SC1007
Data Structures and Algorithms

DFS and Backtracking

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

Lecturer

yrloke@ntu.edu.sg

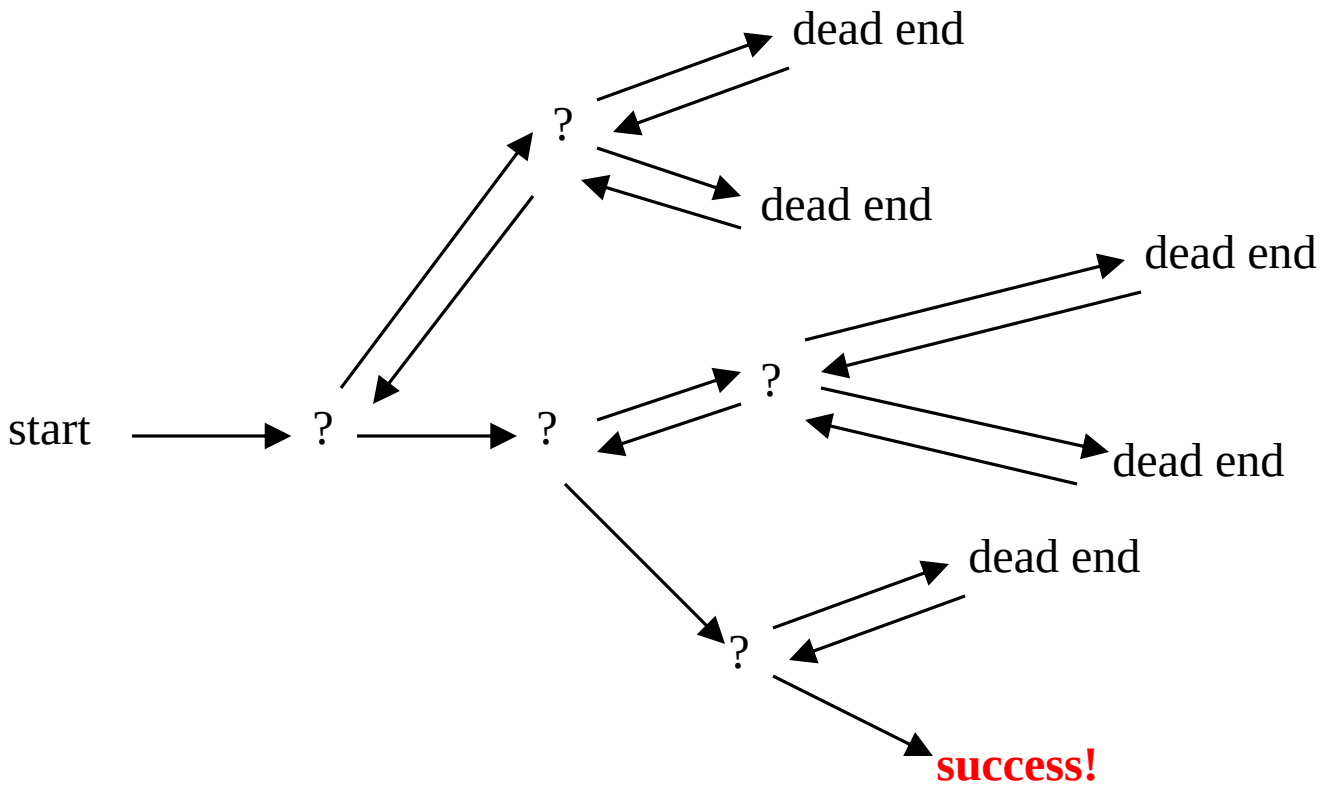
DFS Algorithm

```
function DFS(Graph G, Vertex v)
    create a Stack, S
    push v into S
    mark v as visited
    while S is not empty do
        peek the stack and denote the vertex as w
        if no unvisited vertices are adjacent to w then
            pop a vertex from S
        else
            push an unvisited vertex u adjacent to w
            mark u as visited
        end if
    end while
end function
```

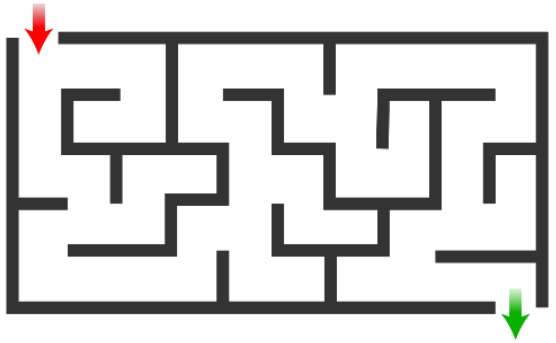
Backtracking

- Suppose you make a series of *decisions,* among various *choices,* where:
  - You don’t have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”

Backtracking (animation)

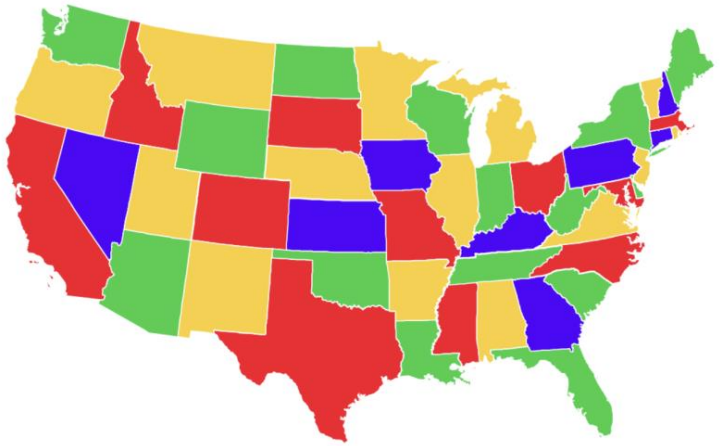


Solving a maze



- Given a maze, find a path from start to finish
- At each intersection, you have to decide:
  - Go straight
  - Go left
  - Go right
- You don’t have enough information to choose correctly
  - Each choice leads to another set of choices
  - One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

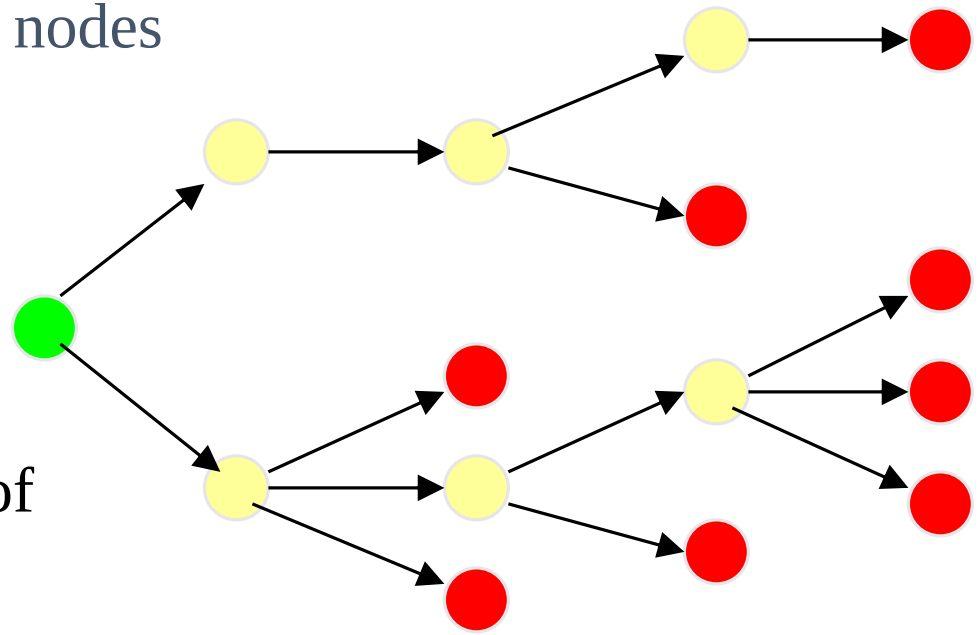
Coloring a map



- You wish to color a map with not more than n colors
- Adjacent regions must be in different colors
- You don’t have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

Terminology

A tree is composed of nodes



There are three kinds of nodes:

- The (one) root node
- Internal nodes
- Leaf nodes

Backtracking can be thought of as searching a tree for a particular “goal” leaf node

The backtracking algorithm

- Backtracking is really quite simple--we “explore” each node, as follows:
- To “explore” node N:

  If N is a goal node, return “success”
  Else if N is a leaf node, return “failure”
  For each child C of N:
      Explore C
      If C was successful, return “success”
  Return “failure”

Backtracking Algorithm

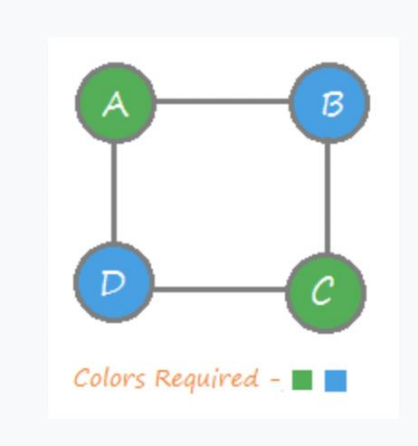
- How to backtrack?
  - Recursive function

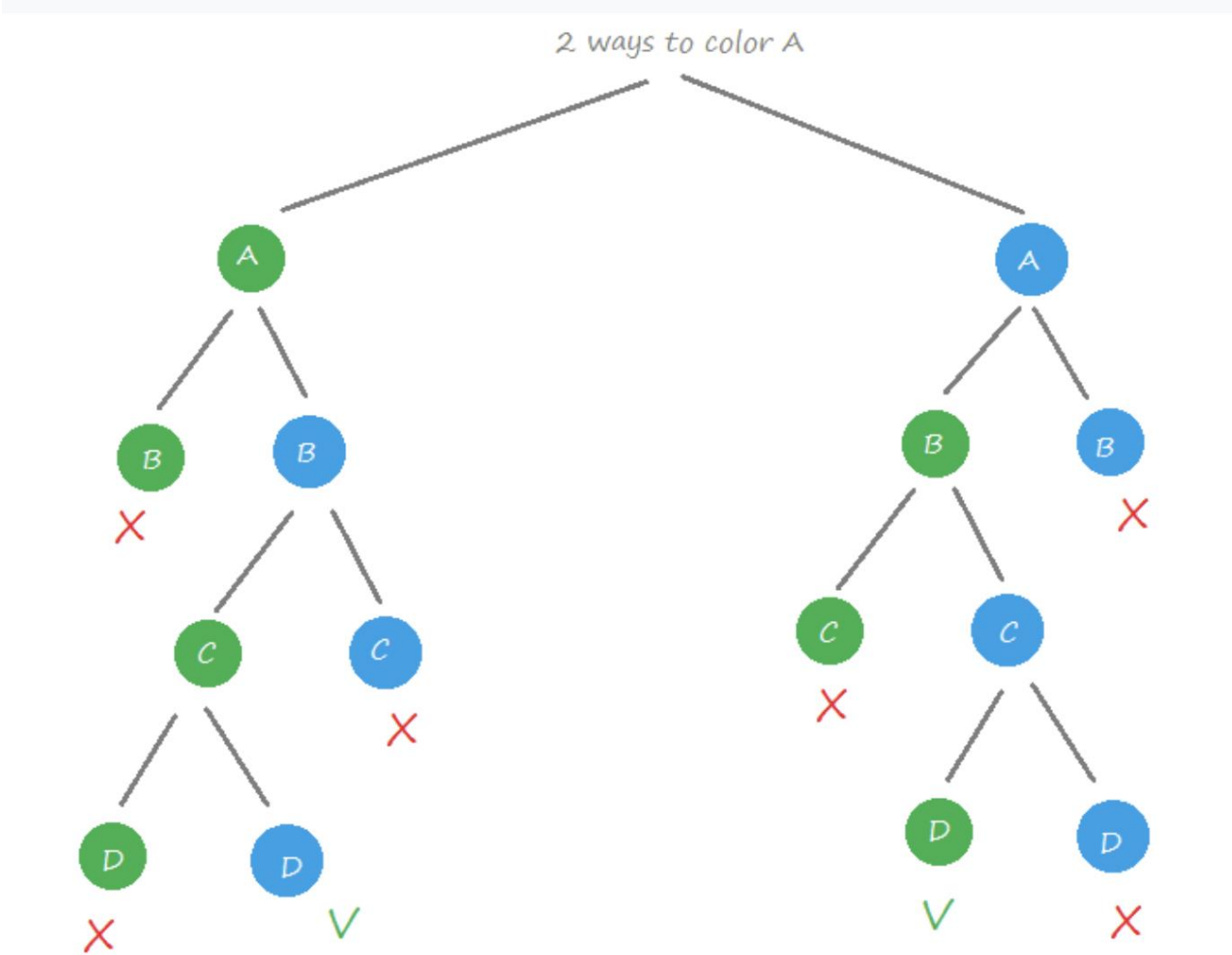
```
Backtracking(N)
        If N is a goal node, return “success”
        Else if N is a leaf node, return “failure”
        For each child C of N,
                If Backtracking(C) == “success”
                        Return “success”
        Return “failure”
```

Coloring Problem

1. Start with an initial coloring of the map, such as assigning the first color to the first region.
2. Select an uncolored region and try to assign it a color.
3. Check if the color assignment is valid, i.e., check if no adjacent regions have the same color.
4. If the color assignment is valid, move to the next uncolored region and repeat steps 2-3.
5. If the color assignment is not valid, backtrack to the previous region and try a different color assignment for that region.
6. If all possible color assignments for a region have been tried and none of them are valid, backtrack to the previous region and try a different color assignment for that region.
7. Repeat steps 2-6 until all regions have been colored or until it is determined that a valid coloring cannot be found.

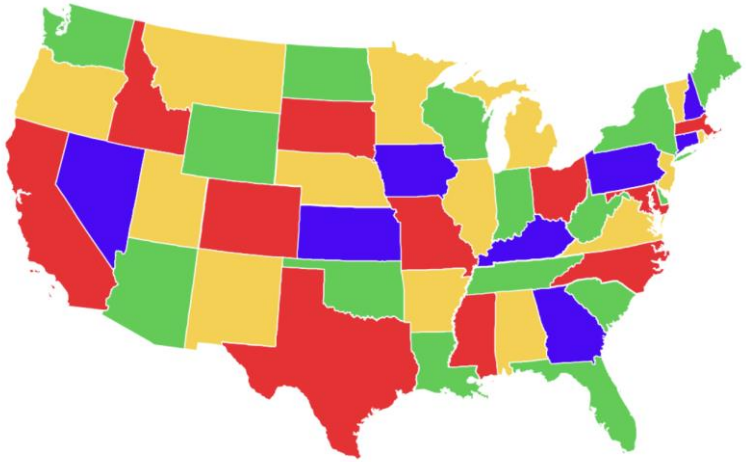
Backtracking





Coloring problem

- Input format:
  - 2D adjacency matrix representation graph[V][V]
  - Number of colors m
- Output format:
  - array color[V] that should have numbers from 1 to m
- Naïve solution: check all possible combinations
  - $m^{|V|}$ combinations
  - If m= 4 and |V| = 64 → billion years to run 😵



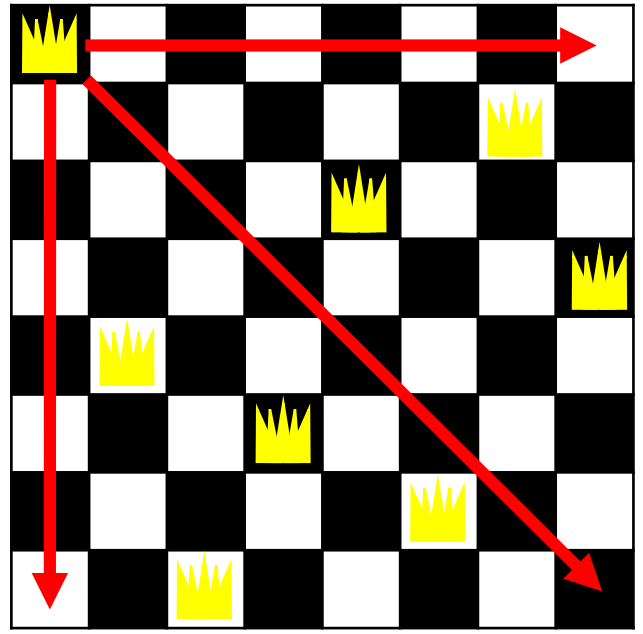
Coloring problem: Backtracking

- Create a recursive function that takes current index
- If the current index is equal to the number of vertices
  - Print the result.
- Assign each color to a vertex (1 to m).
- For every assigned color, check if the configuration is safe, recursively call the function with next index
  - If any recursive function returns True
    - break the loop and return True.
- If no recursive function returns True then return False.

```
bool graphColoringUtil(
    bool graph[V][V], int m,
    int color[], int v)
{
    /* base case: */
    if (v == V)
        return true;

    /* Consider this vertex v and
    try different colors */
    for (int c = 1; c <= m; c++) {
        /* Check if color c to v is fine*/
        if (isSafe(
                v, graph, color, c)) {
            color[v] = c;

            /* recur to assign colors to
            rest of the vertices */
            if (
                graphColoringUtil(
                    graph, m, color, v + 1)
                == true)
                return true;

            /* If c is not successful -> remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned \*/
    return false;
}
```

```
bool isSafe(
    int v, bool graph[V][V],
    int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (
            graph[v][i] && c == color[i])
            return false;
    return true;
}
```

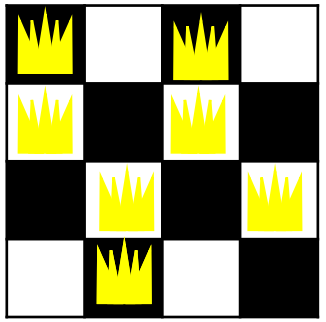
The Eight Queens Problem



- A chessboard has 8 rows and 8 columns
- A queen can move within its row or its column or along its diagonal
- Place 8 queens on the board
  - No queen can attack any other queen in a move

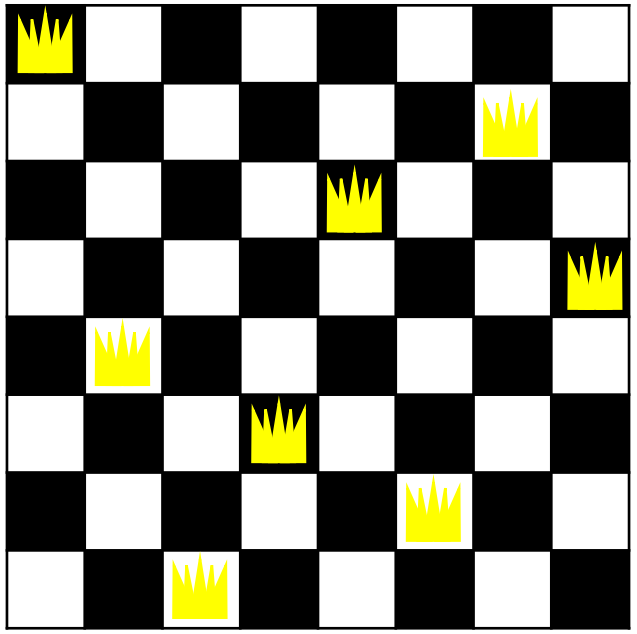
- Exhausting search will take $\binom{64}{8} = 4.43$ billion ways
- Each row can contain exactly one queen: $8! = 40{,}320$
- Better algorithm?

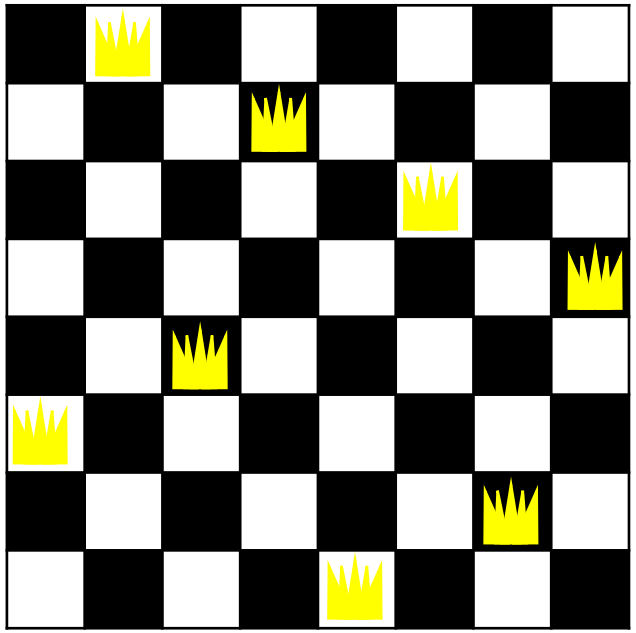
Backtracking Algorithm

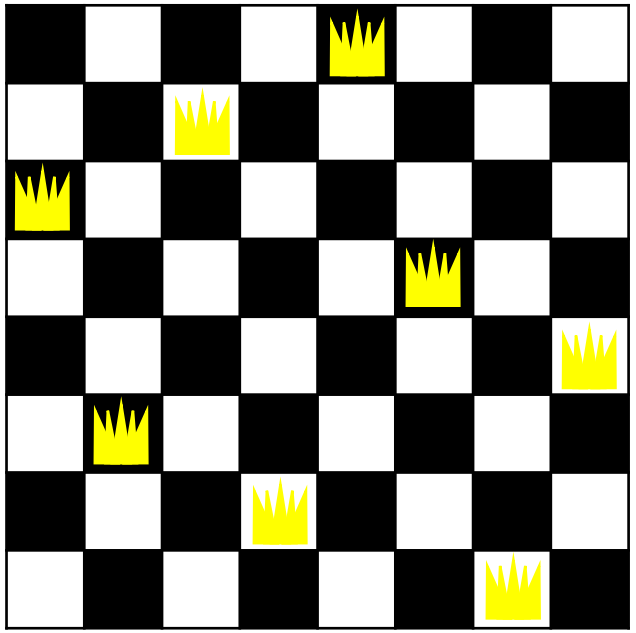


1. Starts by placing a queen on the top left corner of the chess board.
2. Places a queen on the second column and moves her until a place where she cannot be hit by the queen on the first column.
3. Places a queen on the third column and moves her until she cannot be hit by either of the first two queens and so on.
4. If there is no place for the i^{th} queen, the program backtracks to move the $(i - 1)^{\text{th}}$ queen.
5. If the $(i - 1)^{\text{th}}$ queen is at the end of the column, the program removes the queen and backtracks to the $(i - 2)$ column and so on.
6. If the current column is the last column and a safe place has been found for the last queen, then a solution to the puzzle has been found.

Backtracking Algorithm



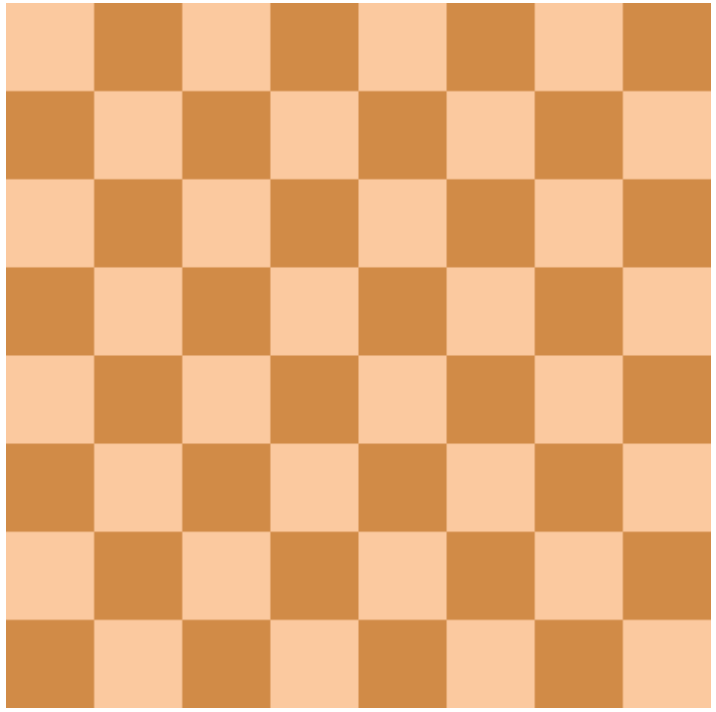




- If the current column is the first column and its queen is being moved off the board then all possible configurations have been examined, all solutions have been found, and the algorithm terminates.
- This puzzle has **92** solutions.

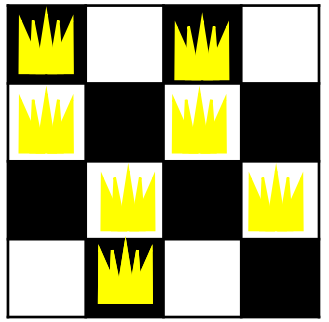
N-Queens Problem

n	Possible Solutions
4	2
5	10
6	4
7	40
8	92
10	724
12	14,200
15	2,279,184
20	39,029,188,884



The Eight Queens Problem's Algorithm

```
function NQueens(Board[N][N], Column)
    if Column >= N then return true                                  ▷ Solution is found
    else
        for i ← 1, N do
            if Board[i][Column] is safe to place then
                Place a queen in the square
                if NQueens(Board[N][N], Column + 1) then return true ▷ Solution is found
                end if
                Delete the queen
            end if
        end for
    end if
return false                                                         ▷ no solution is found
end function
```



Backtracking Algorithm Problems

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Maze problem
- Coloring Problem
- N Queen Problem
- Sudoku
- Hamiltonian Cycle
  - Hamiltonian Path is a path that visits each vertex exactly once in an undirected graph.
  - If the last vertex has an edge back to the first vertex, then a Hamiltonian Cycle exists.

Backtracking Algorithm

- Trying all possible permutations
- Usually not all permutations are worth to investigate
  - You may terminate it in the halfway of your searching
- It is a “not-very-smart” approach

```
Backtracking(n)
        Base case: return true

        for 1 to n
                do something/move forward
                if (Backtracking(n-1)) return true
                reverse whatever you have done earlier (backtracking)
        return false;
```

Summary

- Backtracking
  - Coloring Problem
  - N Queens Problem
  - Hamilton Cycle Problem