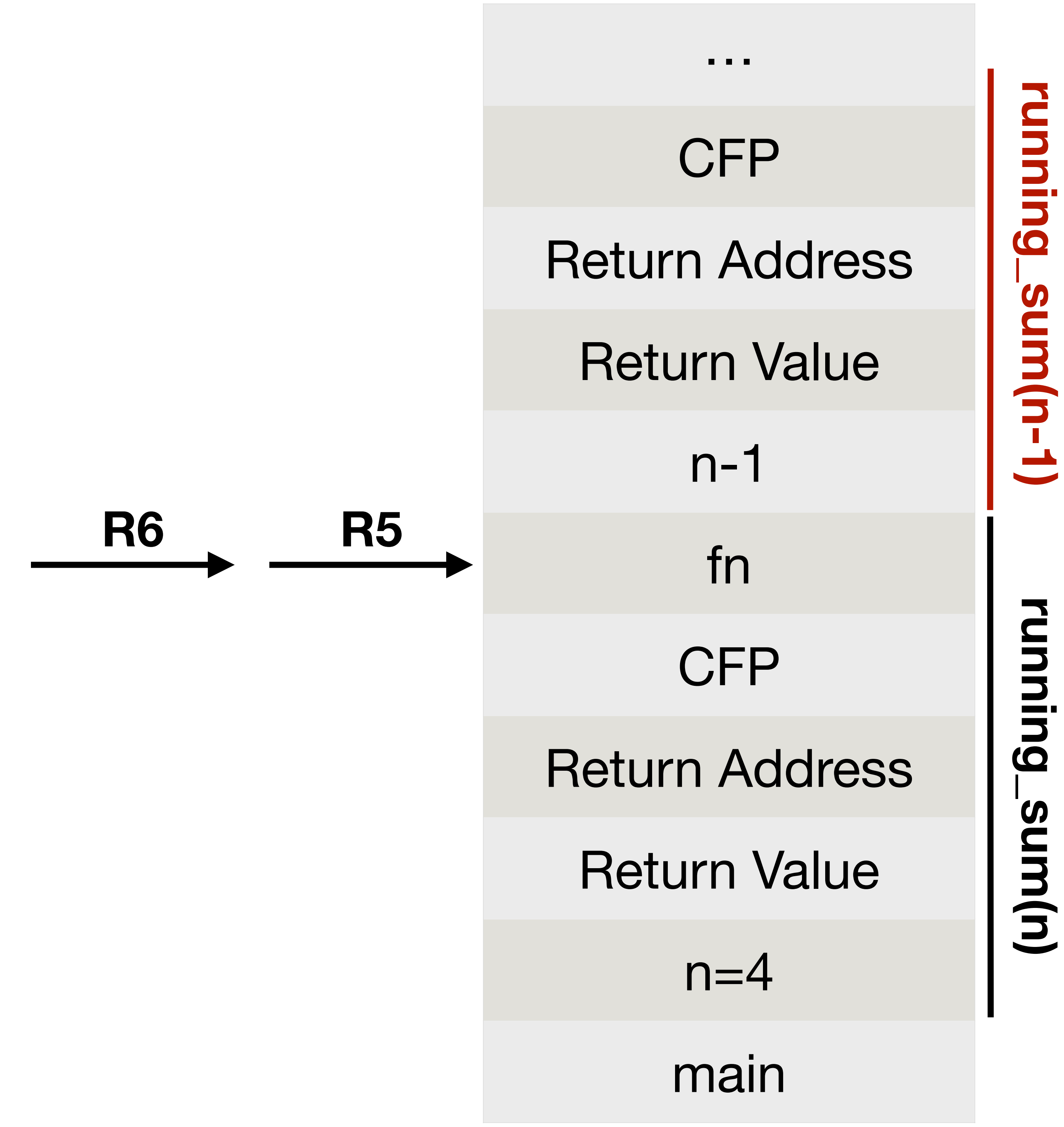
Gitlab C2L3 steps

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Review

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
;Recursive case
;Caller setup for Running(n-1): push argument n-1 onto RST
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)

;Callee tear-down for Running(n-1) not shown

;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
LDR R1, R5, #4
ADD R0, R1, R0
STR R0, R5, #0 ;store result in fn

;ready to return
BRnzp RETURN
```



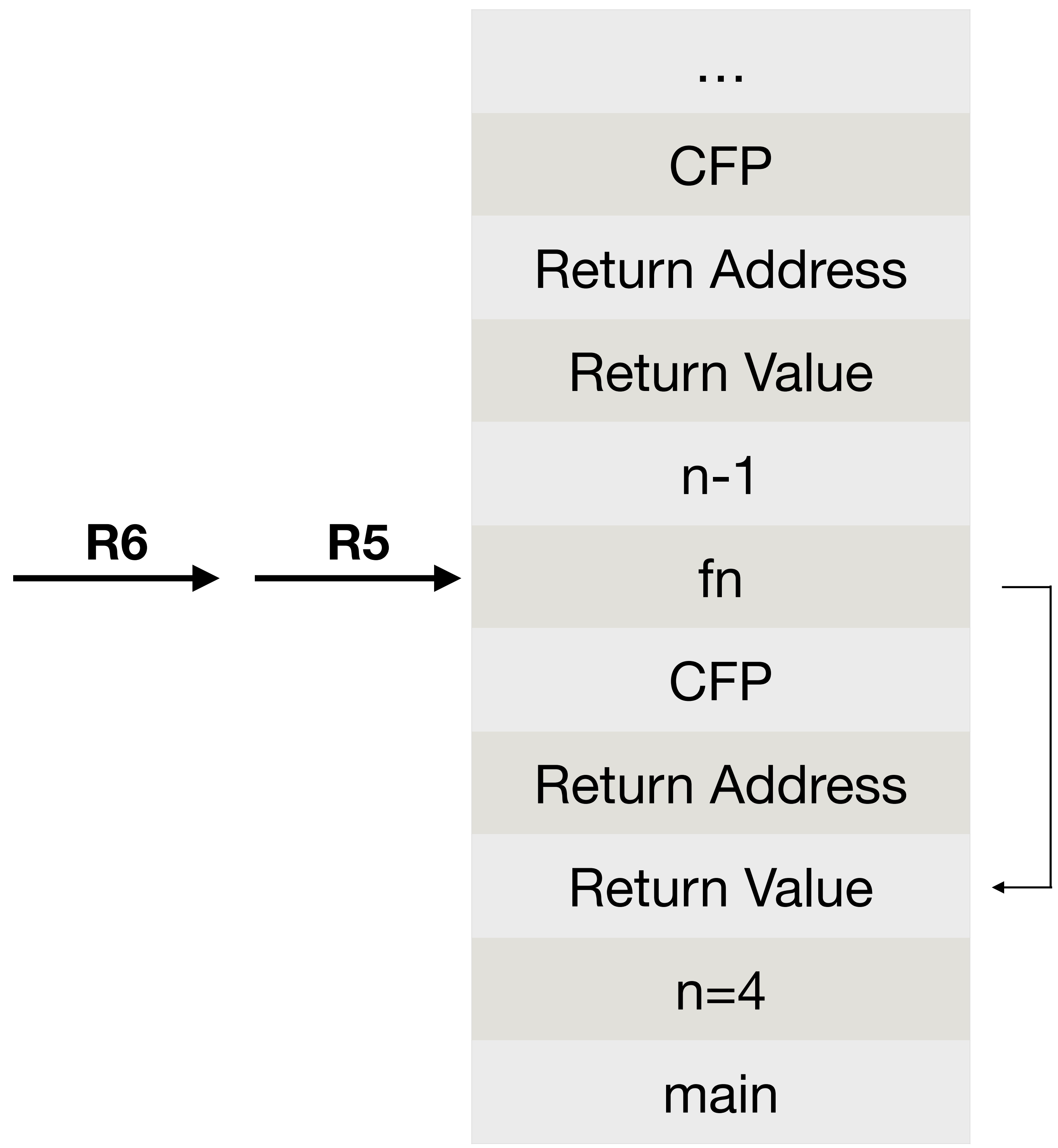
Gitlab C2L3 steps

Review

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R6, #1 ;pop local variables

;restore caller's frame pointer and return address
LDR R5, R6, #0 ; restore CFP
ADD R6, R6, #1
LDR R7, R6, #0 ; prime R7 for RET
ADD R6, R6, #1

;return to caller
RET
```



Gitlab C2L3 steps

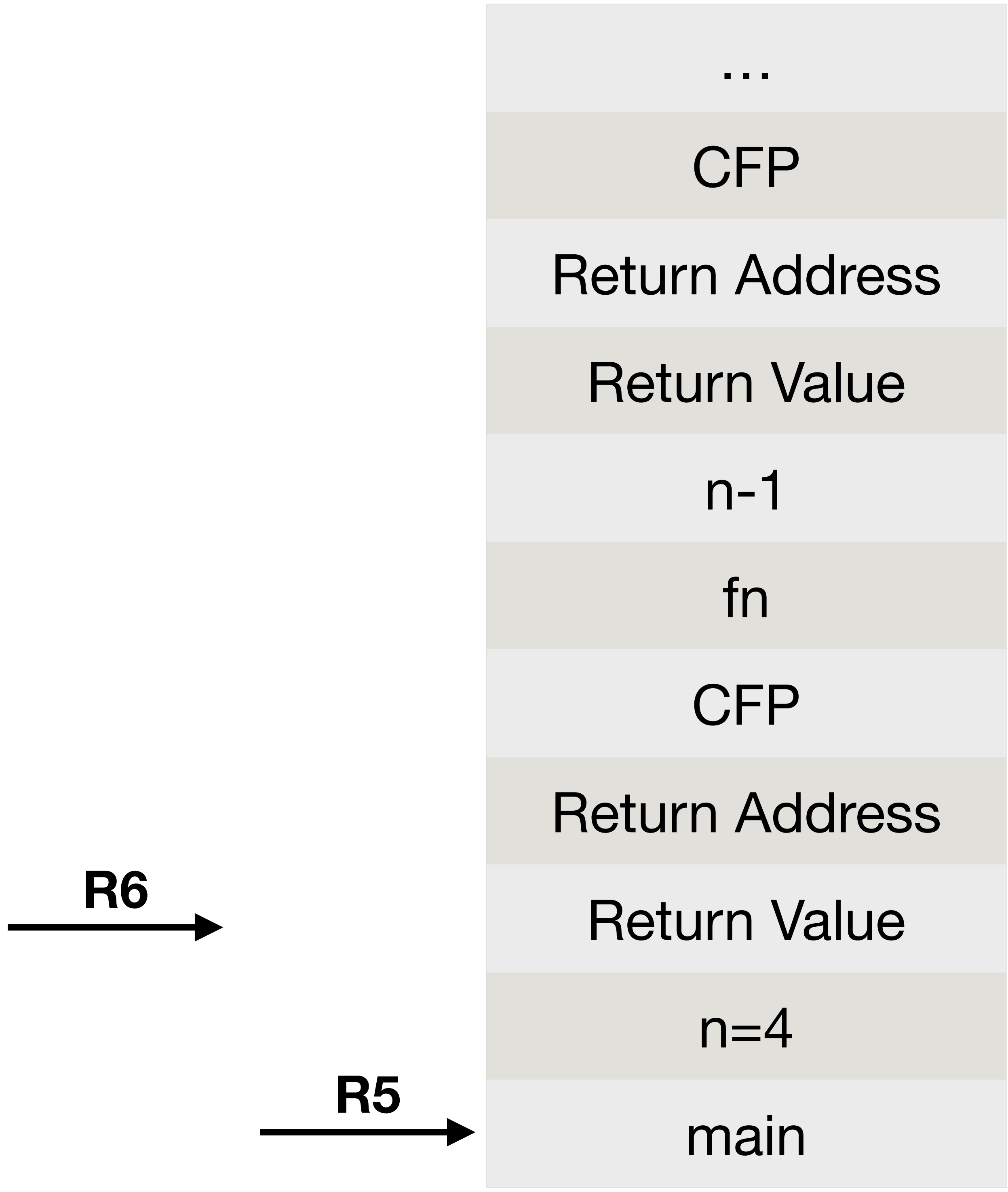
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

Review

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0   ;copy return value to R0
STR R0, R5, #-1  ;save return value to answer
ADD R6, R6, #1   ;pop return value from stack
ADD R6, R6, #1   ;pop argument from stack
```

Inside main's activation frame, answer is the second local variable



Practice practice practice!

Back to where we started!

Gitlab C2L3 steps

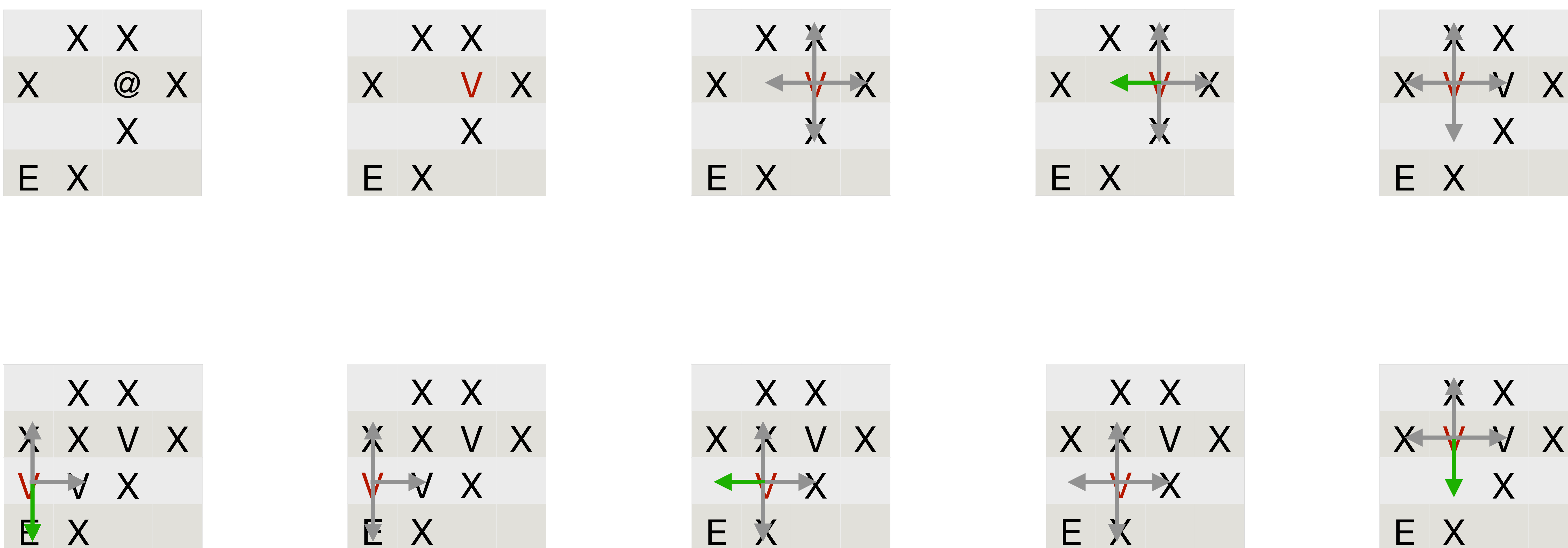
Solving a maze

- We represent a maze by a 2D grid of size $N \times M$
- Walls are marked with X and the exit with E.
- Given starting point (i, j) marked with @, find a path to E (if it exists).
    - Do not go outside grid
    - Avoid going around in circles.
    - Mark valid path with P.

	X	X	
X		@	X
		X	
E	X		

Solving a maze

Strategy: Mark current cell as visited and explore solution space. Exploration defined by four possible moves (U, D, L, R).



Solving a maze

- What should be the base case?
  - Found exit (return “good”) or hit X or hit V or out-of-bounds (return “bad”)
  - Let `xpos` and `ypos` be the *row* and *column* index.

```
if (xpos < 0 || xpos >= MAZE_WIDTH || ypos < 0 || ypos >= MAZE_HEIGHT)
  return 0;

if (maze[xpos][ypos] == 'E')          // Found the Exit!
  return 1;

if (maze[xpos][ypos] != ' ')          // Space is not empty (possibly X or V)
  return 0;
```

Solving a maze

- What should be the recursive call?
  - Go down, up, left or right.
  - `ExitMaze` is the function exploring the solution space.

```
// Go Down
if (ExitMaze(maze, xpos + 1, ypos)) {
  maze[xpos][ypos]='P';
  return 1;
}
```

```
// Go Up
if (ExitMaze(maze, xpos - 1, ypos)) {
  maze[xpos][ypos]='P';
  return 1;
}
```

```
// Go Right
if (ExitMaze(maze, xpos, ypos + 1)) {
  maze[xpos][ypos]='P';
  return 1;
}
```

```
// Go Left
if (ExitMaze(maze, xpos, ypos - 1)) {
  maze[xpos][ypos]='P';
  return 1;
}
```

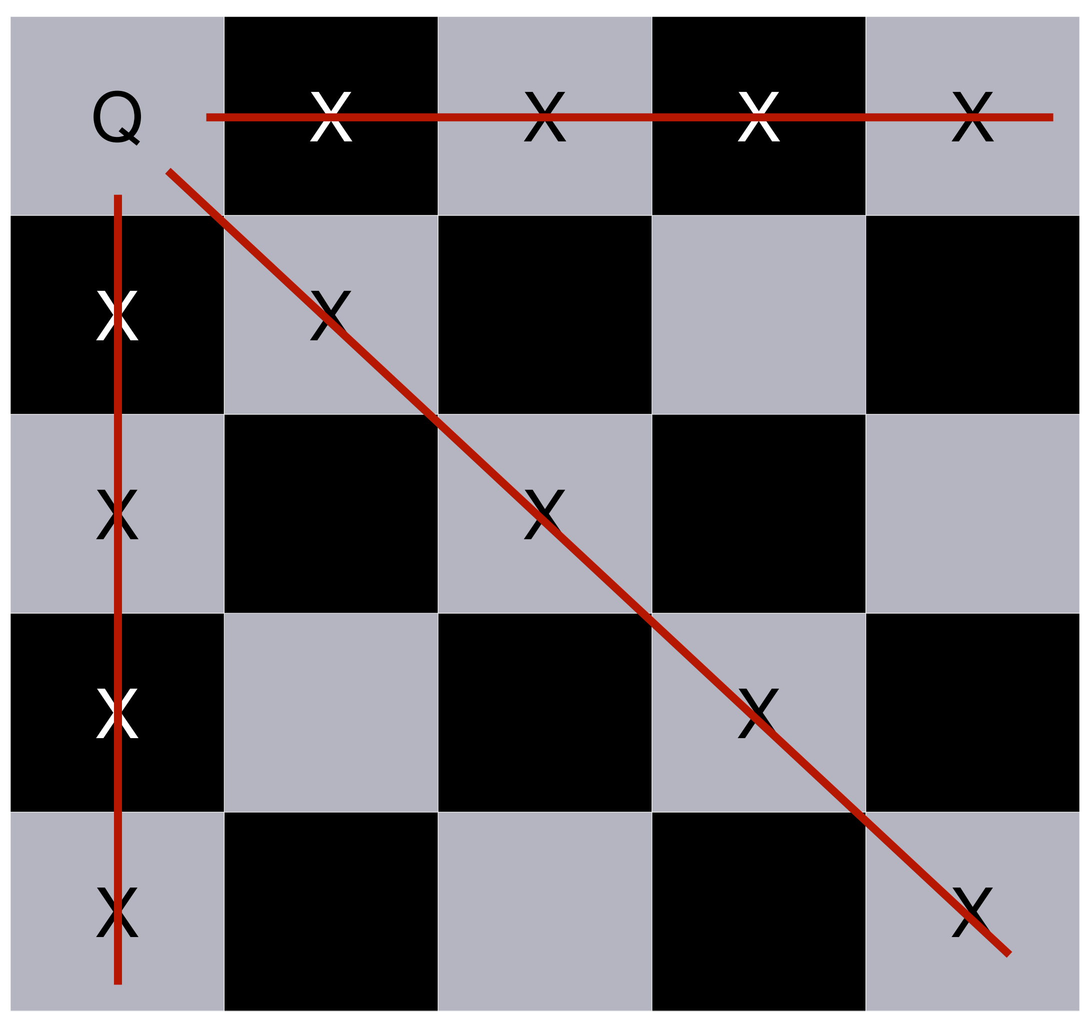
Exercise

- There is an `ExitMaze` function on Gitlab which I tested to work.
- Modify it by adding a `main` function, board definition and try it on this maze.

	X					X
	X			X		
			@	X	X	X
	X					
	X	X	X	X		
X			E	X		
X	X		X	X		X
						X

N - Queens Problem

- In chess, a Queen can attack another piece within its line of sight as long as that piece is in the same: **row**, **column** or **diagonal**.

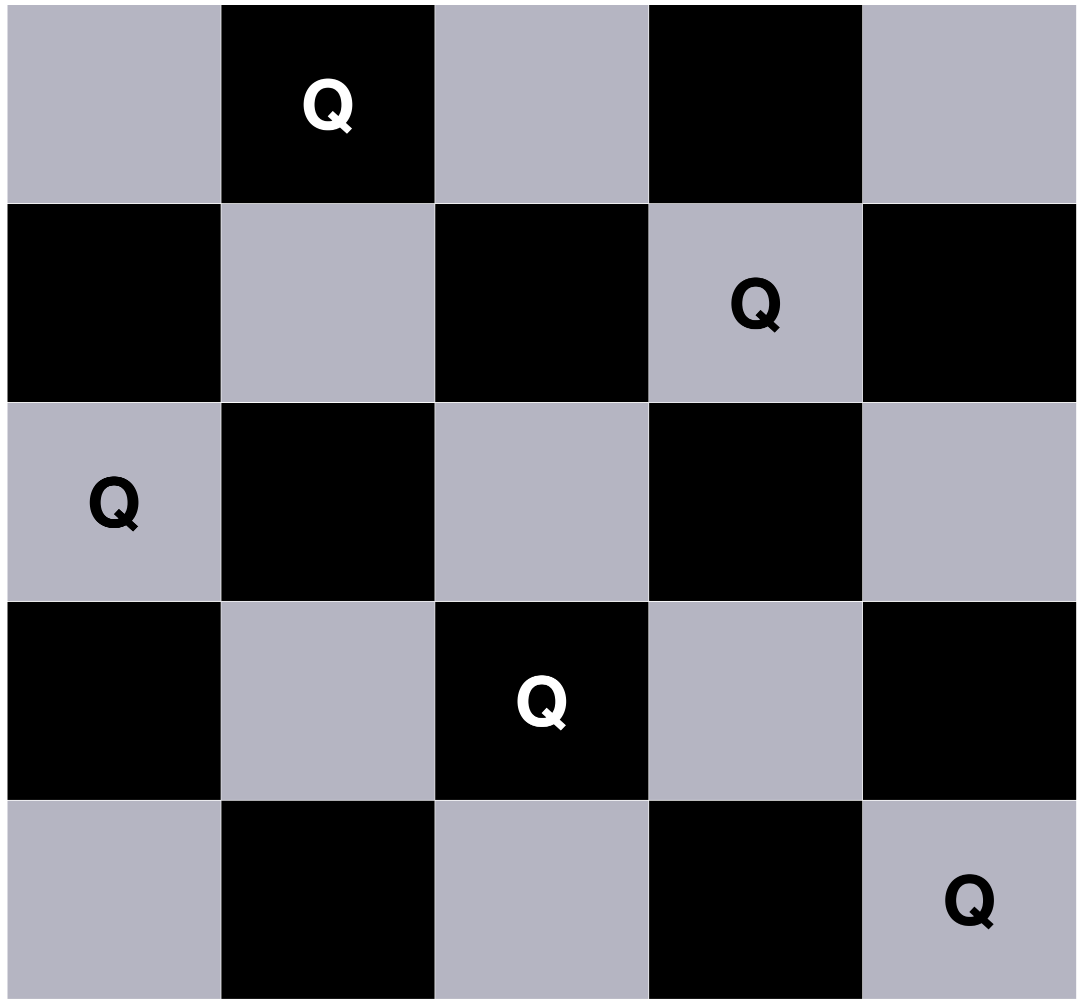


- **Question**: Given an N x N grid, is it possible to place N Queens in the grid so that no two Queens can attack each other ?

- **Answer:** Yes.

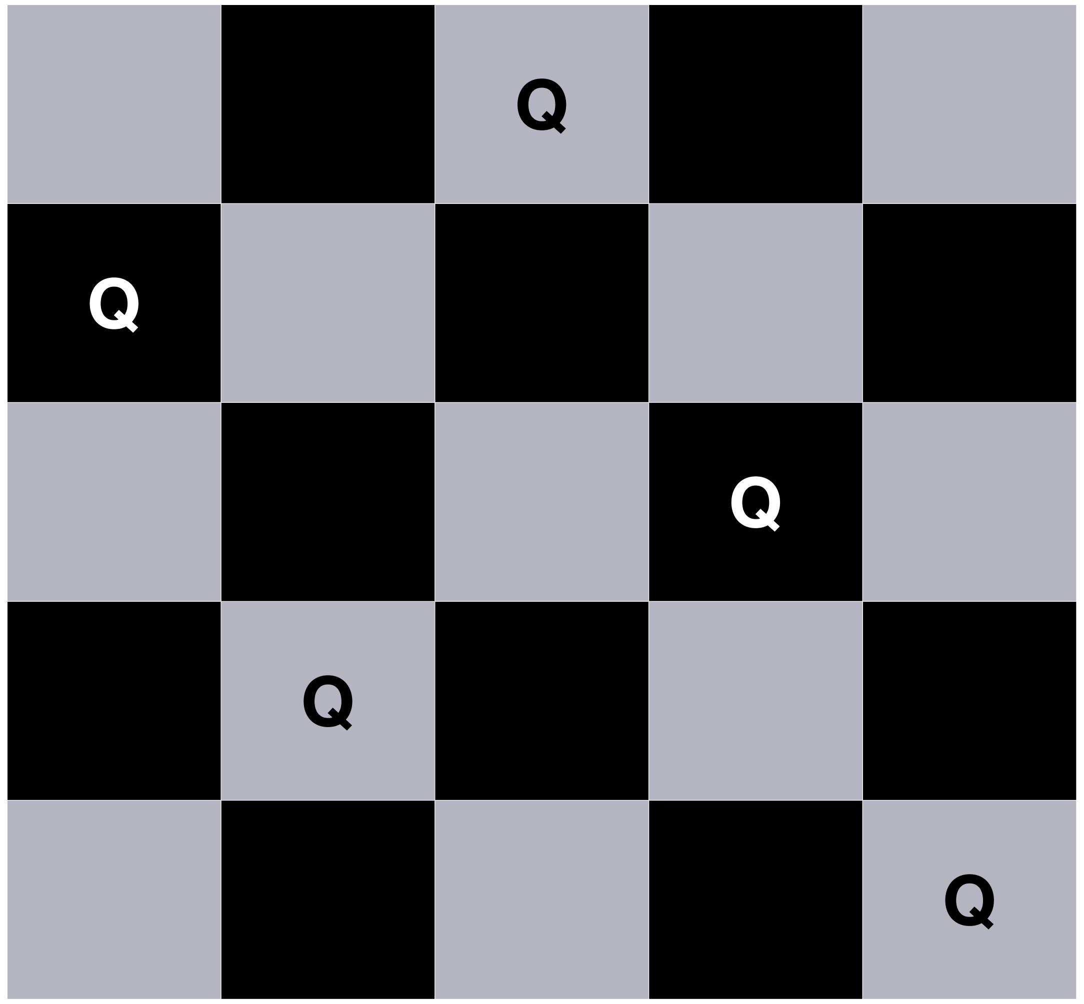
N - Queens Problem

- Here is a possible solution for the 5 x 5 grid.
  - Not unique
- Can we make the computer solve it for any given N?
  - Solution: Recursion with *backtracking.*



N - Queens Problem

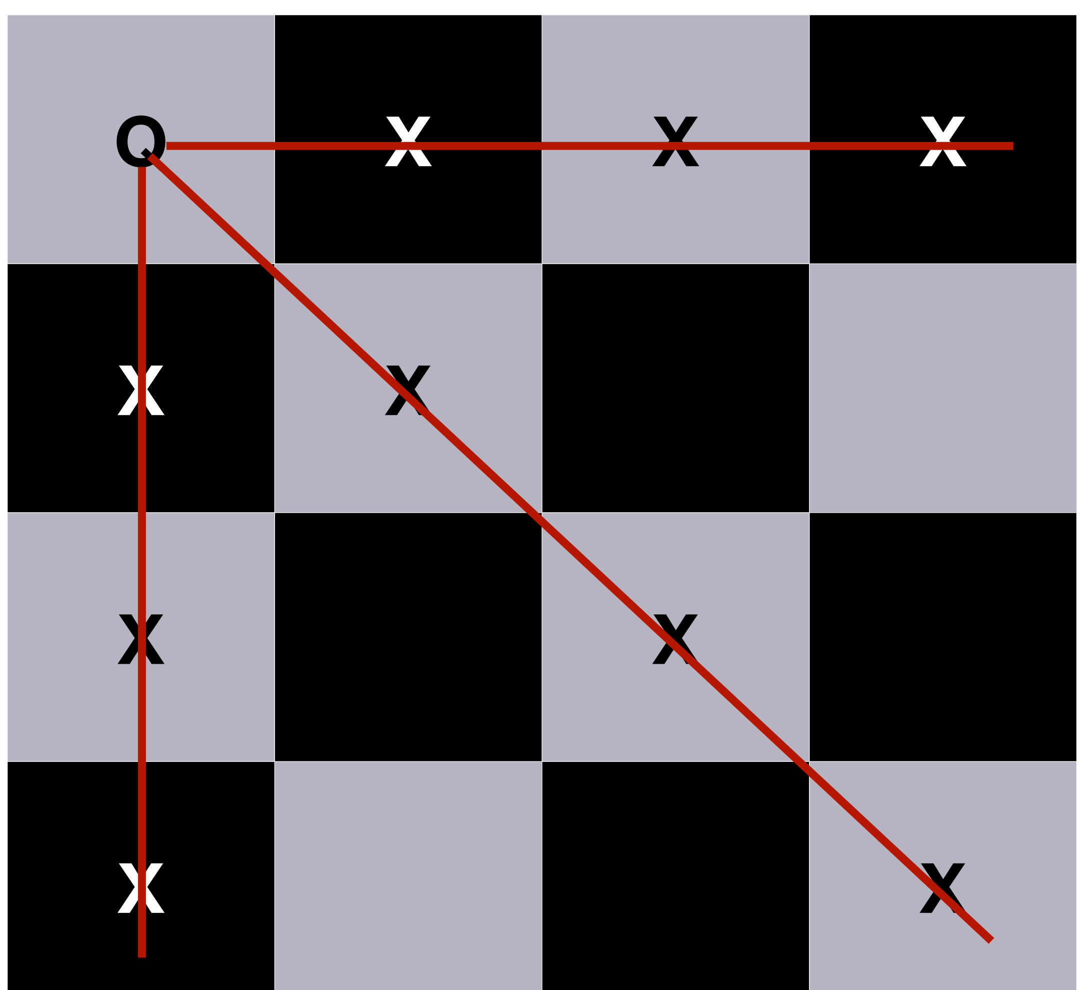
- Here is a possible solution for the 5 x 5 grid.
    - Not unique
- Can we make the computer solve it for any given N?
    - Solution: Recursion with *backtracking.*

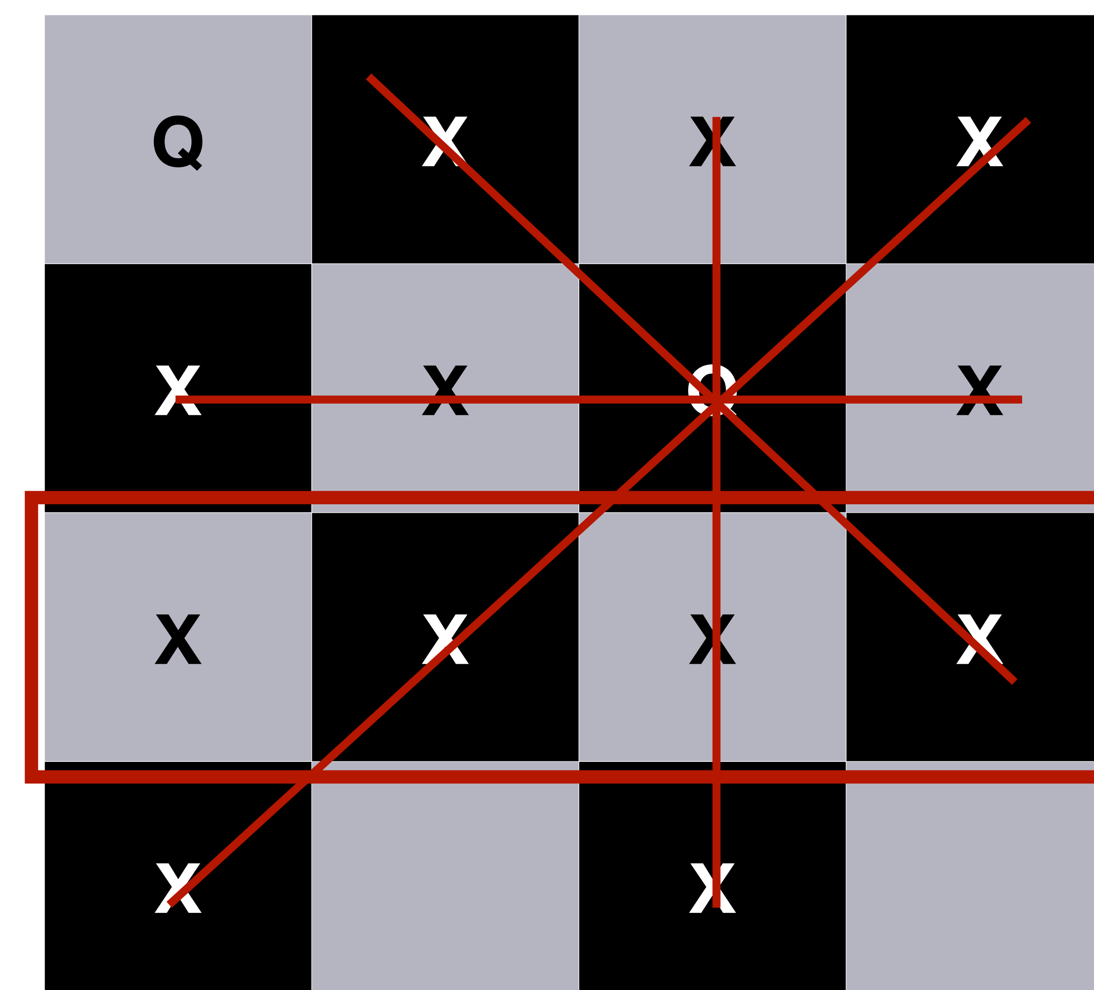


N - Queens Problem

- **Back-tracking**: Make a choice and search the solution space. If solution space is empty, return and make a different choice.

Choice #1





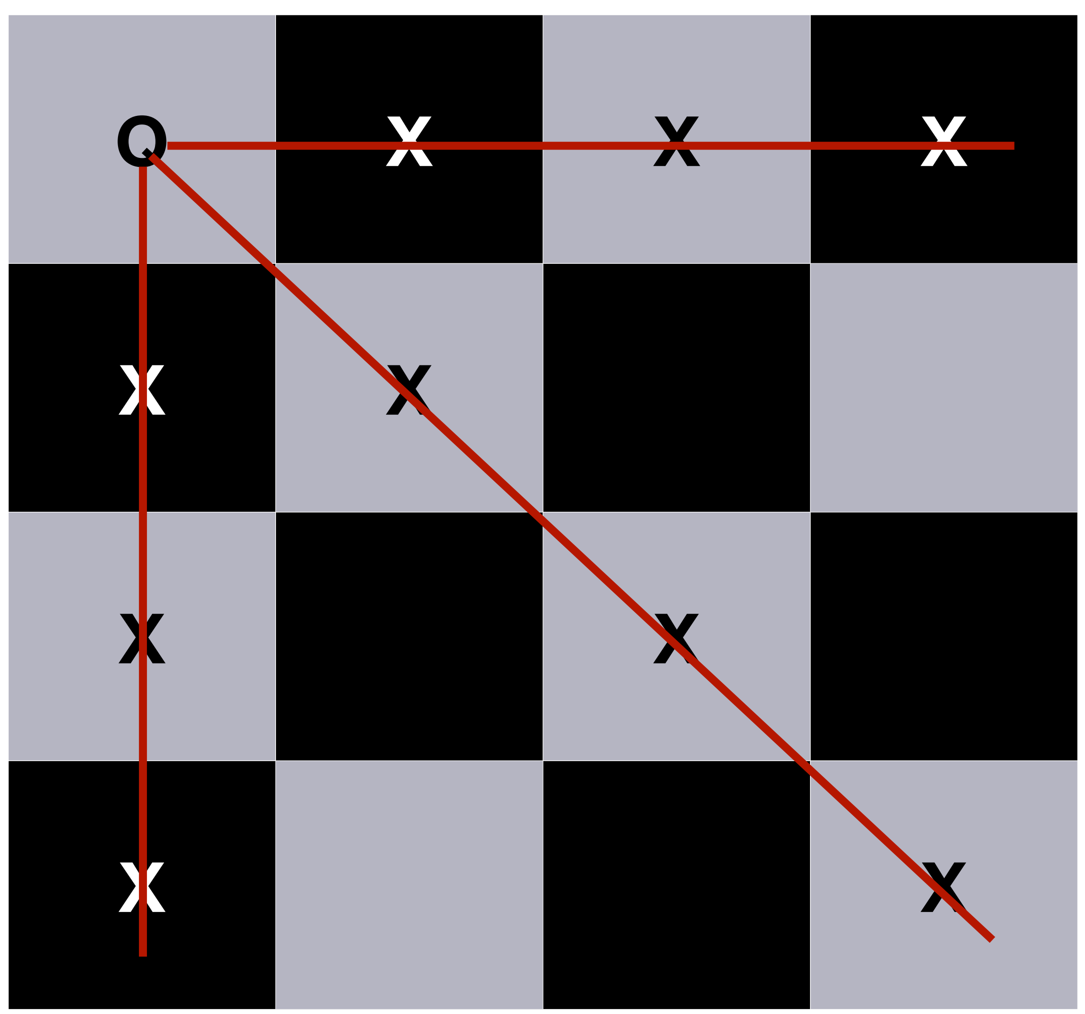
Choice #1.1

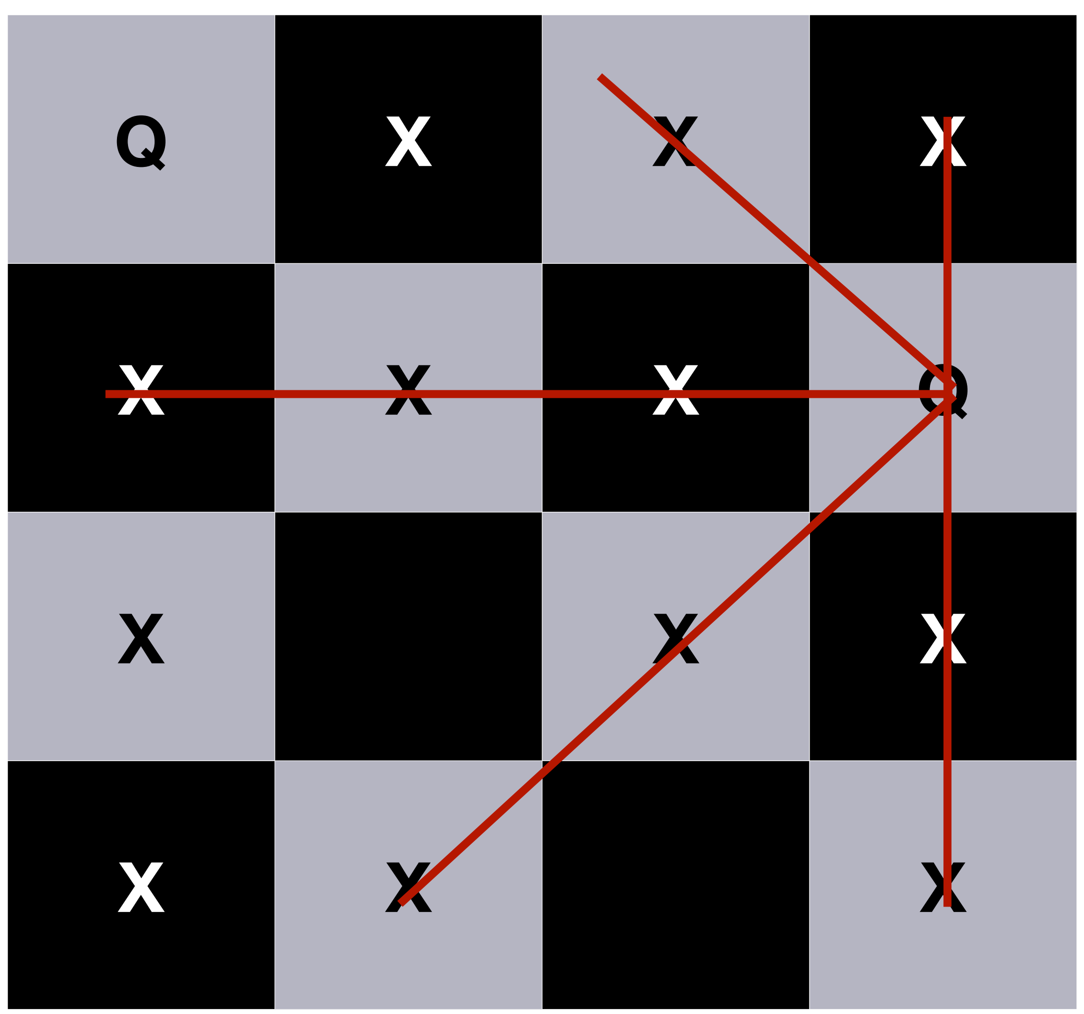
Not a solution!

N - Queens Problem

- **Back-tracking**: Make a choice and search the solution space. If solution space is empty, return and make a different choice.

Choice #1



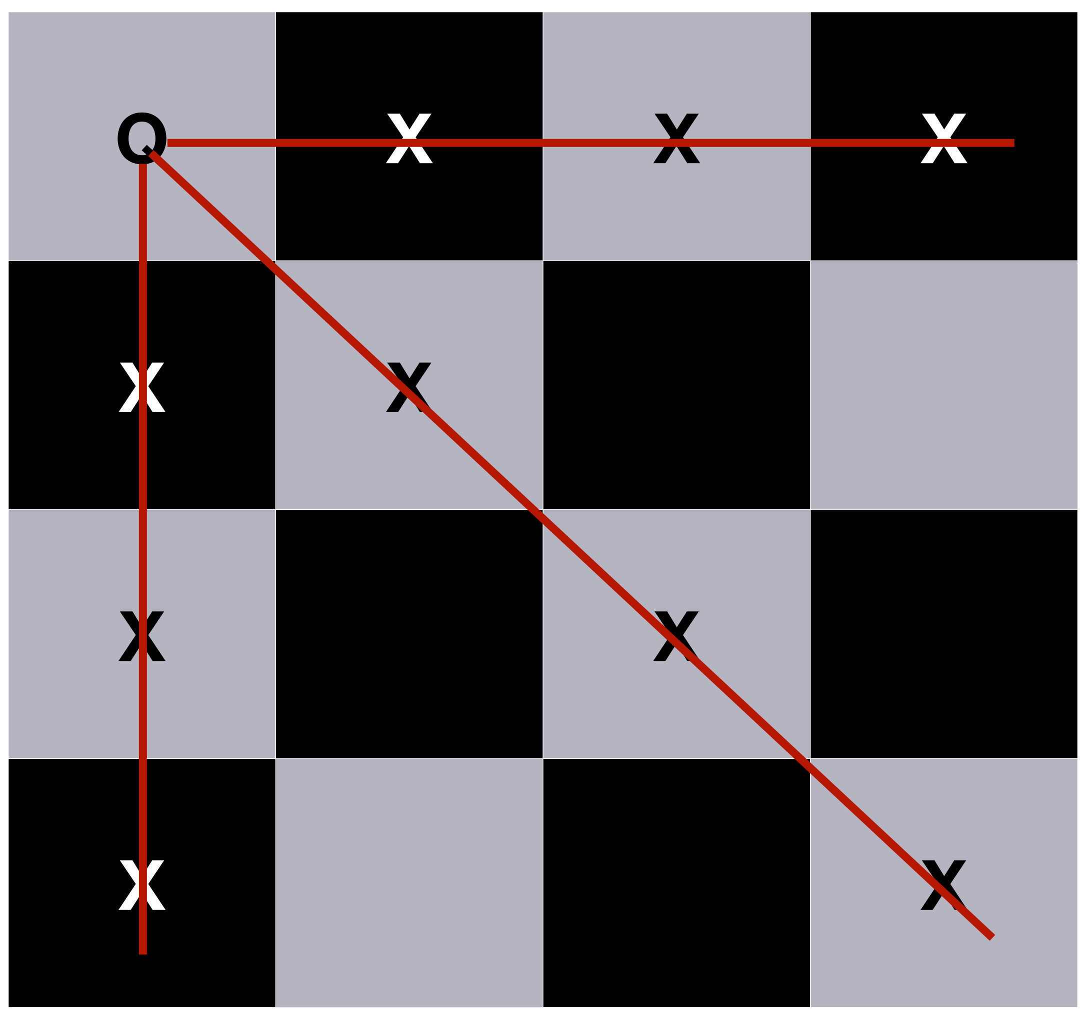


Choice #1.2

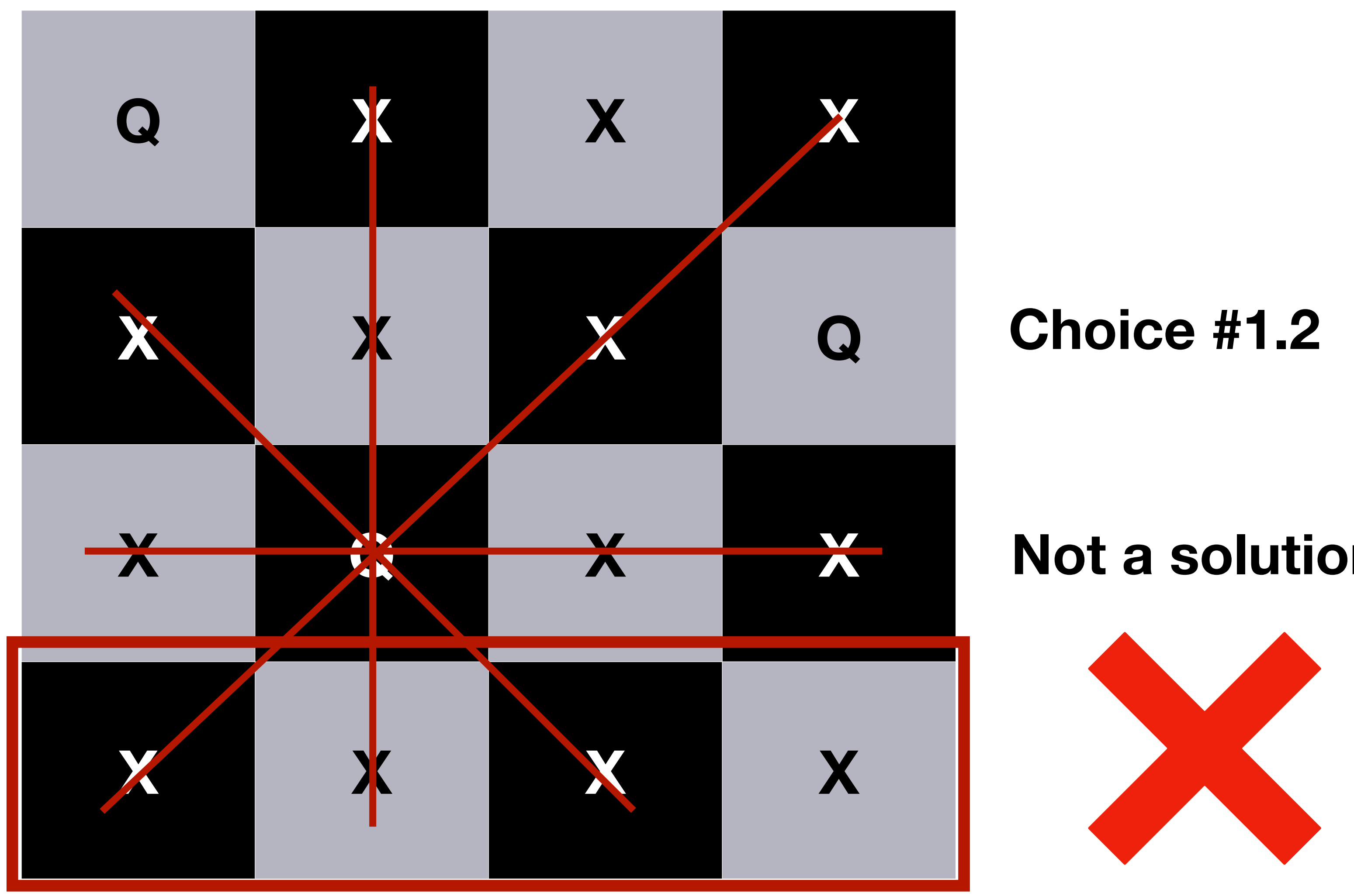
N - Queens Problem

- **Back-tracking**: Make a choice and search the solution space. If solution space is empty, return and make a different choice.

Choice #1



Choice #1.2.1



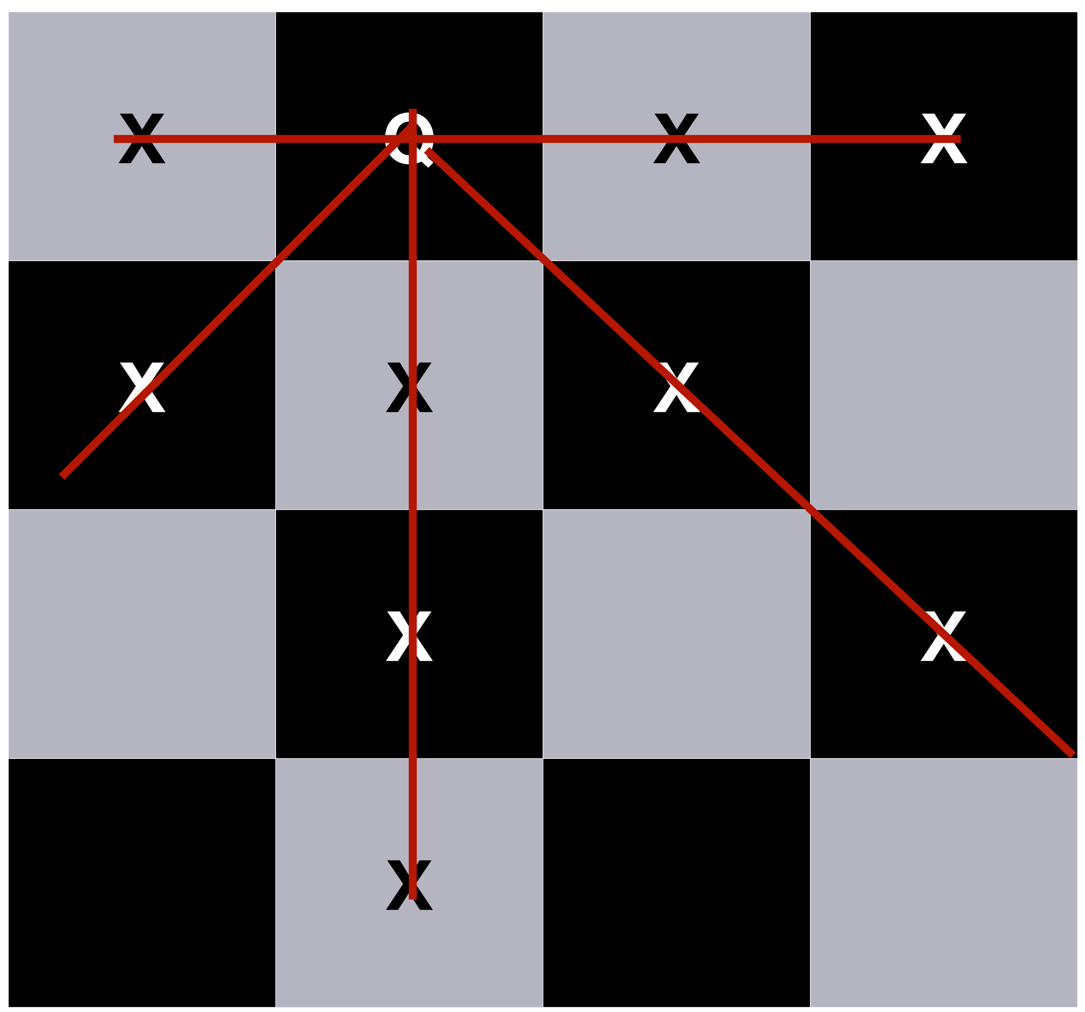
Choice #1.2

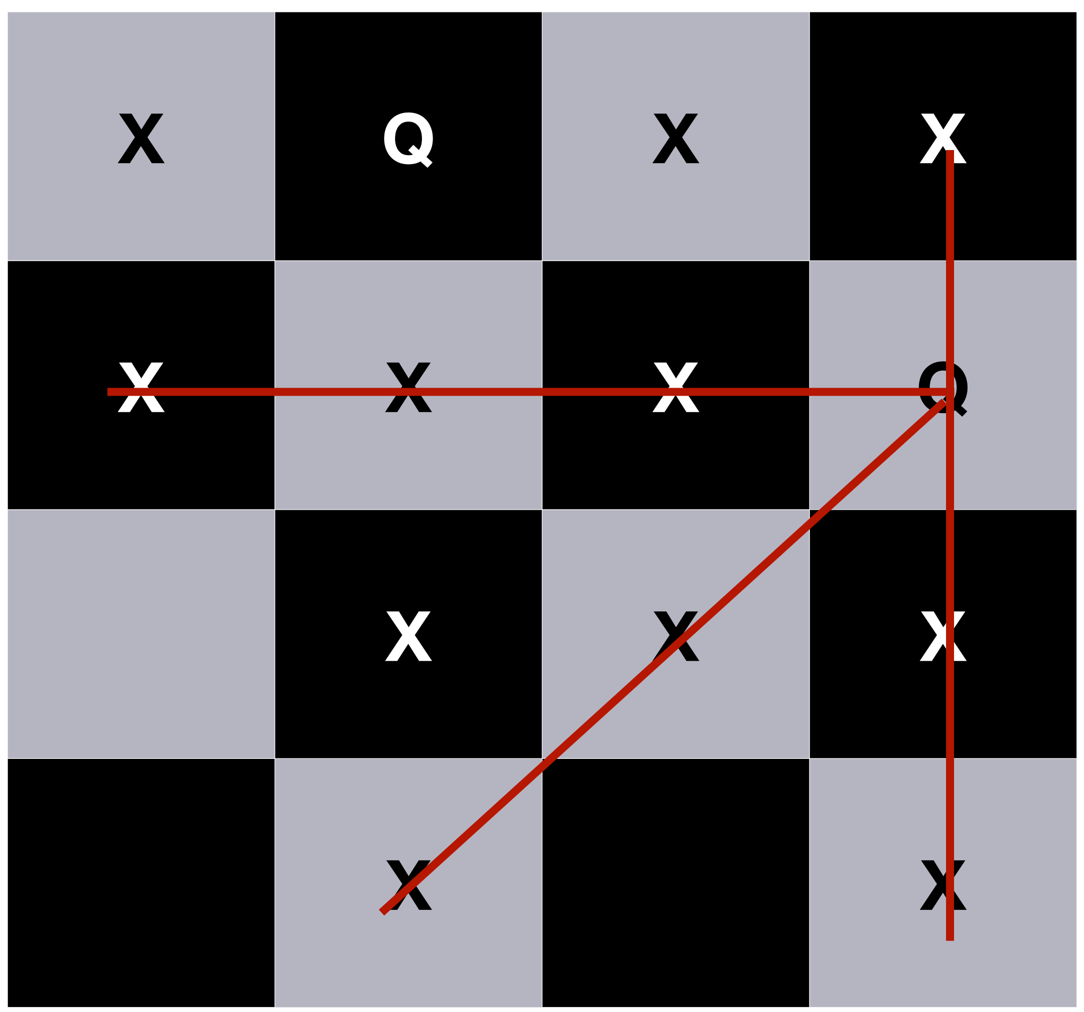
Not a solution!

N - Queens Problem

- **Back-tracking**: Make a choice and search the solution space. If solution space is empty, return and make a different choice.

Choice #2

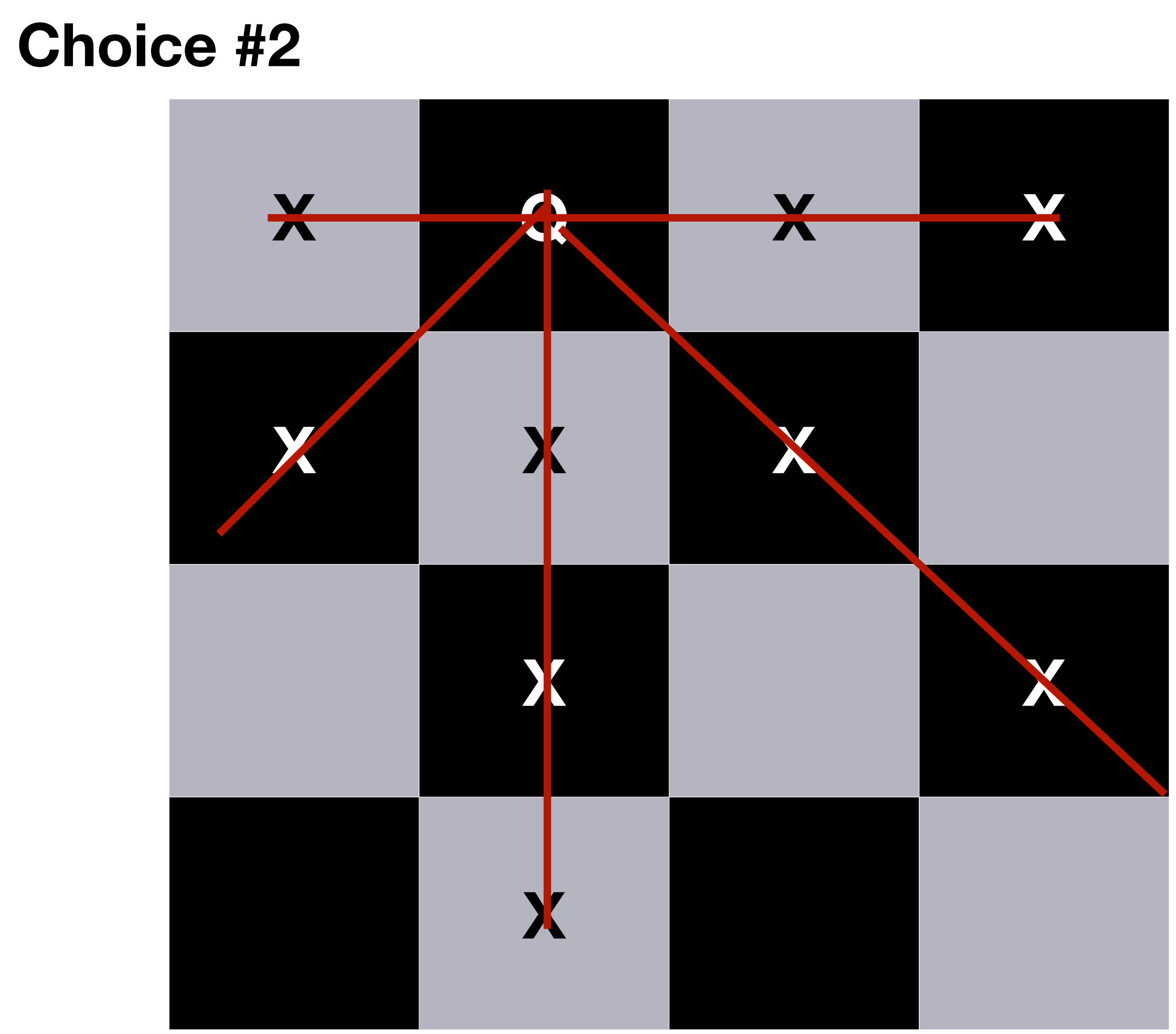


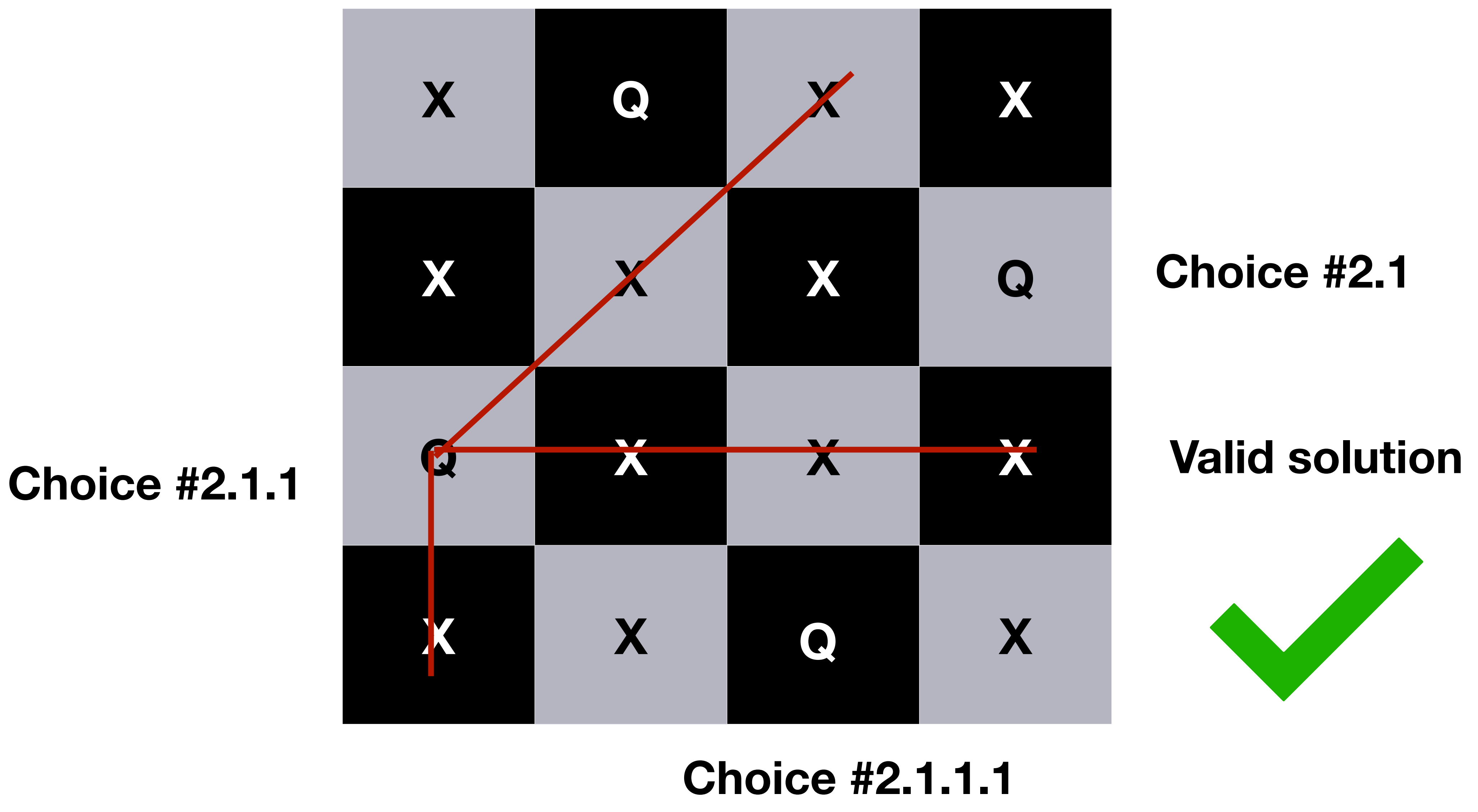


Choice #2.1

N - Queens Problem

- **Back-tracking**: Make a choice and search the solution space. If solution space is empty, return and make a different choice.





N - Queens implementation?

- **Question:** Can we set this up as a recursive problem?
  - What is the action/sub-problem that we want to repeat?

    → Placing a Queen in a row

    - If not successful how do we *backtrack?*

      → Undo placing a queen
  - How do we know we have reached an end case?

    → No more rows to fill.

N - Queens set-up?

- We represent the configuration space with a grid.
    - We will denote with digit **zero** an empty spot (maybe safe or unsafe, but its unoccupied).
    - We will denote with the digit **one** a space occupied by a queen.
    - We will fill in rows starting with the first row and proceeding downward.

N - Queens implementation

```
int is_safe(int board[N][N], int rnum, int cnum);

/*Function places a queen in row rnum */
int place_queen(int board[N][N], int rnum){
  if (          )  // Finished all rows
    return 1;      // Found a solution
  else{
    // Iterate over possible columns
    for(int cnum=0;          ; cnum++)
      if (is_safe(                 )==1){
        board[rnum][cnum] = 1; // Place a queen there
        // Update row number and recurse
        if (                           ==1)
          return 1;
        else // Hit a road block down the line
                                // Remove queen
    } // Try next column along row
  }   // For loop finished without hitting a return
            // Solution doesn't exist.
}
```

is_safe checks whether it is possible to place a queen on position (**rnum, num**) given the configuration of the board at some given time. It returns 1 if safe or 0 if unsafe

place_queen fills the board with a valid solution and returns 1 or returns 0 if no solution found.

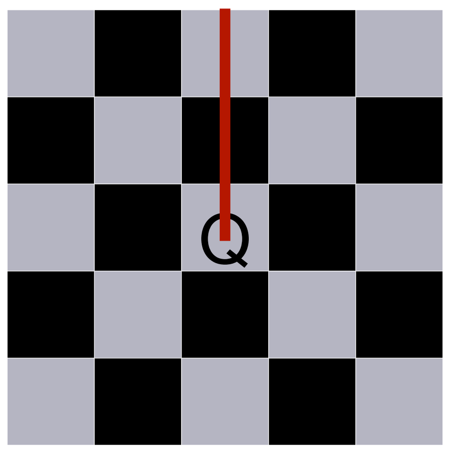
Is it safe/unsafe?

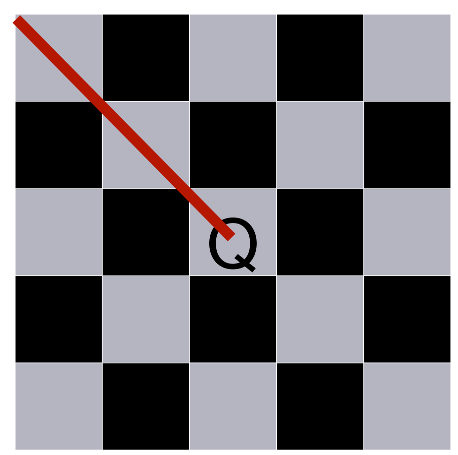
- On the N-th row when we place a queen on a square `(i,j)` what do we need to check?

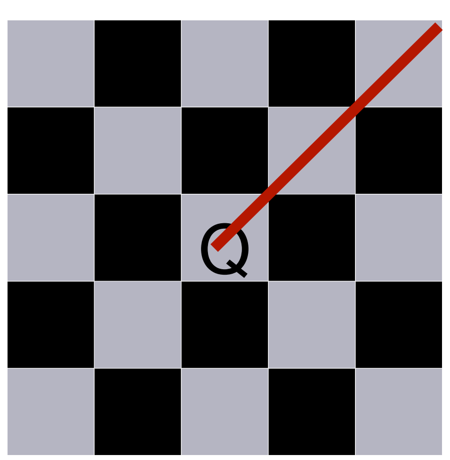
→ Are we in the line of sight (LOS) for any previous Queen.

- We are in LOS if
    - The column `i` contains any Queen OR
    - The diagonals to the top-left of `(i,j)` contains a Queen OR
    - The diagonals to the top-right of `(i,j)` contains a Queen
    - What about diagonals to the bottom left or bottom right?

Is it safe/unsafe?








```
int is_safe(int board[N][N], int row, int col){
  int i, j;
  for (                    ){ //Check along column
    if (board[i][col]==1)
      return 0;
  }
  // Check diagonal to upper left
  for (                 ; i>=0 && j>=0; i--, j--){
    if (board[i][j] == 1)
      return 0;
  }
  // Check diagonal to upper right
  for (i=row-1, j=col+1;            ;          ){
    if (board[i][j]==1)
      return 0;
  }
  return 1;
}
```

Exercise - practice, practice, p....

- You have a pile of wood sticks with 3 different lengths: 3, 7, and 10 feet. You wants to connect them and make an X-feet long stick using **at most** 10 sticks.
- To make a stick 33 feet long you can do:
  - 4 x 3F + 3 x 7F ✔
  - 11 x 3F ✘
- Use recursion with backtracking to find a solution

Exercise

```c
#define N 10 // Number allowed
#define M 3  // Types of lengths

// Implement this function
// solution[N]: stores the solution
// idx: index for the solution matrix
// total: remaining length
int solve(int solution[N], int idx, int total);

const int set[M] = {3,7,10};

int main(){
  int solution[N] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  int total;
  printf("Enter total length: ");
  scanf("%d", &total);
   // Write your code here
}
```

Good recursion vs. bad recursion

- Consider the recursive Fibonacci function from last time.

```
long long fib(long long n){
     long long sum;

     if (n == 0 || n == 1)
          return 1;
     else {
          sum = (fib(n-1) + fib(n-2));
          return sum;
          }
}
```

- Let’s do an activity
- Convert this function to an iterative version.
- Compare run times.

Exercise for fun outside lecture

- There is a file on Gitlab which solves the problem for N=5 queens. Make the function work for N=6 and N=7 queens.
- Exercise for the *curious/mighty/brave*:
  - Modify the source so that it keeps a static variable to keep track of the recursive calls.
  - Varying N, generate a plot (plain old Excel is fine) of N vs number of recursive calls. Try N=4, 5, …, 15. What kind of growth is it?