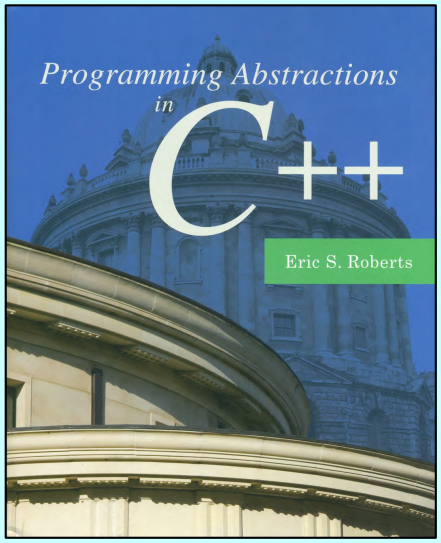
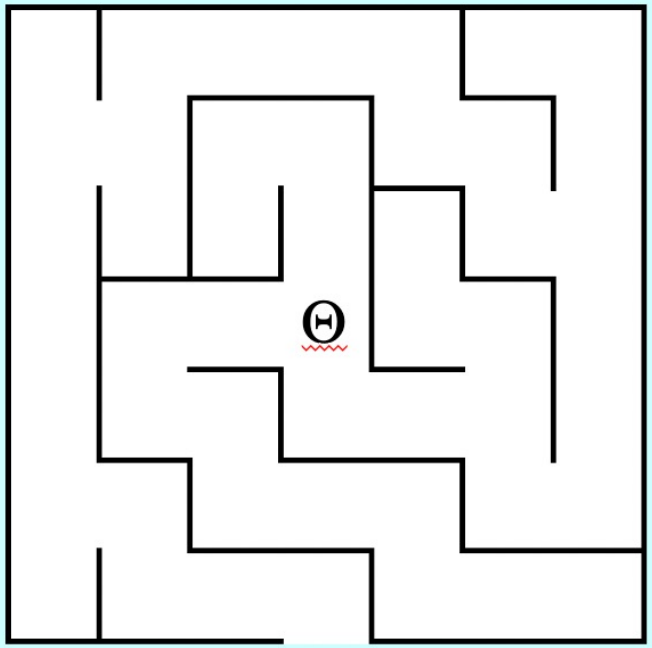
CHAPTER 9

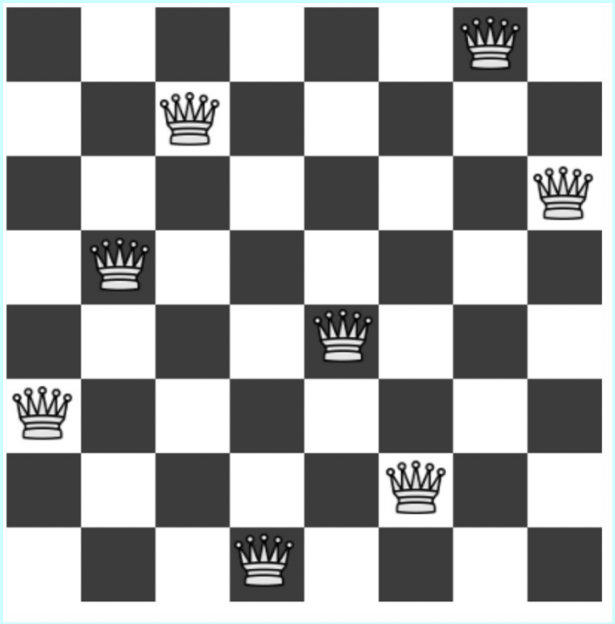
Backtracking Algorithms

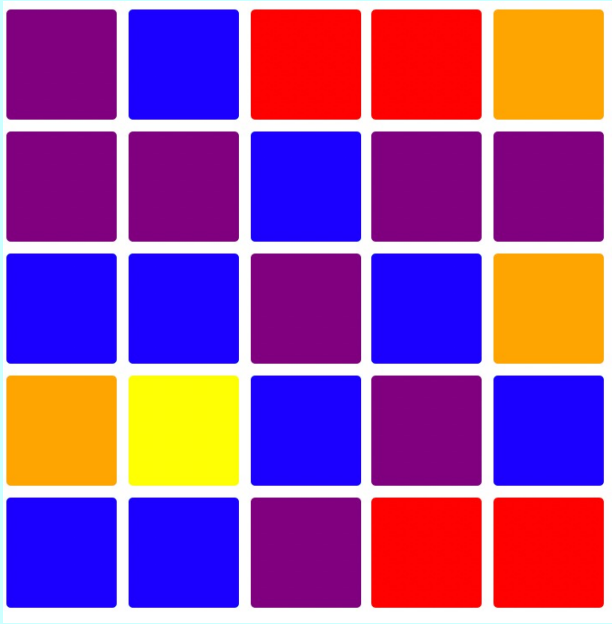
Truth is not discovered by proofs but by exploration. It is always experimental.

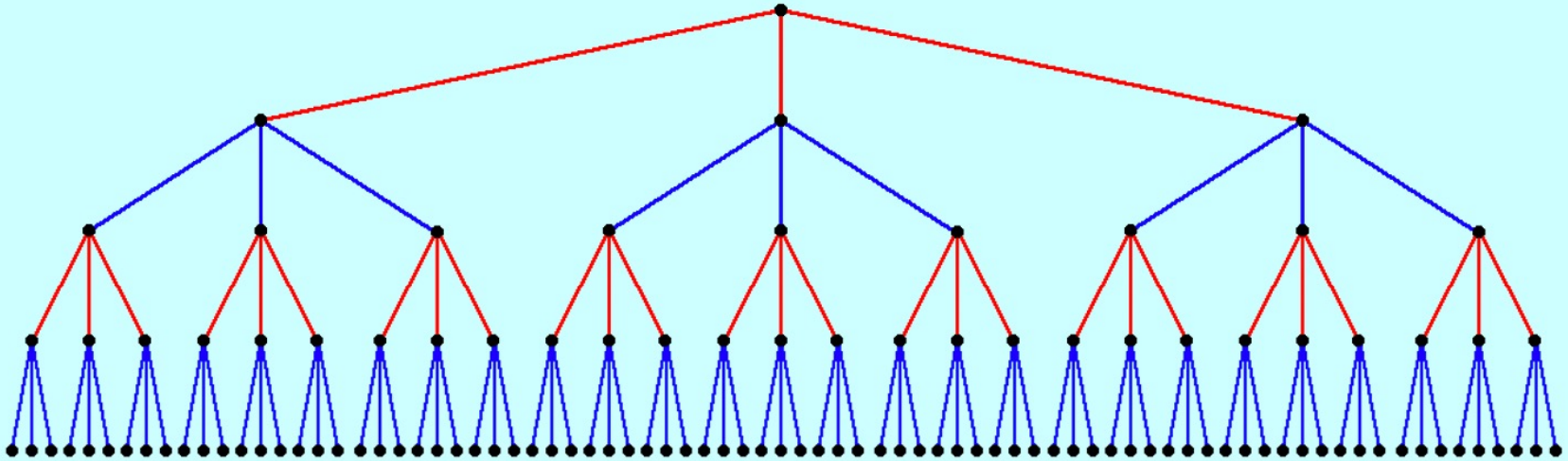
—Simone Weil, *The New York Notebook,* 1942




Θ





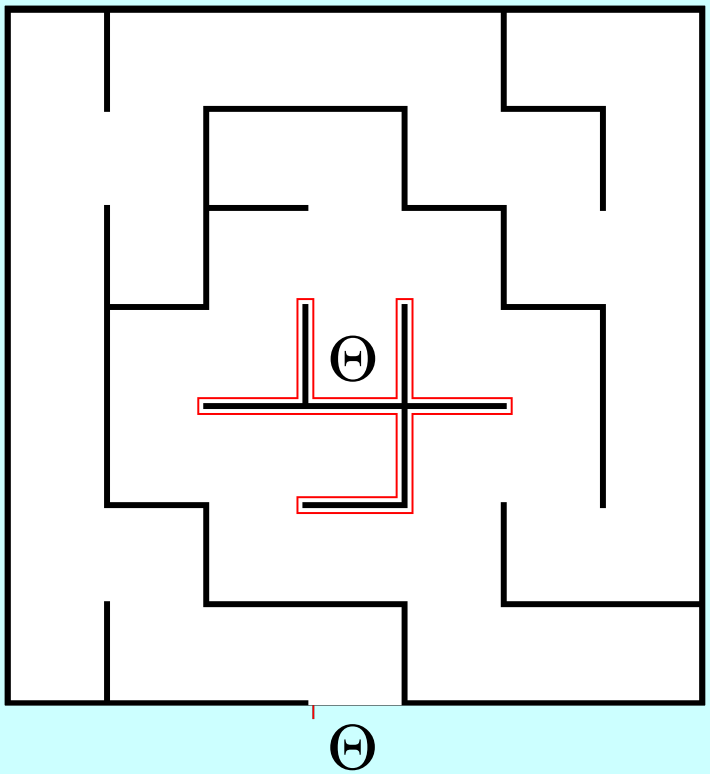


Recursive Backtracking

- For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path. If you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called ***backtracking algorithms***.
- If you think about a backtracking algorithm as the process of repeatedly exploring paths until you encounter the solution, the process appears to have an *iterative* character. As it happens, however, most problems of this form are easier to solve *recursively*. The fundamental recursive insight is simply this: a backtracking problem has a solution if and only if at least one of the smaller backtracking problems that result from making each possible initial choice has a solution.

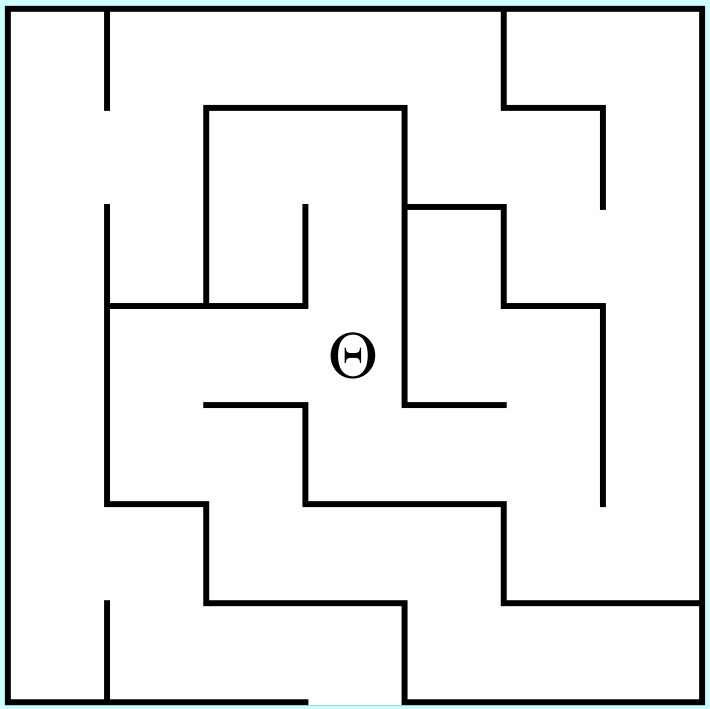
Maze - The Right-Hand Rule

- The most widely known strategy for solving a maze is called the ***right-hand rule***, in which you put your right hand on the wall and keep it there until you find an exit.
- If Theseus applies the right-hand rule in this maze, the solution path looks like this.
- Unfortunately, the right-hand rule doesn't work if there are loops in the maze that surround either the starting position or the goal.
- In this maze, the right-hand rule sends Theseus into an infinite loop.



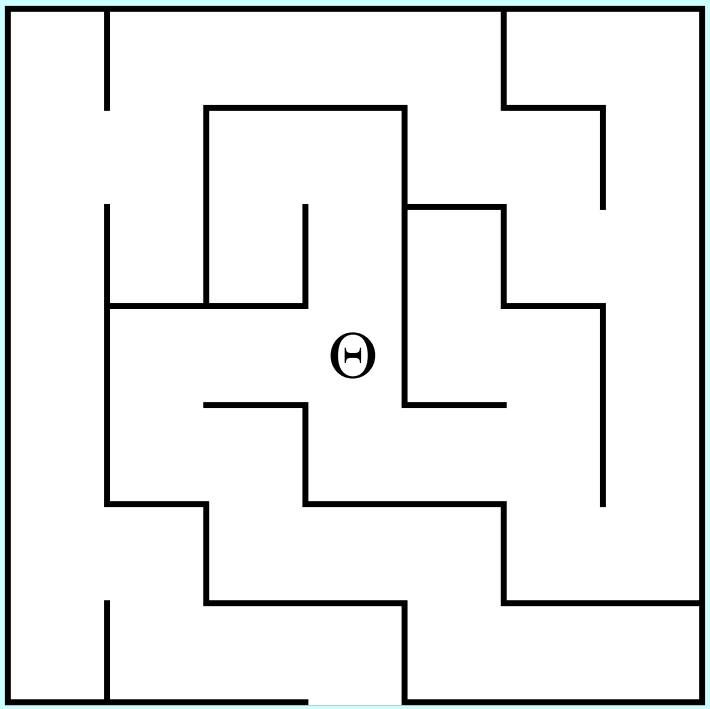
A Recursive View of Mazes

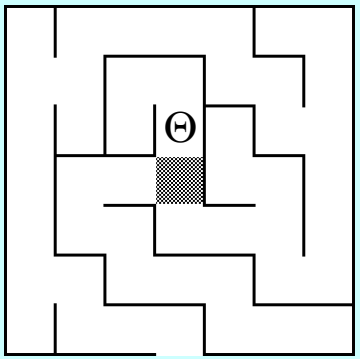
- It is also possible to solve a maze recursively. Before you can do so, however, you have to find the right recursive insight.
- Consider the maze shown at the right. How can Theseus transform the problem into one of solving a simpler maze?
- The insight you need is that a maze is solvable only if it is possible to solve one of the simpler mazes that results from shifting the starting location to an adjacent square and taking the current square out of the maze completely.

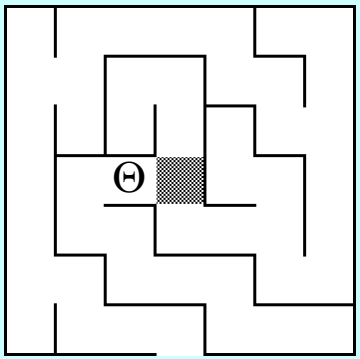


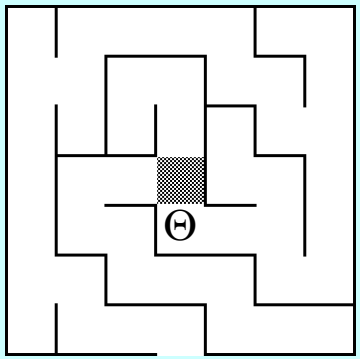
A Recursive View of Mazes

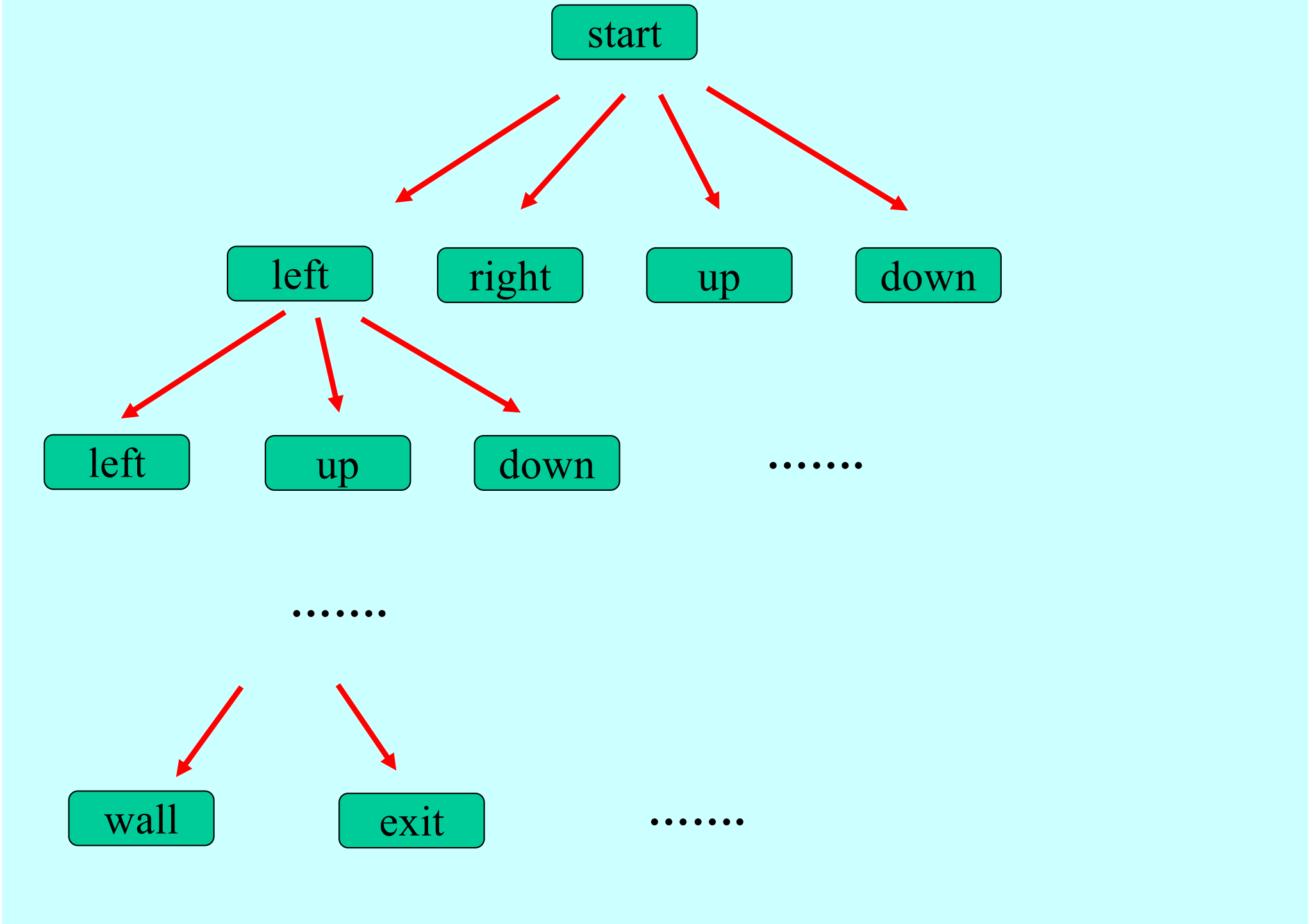
- Thus, the original maze is solvable only if one of the three mazes at the bottom of this slide is solvable.
- Each of these mazes is "simpler" because it contains fewer squares.
- The simple cases are:
  - Theseus is outside the maze
  - There are no directions left to try










start
left
right
up
down
left
up
down
.....
.....
wall
exit
.....

The `solveMaze` Function

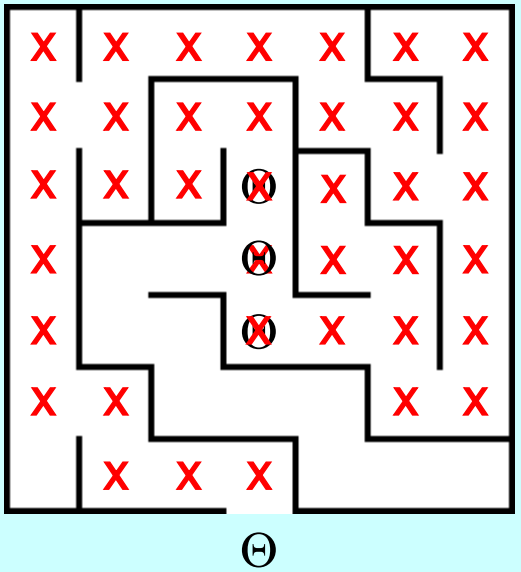
```
/*
 * Function: solveMaze
 * Usage: solveMaze(maze, start);
 * ------------------------------
 * Attempts to generate a solution to the current maze from the specified
 * start point.  The solveMaze function returns true if the maze has a
 * solution and false otherwise.  The implementation uses recursion
 * to solve the submazes that result from marking the current square
 * and moving one step along each open passage.
 */

bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjacentPoint(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

Tracing the `solveMaze` Function

```
bool solveMaze(Maze & maze, Point start) {
```

```
bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjPt(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

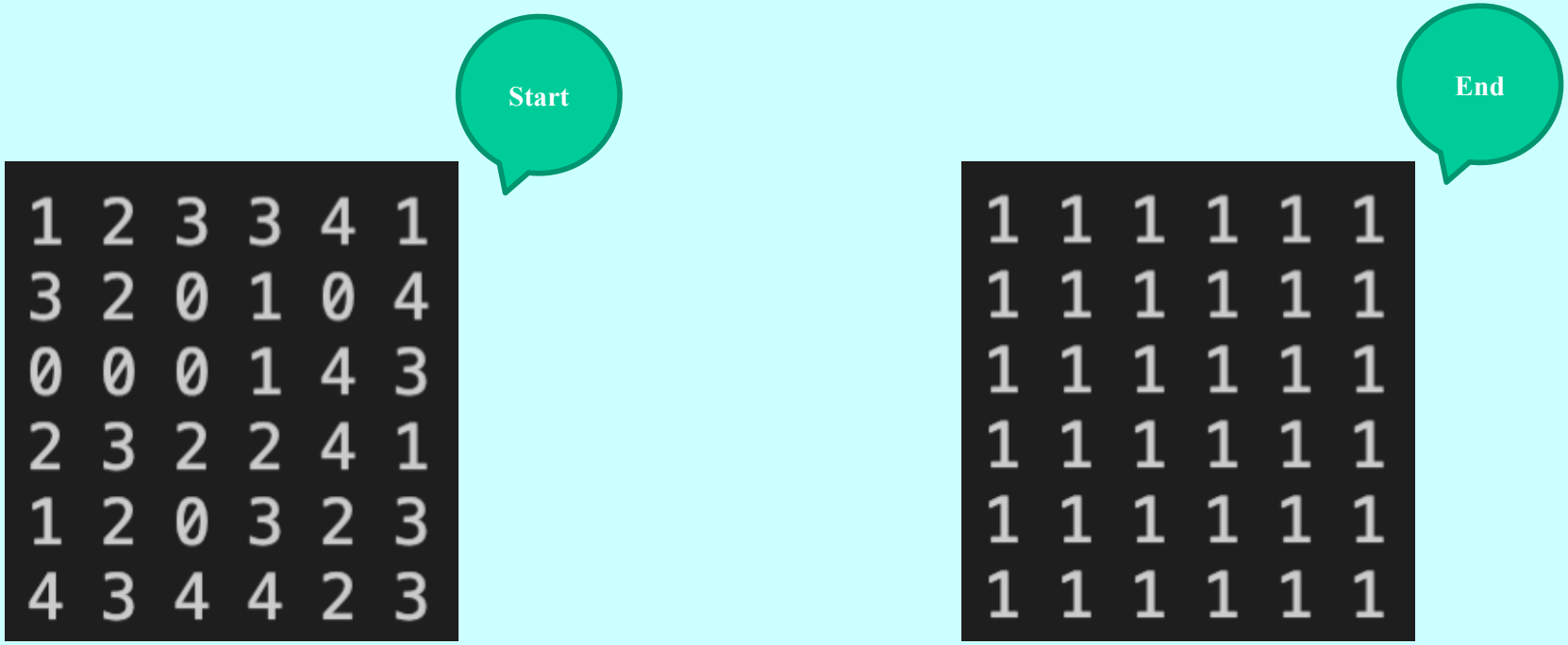
start	dir
(3, 4)	



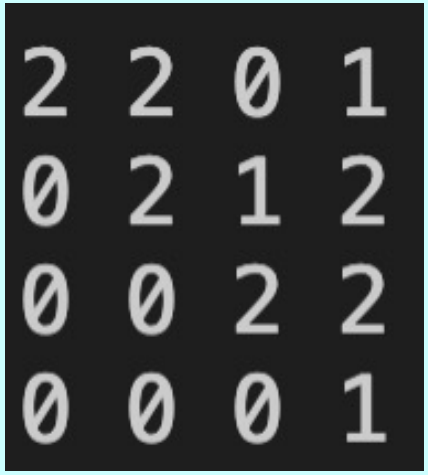
Flipping Number – CSC1002

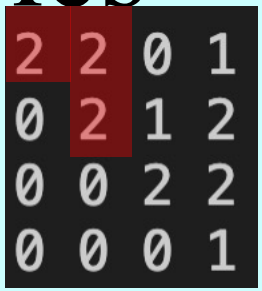
Show Demo

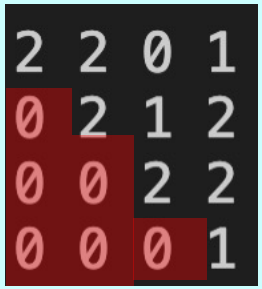
Number-Flipping Game

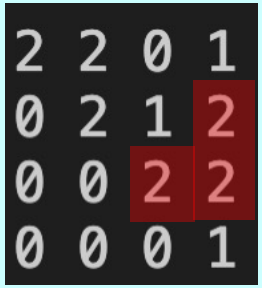


Examples


2 2 0 1
0 2 1 2
0 0 2 2
0 0 0 1


2 2 0 1
0 2 1 2
0 0 2 2
0 0 0 1


2 2 0 1
0 2 1 2
0 0 2 2
0 0 0 1


2 2 0 1
0 2 1 2
0 0 2 2
0 0 0 1

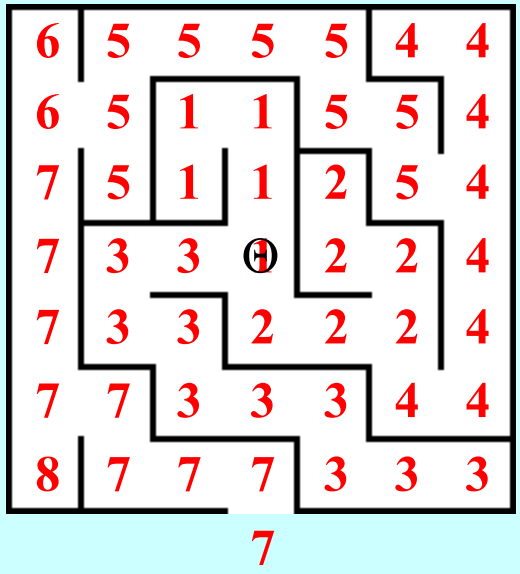
flipNumber() - recursion version

```
48  def flipNumber(row, col, game, orig, to):
49      if orig == to:
50          return game
51      if row < 0 or row >= g_dim:
52          return
53      if col < 0 or col >= g_dim:
54          return
55
56      idx = row*g_dim+col
57      if game[idx] != orig:
58          return
59
60      game[idx] = to
61      flipNumber(row-1, col, game, orig, to)
62      flipNumber(row+1, col, game, orig, to)
63      flipNumber(row, col-1, game, orig, to)
64      flipNumber(row, col+1, game, orig, to)
65
66      return game
```

Back to CSC3002

Recursion and Concurrency

- The recursive decomposition of a maze generates a series of independent submazes; the goal is to solve any one of them.
- If you had a multiprocessor computer, you could try to solve each of these submazes in parallel. This strategy is analogous to cloning yourself at each intersection and sending one clone down each path.



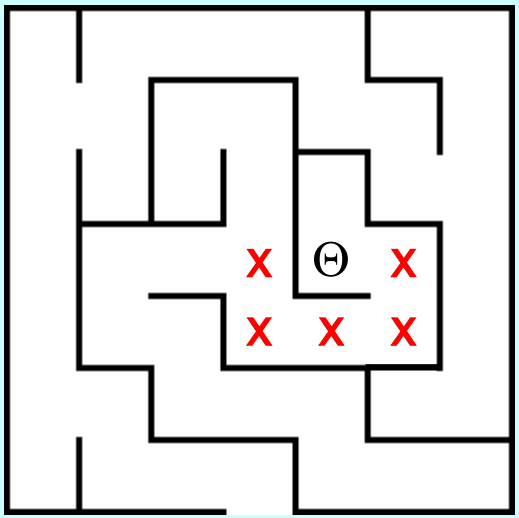
Reflections on the Maze Problem

- The **`solveMaze`** program is a useful example of how to search all paths that stem from a branching series of choices. At each square, the **`solveMaze`** program calls itself recursively to find a solution from one step further along the path.
- To give yourself a better sense of why recursion is important in this problem, think for a minute or two about what it buys you and why it would be difficult to solve this problem iteratively.
- In particular, how would you answer the following questions:
  - What information does the algorithm need to remember as it proceeds with the solution, particularly about the options it has already tried?
  - In the recursive solution, where is this information kept?
  - How might you keep track of this information otherwise?

Each Frame Remembers One Choice

```
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjPt(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

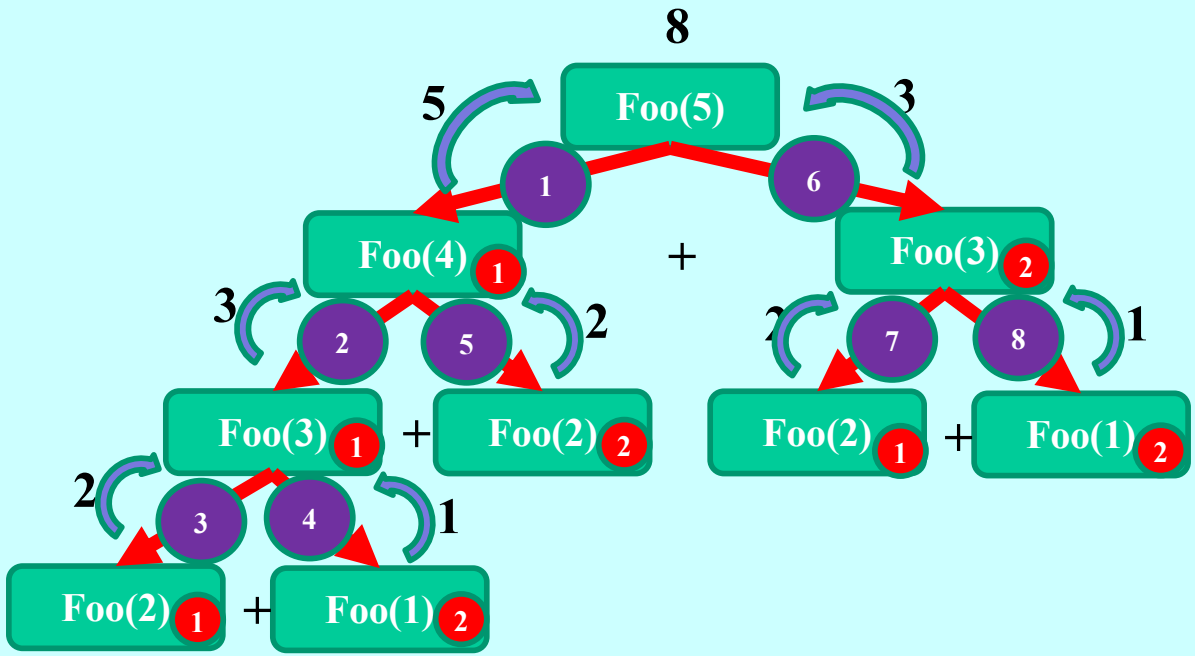
start	dir
(4, 3)	



Recursion – CSC1002

Recursion (CSC1002)

```
def foo(n):
    if n <= 2:
        return n
    return foo(n-1) + foo(n-2)
```



foo(n) Recursive Call + Parameter

x Internal Sequence

\+ Operation applied to data

Exit Condition

x Backtrack with some data x

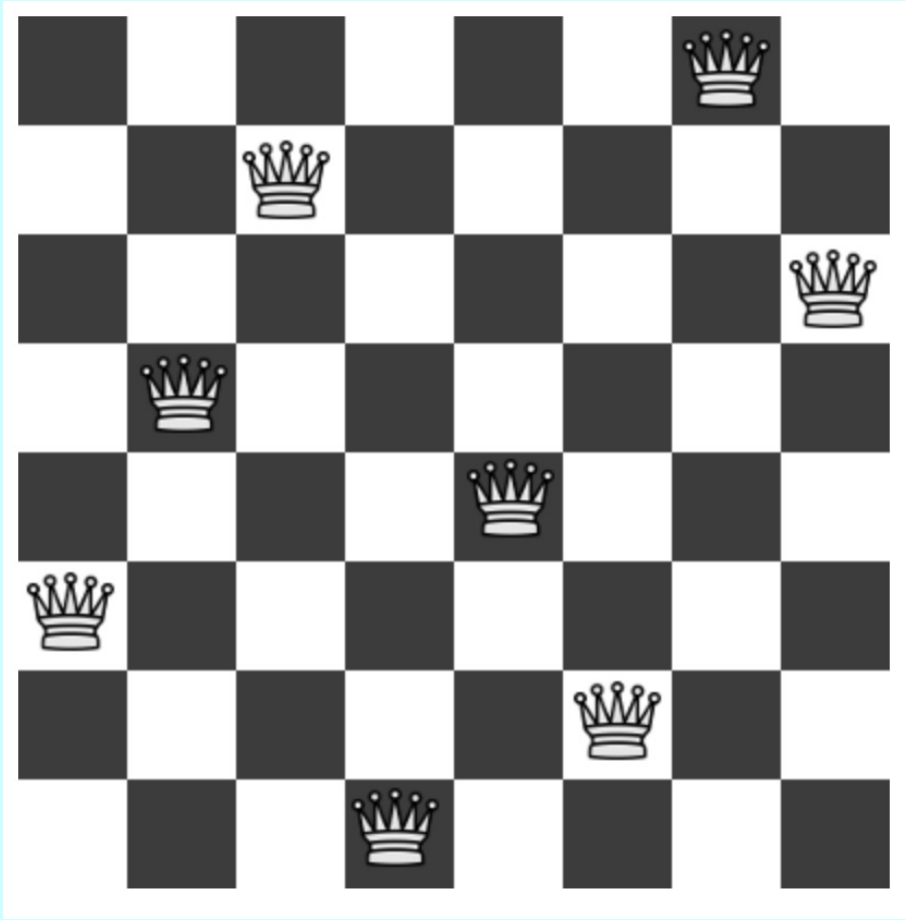
Call Sequence

Conversion of Recursion to Iterative Approach

- Using Stack to simulate the recursive call for backtracking
- Create a stack entry for each recursive call, and push it to the stack
- Each stack entry represents a recursive call, containing the followings:
  - Parameters (variables)
  - A placeholder for return value(s): a single value, a list, or a dictionary
  - Internal Sequence Pointer, initialized to 0
- On each iteration, perform followings in sequence
  - Exit Recursion (Exit Condition) – pop entry and save value to stack
  - Back Track (End of Sequence Call) – pop entry and save values to stack
  - Recursion (Create Call Entry) – create stack entry and push it to stack

Back to CSC3002

Eight Queen

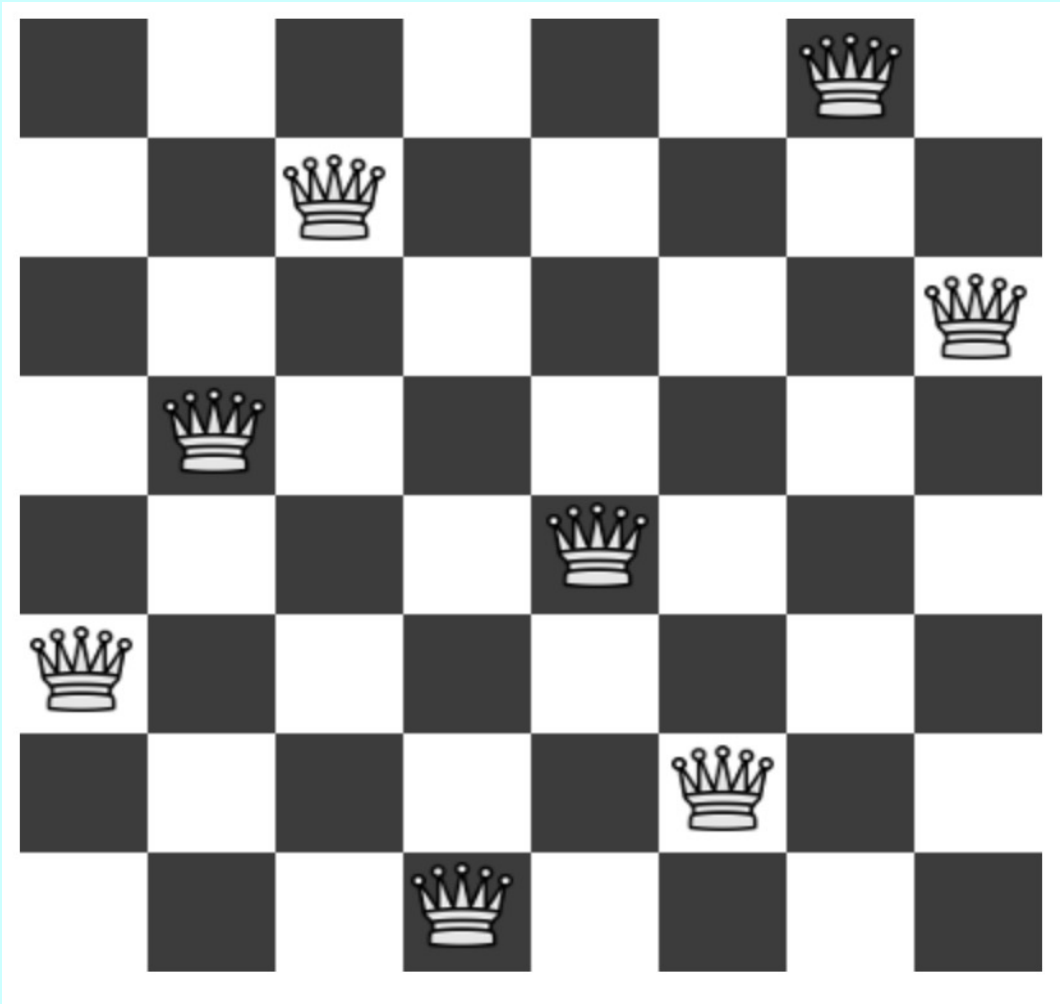


What You Might Attempt to Do

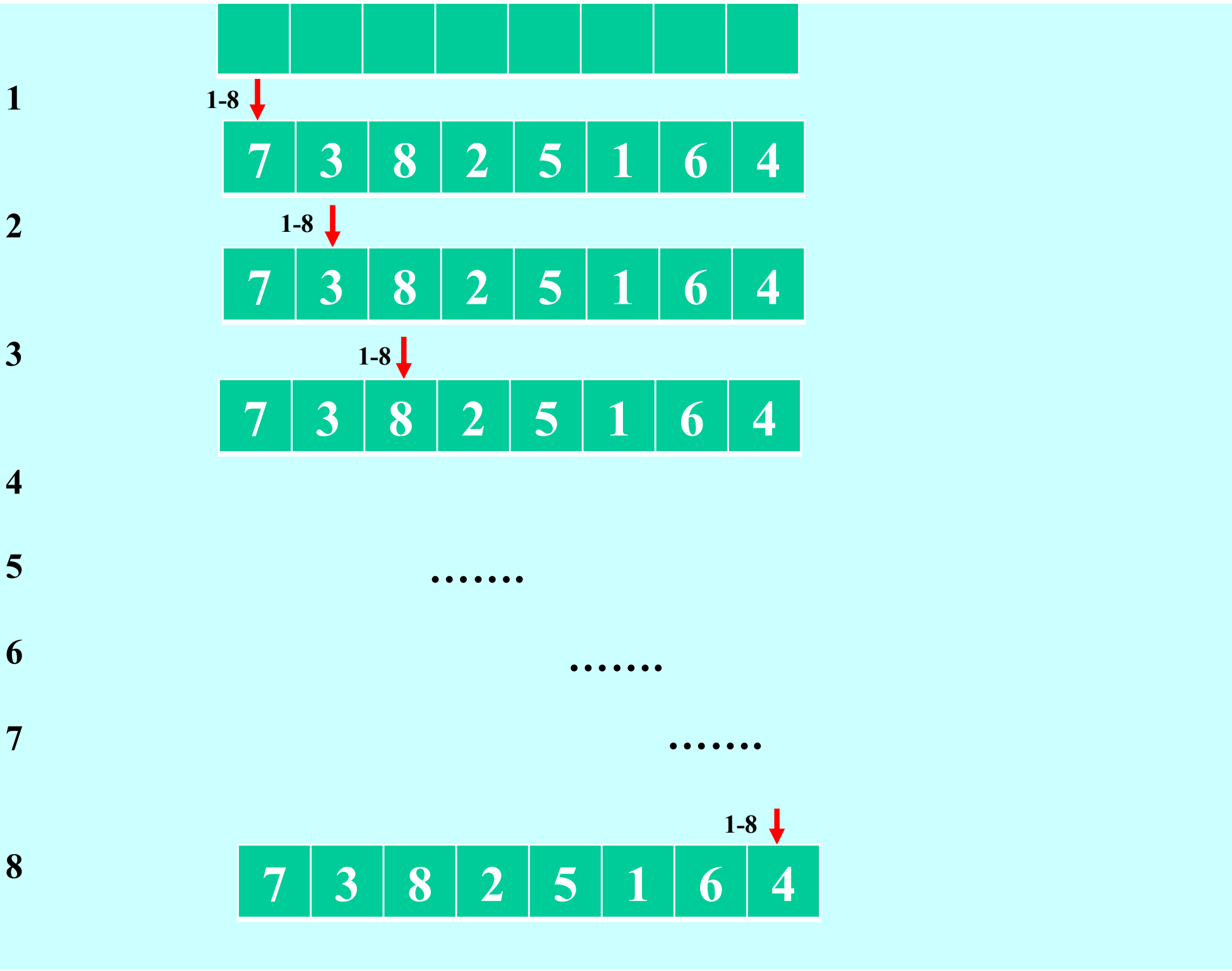
```
bool solveQueens(Grid<char> & board) {
   int n = board.numRows();
   for (int c0 = 0; c0 < n; c0++) {
      board[0][c0] = 'Q';
      for (int c1 = 0; c1 < n; c1++) {
         board[1][c1] = 'Q';
         for (int c2 = 0; c2 < n; c2++) {
            board[2][c2] = 'Q';

            . . .

                  if (boardIsLegal(board)) return true;

            . . .

            board[2][c2] = ' ';
         }
         board[1][c1] = ' ';
      }
      board[0][c0] = ' ';
   }
   return false;
}
```

Decompostion - Scope

7	3	8	2	5	1	6	4



1 2 3 4 5 6 7 8


1
1-8
7 3 8 2 5 1 6 4
2
1-8
7 3 8 2 5 1 6 4
3
1-8
7 3 8 2 5 1 6 4
4
5
.....
6
.....
7
.....
1-8
8
7 3 8 2 5 1 6 4

NQueens

```
/*
 * Function: solveQueens
 * Usage: bool flag = solveQueens(board, nPlaced);
 * -----------------------------------------------
 * Tries to solve the N-Queens problem on the specified board,
 * which is already populated with nPlaced queens by earlier
 * recursive calls.  The function returns true if a solution
 * is found, otherwise it returns false.
 */

bool solveQueens(Grid<char> & board, int nPlaced) {
   int n = board.numRows();
   if (nPlaced == n) return true;
   for (int row = 0; row < n; row++) {
      if (queenIsLegal(board, row, nPlaced)) {
         board[row][nPlaced] = 'Q';
         if (solveQueens(board, nPlaced + 1)) return true;
         board[row][nPlaced] = ' ';
      }
   }
   return false;
}
```

8-Queen – CSC1002

Recursion Approach

```
def put_queen(positions, target_row):
    global solutions
    # Base (stop) case - all N rows are occupied
    if target_row == size:
        show_full_board(positions)
        solutions += 1
    else:
        # For all N columns positions try to place a queen
        for column in range(size):
            # Reject all invalid positions
            if check_place(positions, target_row, column):
                positions[target_row] = column
                put_queen(positions, target_row + 1)
```

Iteration Approach

```
while True:
    # print(stack)
    if len(stack) == 1:
        break

    row, value, col = stack[-1]

    # Base (stop) case - all N rows are occupied
    if row == size:
        show_full_board(positions)
        solutions += 1
        stack.pop()
        stack[-1][-1] = stack[-1][-1] + 1
        continue

    # Backtrack
    if col == size:
        stack.pop()
        stack[-1][-1] = stack[-1][-1] + 1
        continue

    # For all N columns positions try to place a queen
    if check_place(positions, row, col):
        positions[row] = col
        entry = [row+1, [], 0]
        stack.append(entry)
        continue

    # increment to next column
    stack[-1][-1] = stack[-1][-1] + 1
```

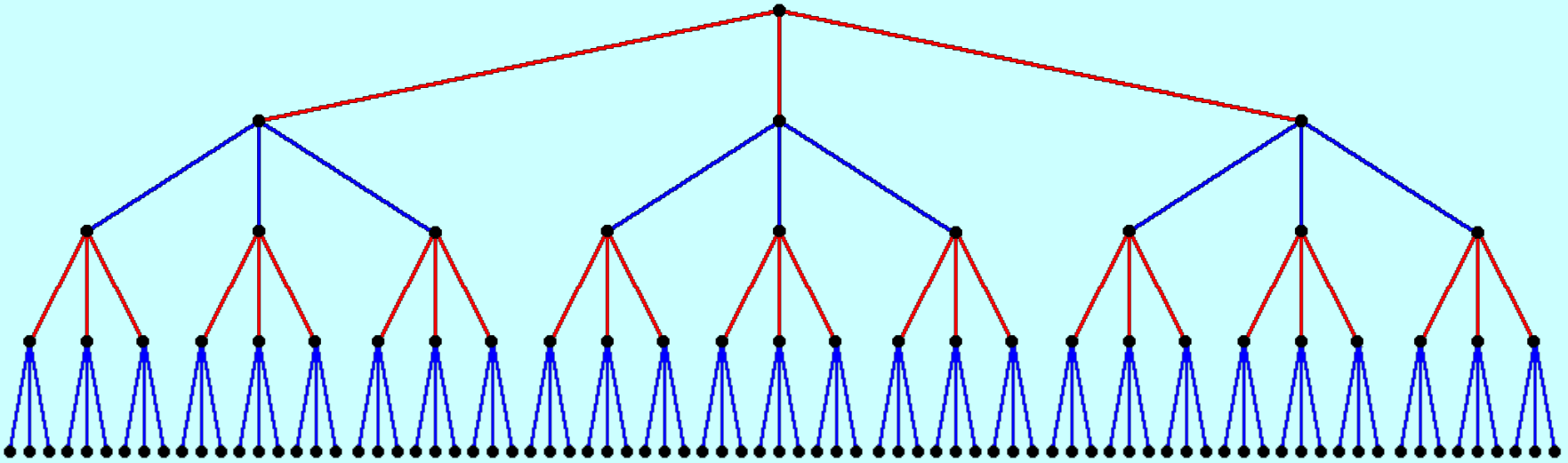
Back to CSC3002

Searching in a Branching Structure

- The recursive structure for finding the solution path in a maze comes up in a wide variety of applications, characterized by the need to explore a range of possibilities at each of a series of choice points.
- The primary advantage of using recursion in these problems is that doing so dramatically simplifies the bookkeeping. Each level of the recursive algorithm considers one choice point. The historical knowledge of what choices have already been tested and which ones remain for further exploration is maintained automatically in the ***execution stack***.
- Many such applications are like the maze-solving algorithm in which the process *searches* a branching structure to find a particular solution. Others, however, use the same basic strategy to explore *every* path in a branching structure in some systematic way.

2-Player Game Trees

- As Shannon observed in 1950, most two-player games have the same basic form:
  - The first player (red) must choose between a set of moves
  - For each move, the second player (blue) has several responses.
  - For each of these responses, red has further choices.
  - For each of these new responses, blue makes another decision.
  - And so on . . .



A Simpler Game

- Chess is far too complex a game to serve as a useful example. The text uses a much simpler game called ***Nim***, which is representative of a large class of two-player games.
- In Nim, the game begins with a pile of coins between two players. The starting number of coins can vary and should therefore be easy to change in the program.
- In alternating turns, each player takes one, two, or three coins from the pile in the center.
- The player who takes the last coin loses.

A Sample Game of Nim

Nim

```
There are 11 coins left.
I'll take 2.
There are 9 coins left.
Your move: 1
There are 8 coins left.
I'll take 3.
There are 5 coins left.
Your move: 2
There are 3 coins left.
I'll take 2.
There is only one coin left.
You lose.
```

Good Moves and Bad Positions

- The essential insight behind the Nim program is bound up in the following mutually recursive definitions:
  - A ***good move*** is one that leaves your opponent in a bad position.
  - A ***bad position*** is one that offers no good moves.
  - E.g., if you ever find yourself with just one coin on the table, you're in a bad position (you have to take that coin and lose). On the other hand, things look good if you find yourself with two, three, or four coins. In any of these cases, you can always take all but one of the remaining coins (good moves), leaving your opponent in the bad position of being stuck with just one coin.
- The implementation of the Nim game is really nothing more than a translation of these definitions into code.

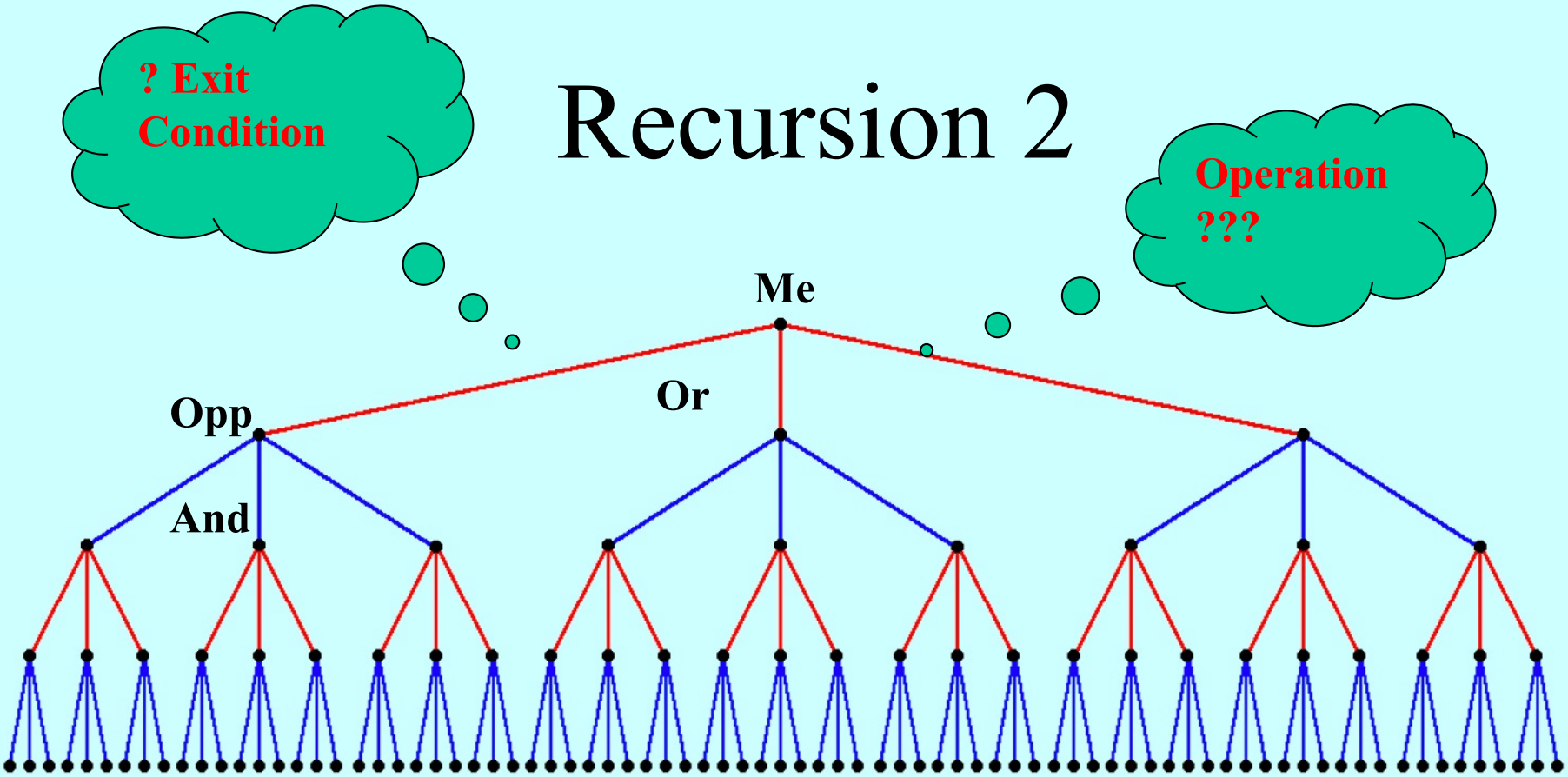
Coding the Nim Strategy

```
/*
 * Looks for a winning move, given the specified number of coins.
 * If there is a winning move in that position, findGoodMove returns
 * that value; if not, the method returns the constant NO_GOOD_MOVE.
 * This implementation depends on the recursive insight that a good move
 * is one that leaves your opponent in a bad position and a bad position
 * is one that offers no good moves.
 */

   int findGoodMove(int nCoins) {
      int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
      for (int nTaken = 1; nTaken <= limit; nTaken++) {
         if (isBadPosition(nCoins - nTaken)) return nTaken;
      }
      return NO_GOOD_MOVE;
   }

/*
 * Returns true if nCoins is a bad position.  Being left with a single
 * coin is clearly a bad position and represents the simple case.
 */

   bool isBadPosition(int nCoins) {
      if (nCoins == 1) return true;
      return findGoodMove(nCoins) == NO_GOOD_MOVE;
   }
```

Coding the Nim Strategy

- `isBadPosition(1)` `true`
- `findGoodMove(2)` `1`
- `isBadPosition(2)` `false`
- `findGoodMove(3)` `2`
- `isBadPosition(3)` `false`
- `findGoodMove(4)` `3`
- `isBadPosition(4)` `false`
- `findGoodMove(5)` `NO_GOOD_MOVE`
- `isBadPosition(5)` `true`
- `...`

Recursion 2



Recursion 2

```
// your turn, me = true
bool isWinnerMove(int cnt, bool me) {

    if (cnt <= 4) return me;

    if (me)
        return isWinnerMove(cnt-1, !me) ||
               isWinnerMove(cnt-2, !me) ||
               isWinnerMove(cnt-3, !me);
    else
        return isWinnerMove(cnt-1, !me) &&
               isWinnerMove(cnt-2, !me) &&
               isWinnerMove(cnt-3, !me);
}
```

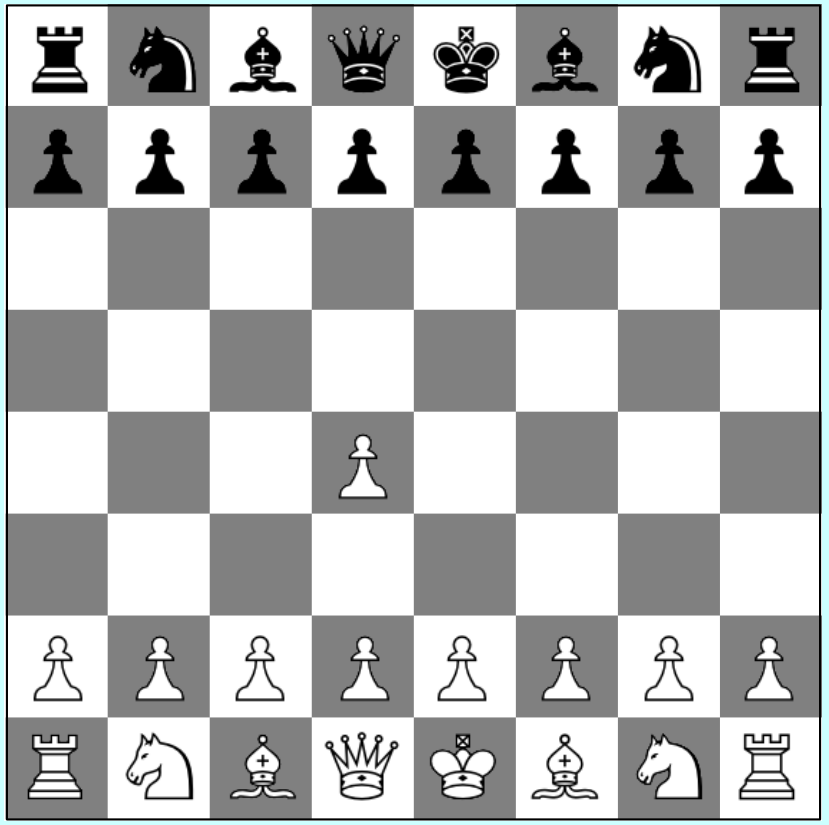
```cpp
for (int i=5; i < 20; i++) {

    auto take1 = isWinnerMove(i-1, false);
    auto take2 = isWinnerMove(i-2, false);
    auto take3 = isWinnerMove(i-3, false);

    cout << i << "= " << take1 << "," << take2 << "," << take3 << endl;

}
```

```
5= 0,0,0
6= 1,0,0
7= 0,1,0
8= 0,0,1
9= 0,0,0
10= 1,0,0
11= 0,1,0
12= 0,0,1
13= 0,0,0
14= 1,0,0
15= 0,1,0
16= 0,0,1
17= 0,0,0
18= 1,0,0
19= 0,1,0
```

```
5= -1
6= 1
7= 2
8= 3
9= -1
10= 1
11= 2
12= 3
13= -1
14= 1
15= 2
16= 3
17= -1
18= 1
19= 2
```

Recursion and Games

- In 1950, Claude Shannon wrote an article for *Scientific American* in which he described how to write a chess-playing computer program.
- Shannon's strategy was to have the computer try every possible move for white, followed by all of black's responses, and then all of white's responses to those moves, and so on.
- Even with modern computers, it is impossible to use this strategy for an entire game, because there are too many possibilities.



Positions evaluated: $\sim 10^{53}$

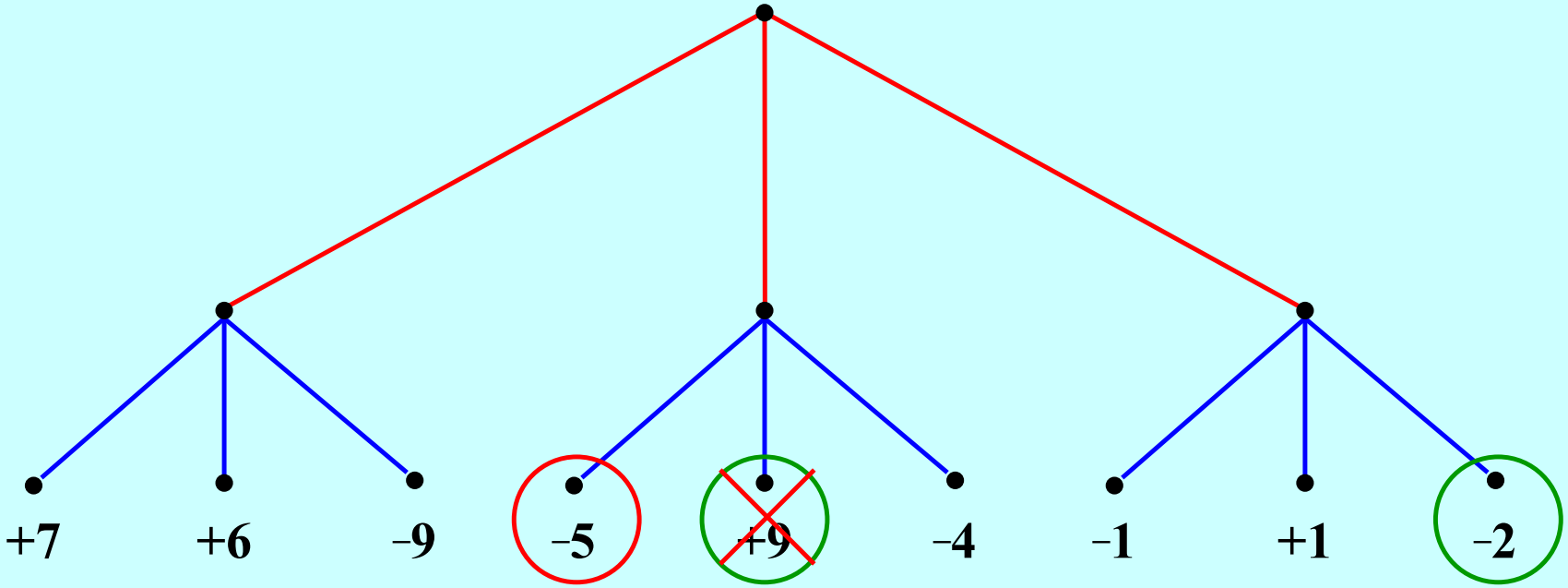
. . . millions of years later . . .

The Minimax Algorithm

- Games like Nim are simple enough that it is possible to solve them completely in a relatively small amount of time.
- For more complex games, it is necessary to cut off the analysis at some point and then evaluate the position, presumably using some function that looks at a position and returns a ***rating*** for that position. Positive ratings are good for the player to move; negative ones are bad.
- When your game player searches the tree for best move, it can't simply choose the one with the highest rating because you control only half the play.
- What you want instead is to choose the move that minimizes the maximum rating available to your opponent. This strategy is called the ***minimax*** algorithm.

A Minimax Illustration

- Suppose that the ratings two turns from now are as shown.
- From your perspective, the +9 initially looks attractive.
- Unfortunately, your can't get there, since the –5 is better for your opponent.
- The best you can do is choose the move that leads to the –2.



The End