

Discrete Optimization

Constraint Programming: Part I

Goal of the Lecture

- ▶ Basic introduction to constraint programming

Constraint Programming

- ▶ Computational paradigm
 - use constraints to reduce the set of values that each variable can take
 - remove values that cannot appear in any solution

Constraint Programming

► Computational paradigm

- use constraints to reduce the set of values that each variable can take
- remove values that cannot appear in any solution

► Modeling methodology

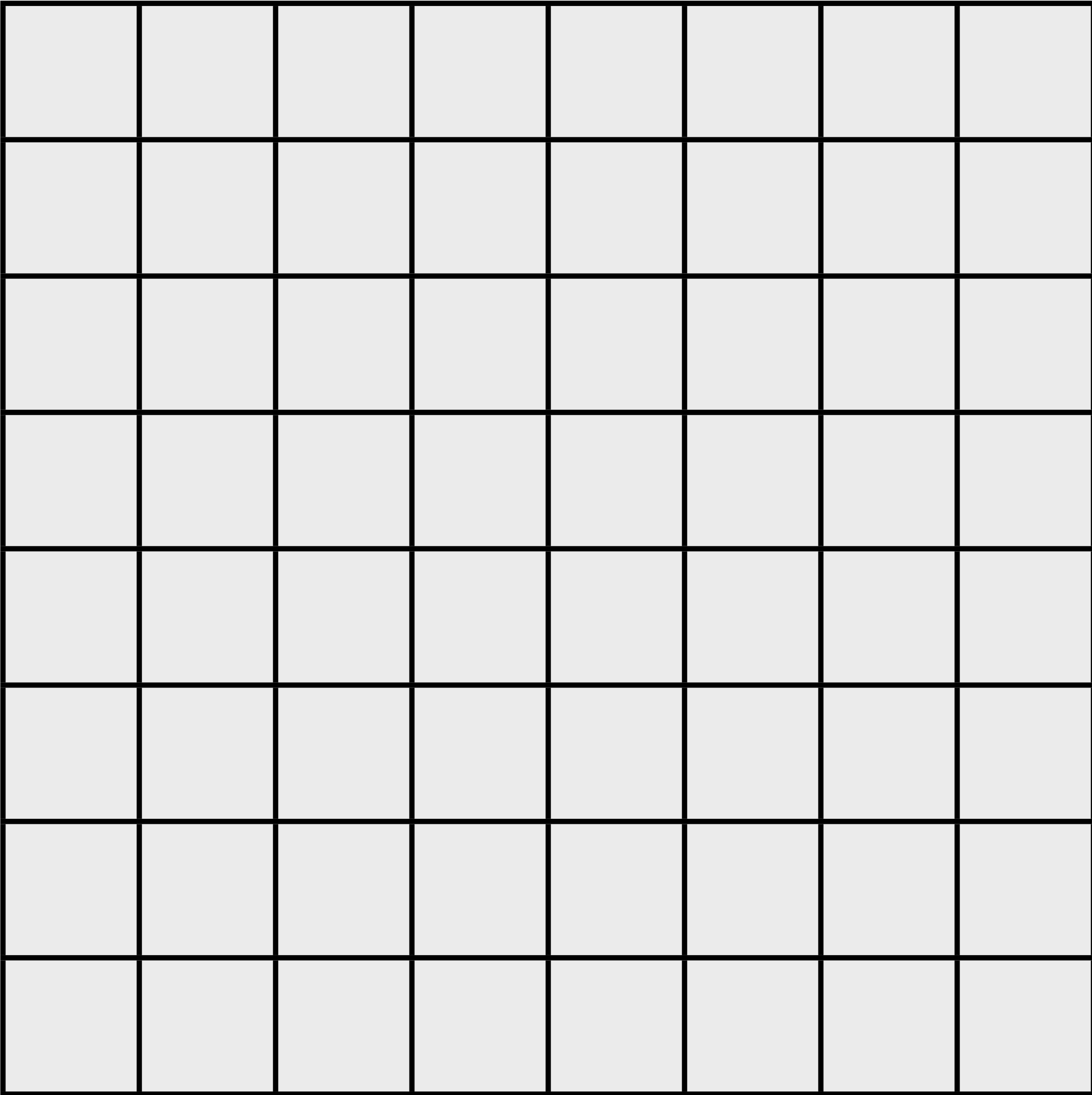
- convey the structure of the problem as explicitly as possible
- express substructures of the problem
- give solvers as much information as possible

The 8-Queens Problem

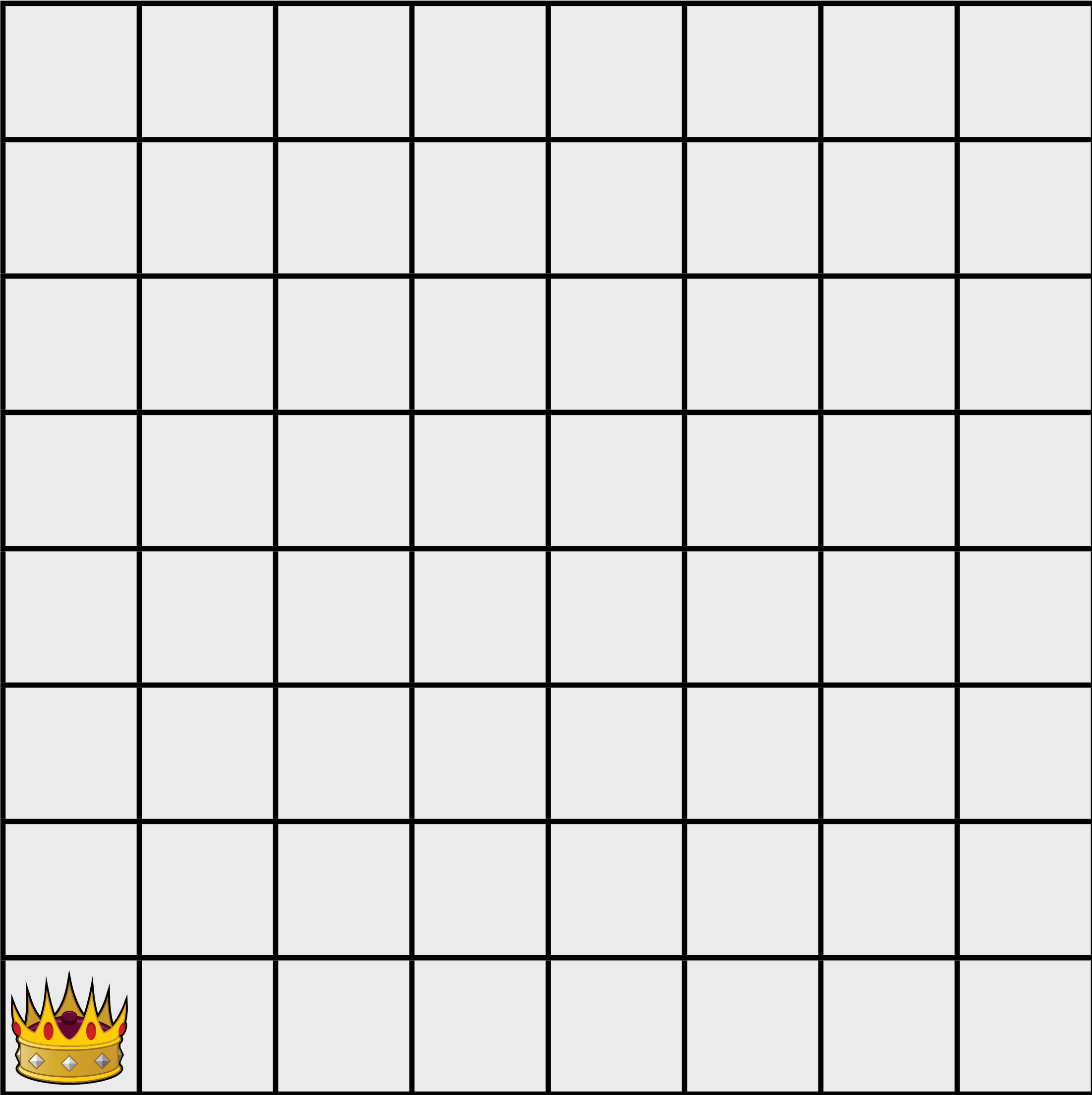
► Specification

- place 8 queens on a chessboard so that none of them attack each other
- two queens attack each other if they are on the same column, row, or diagonal

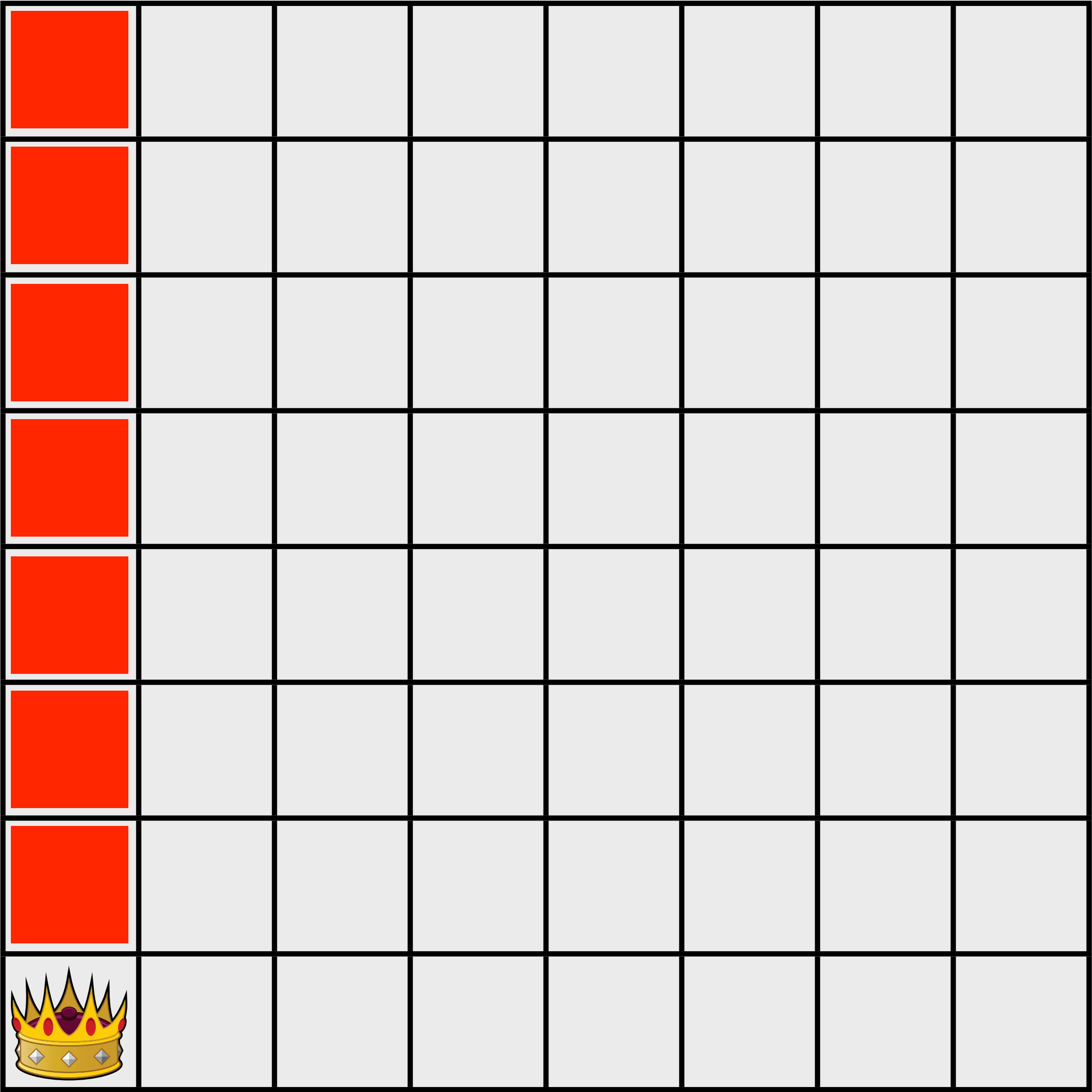
The 8-Queens Problem



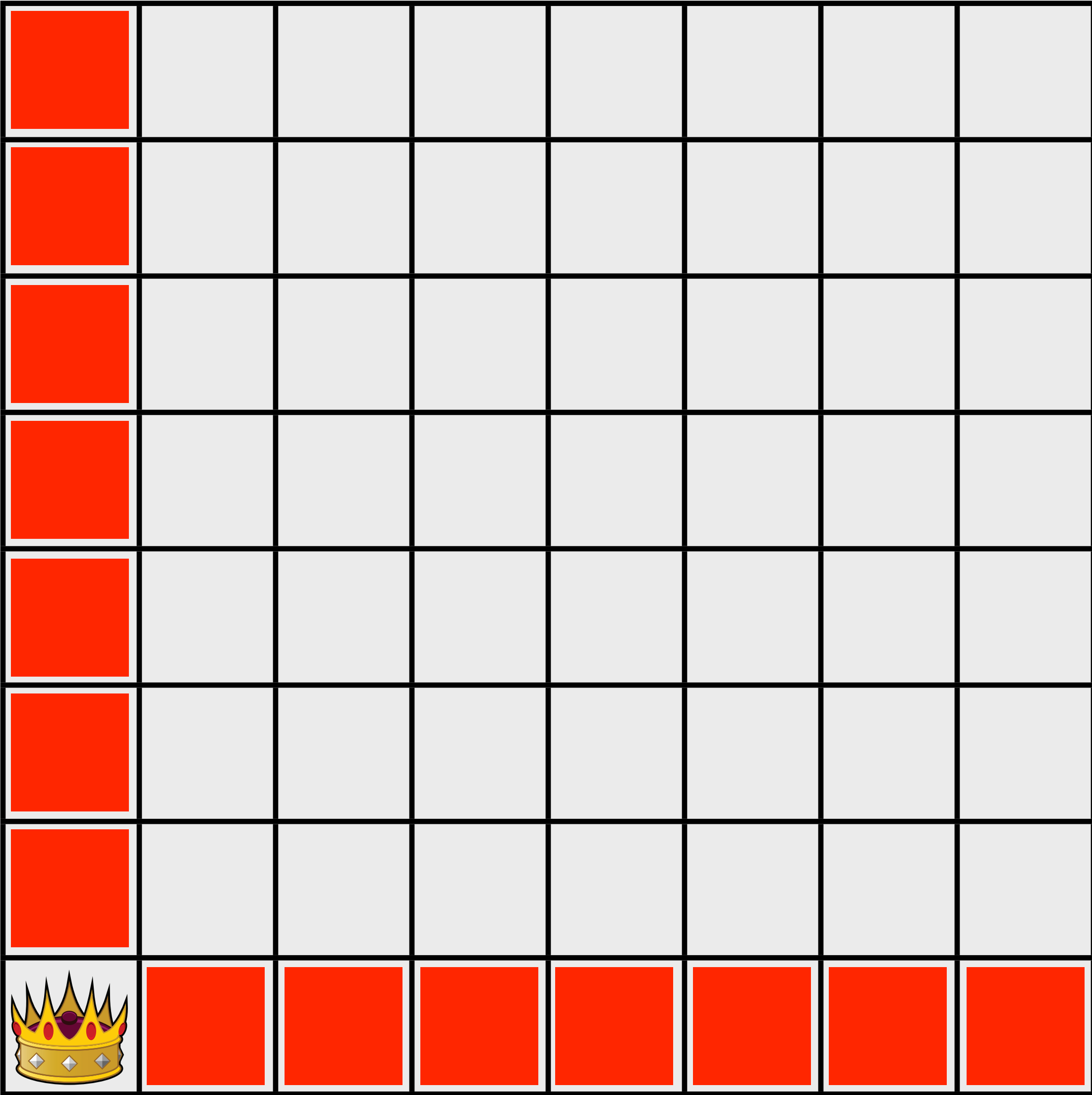
The 8-Queens Problem



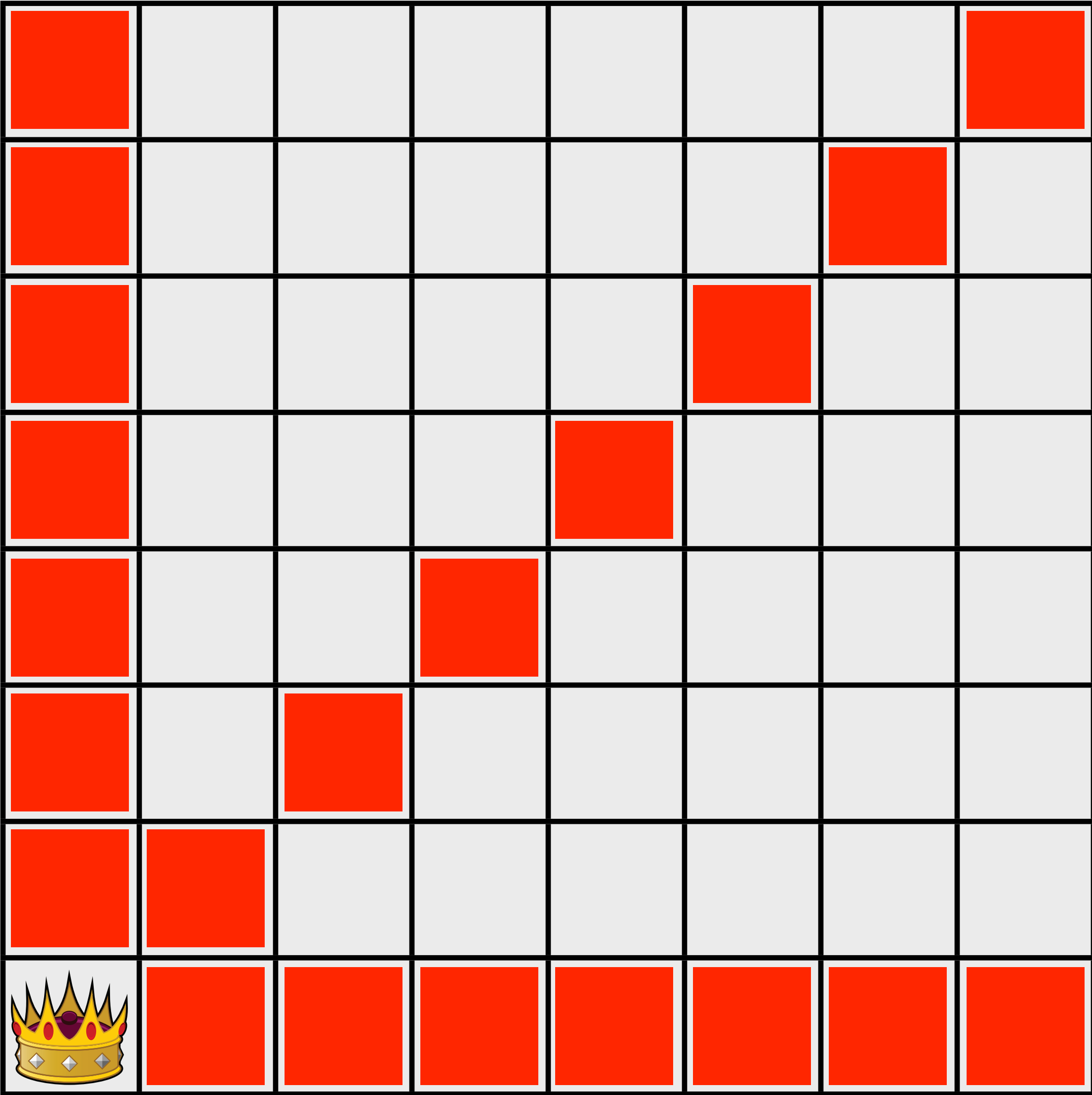
The 8-Queens Problem



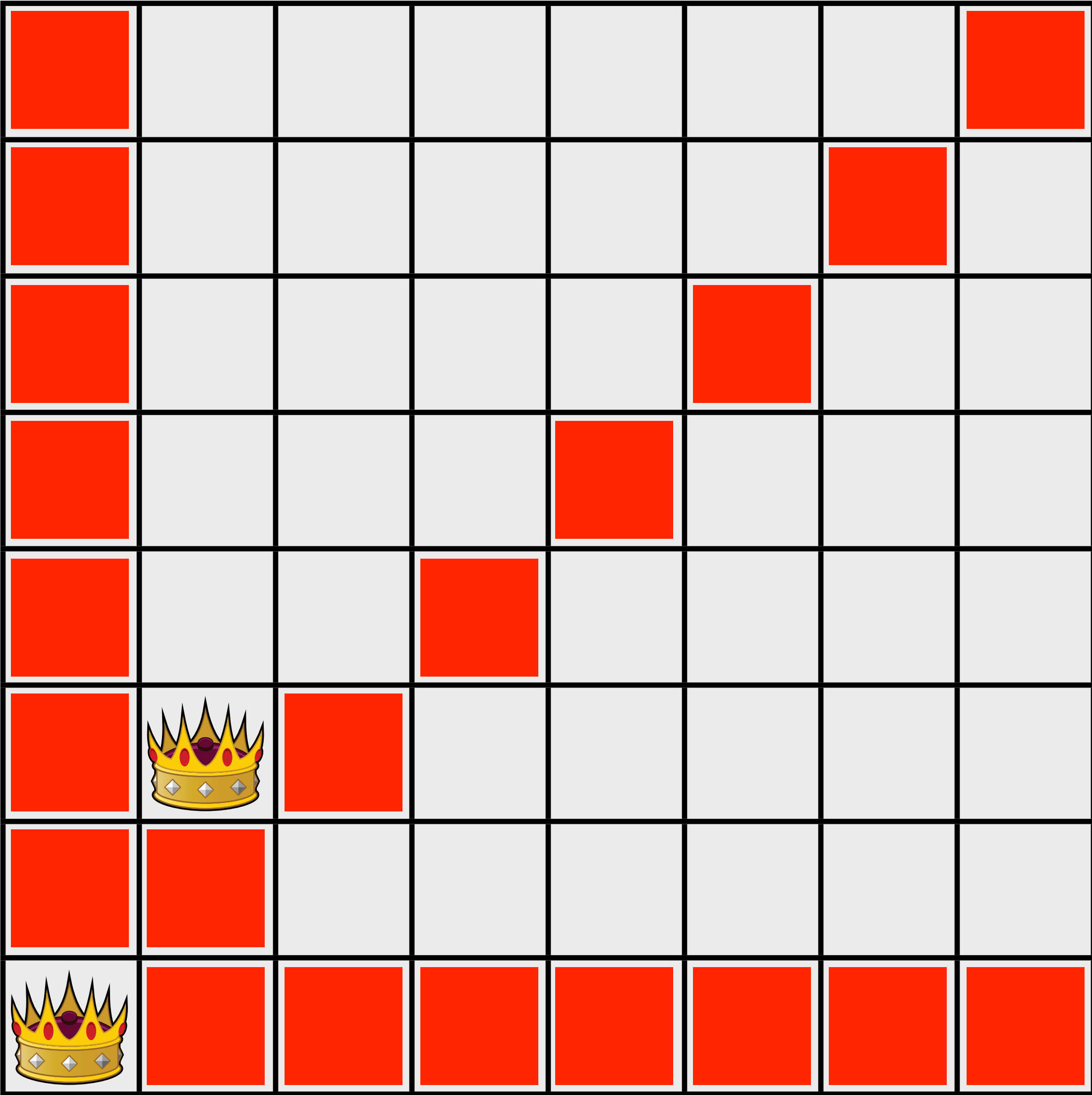
The 8-Queens Problem



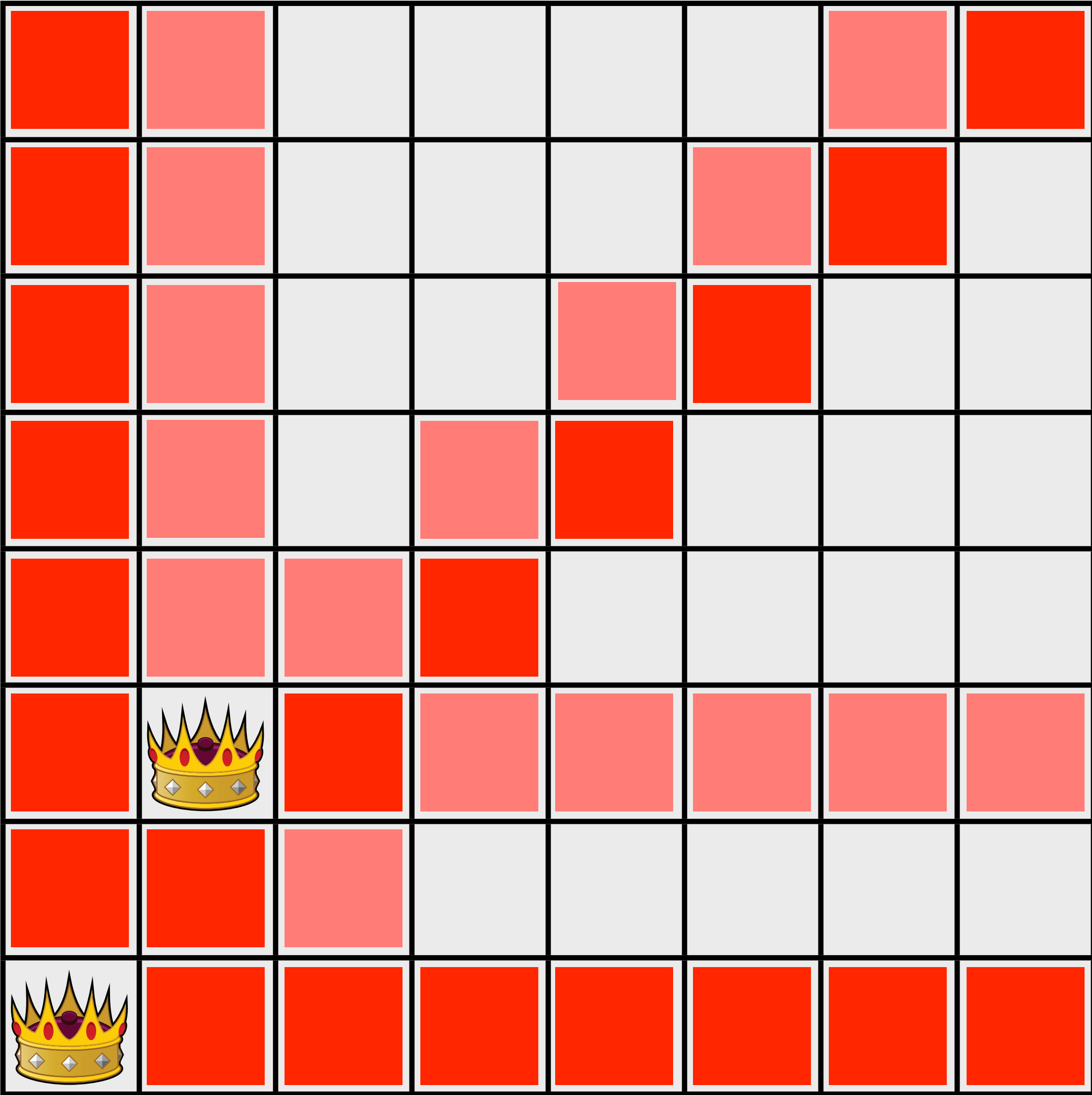
The 8-Queens Problem



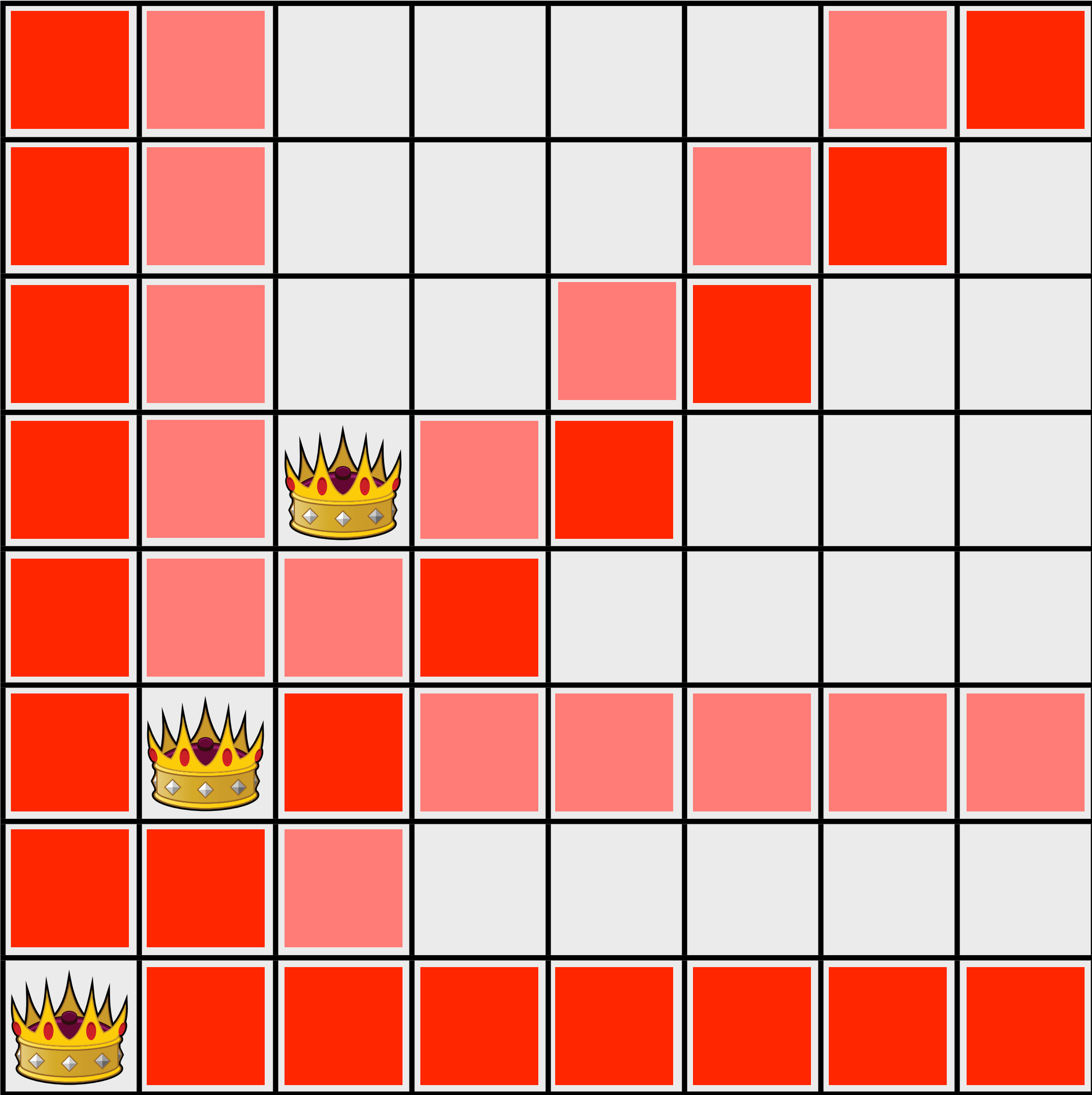
The 8-Queens Problem



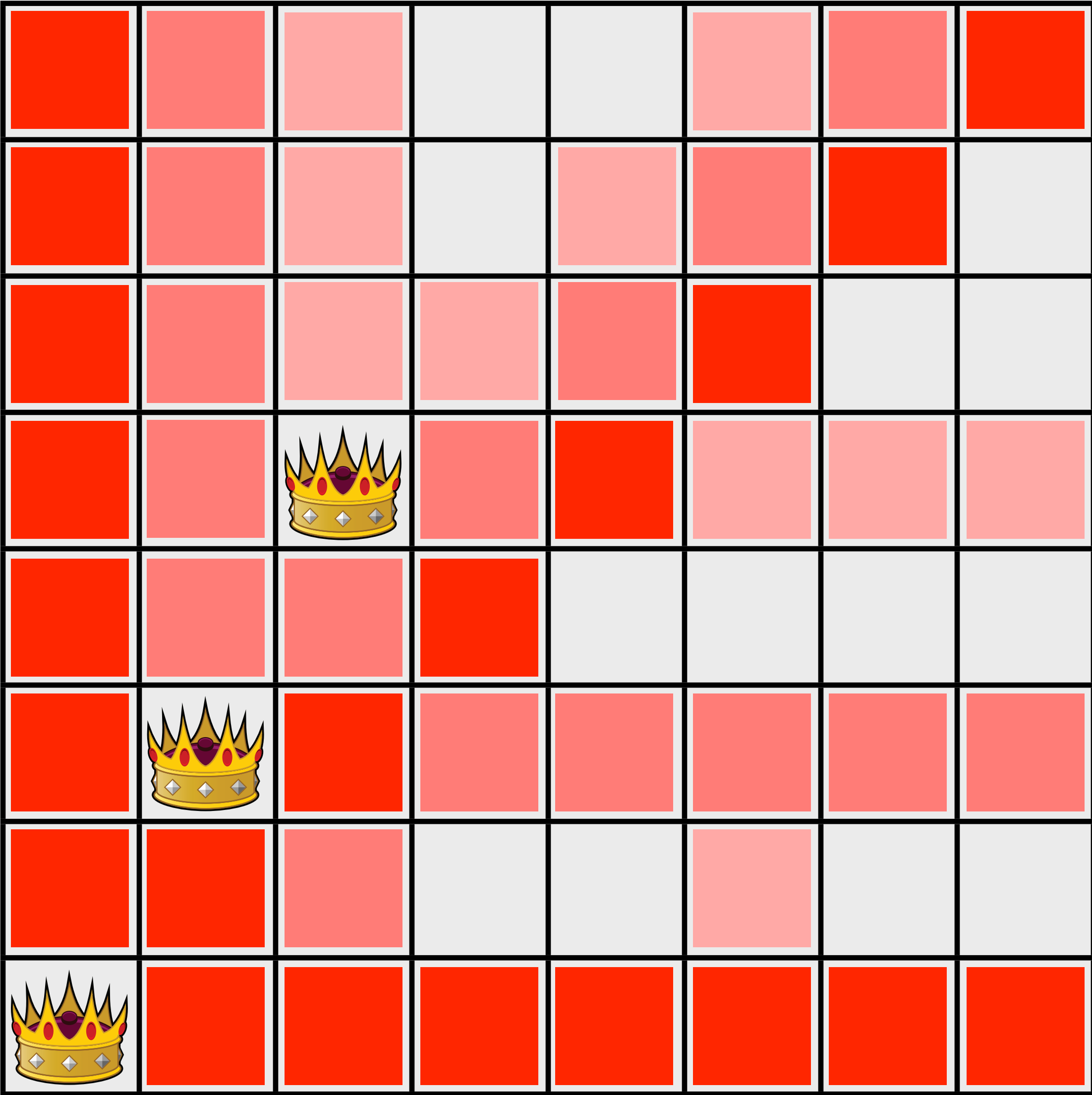
The 8-Queens Problem



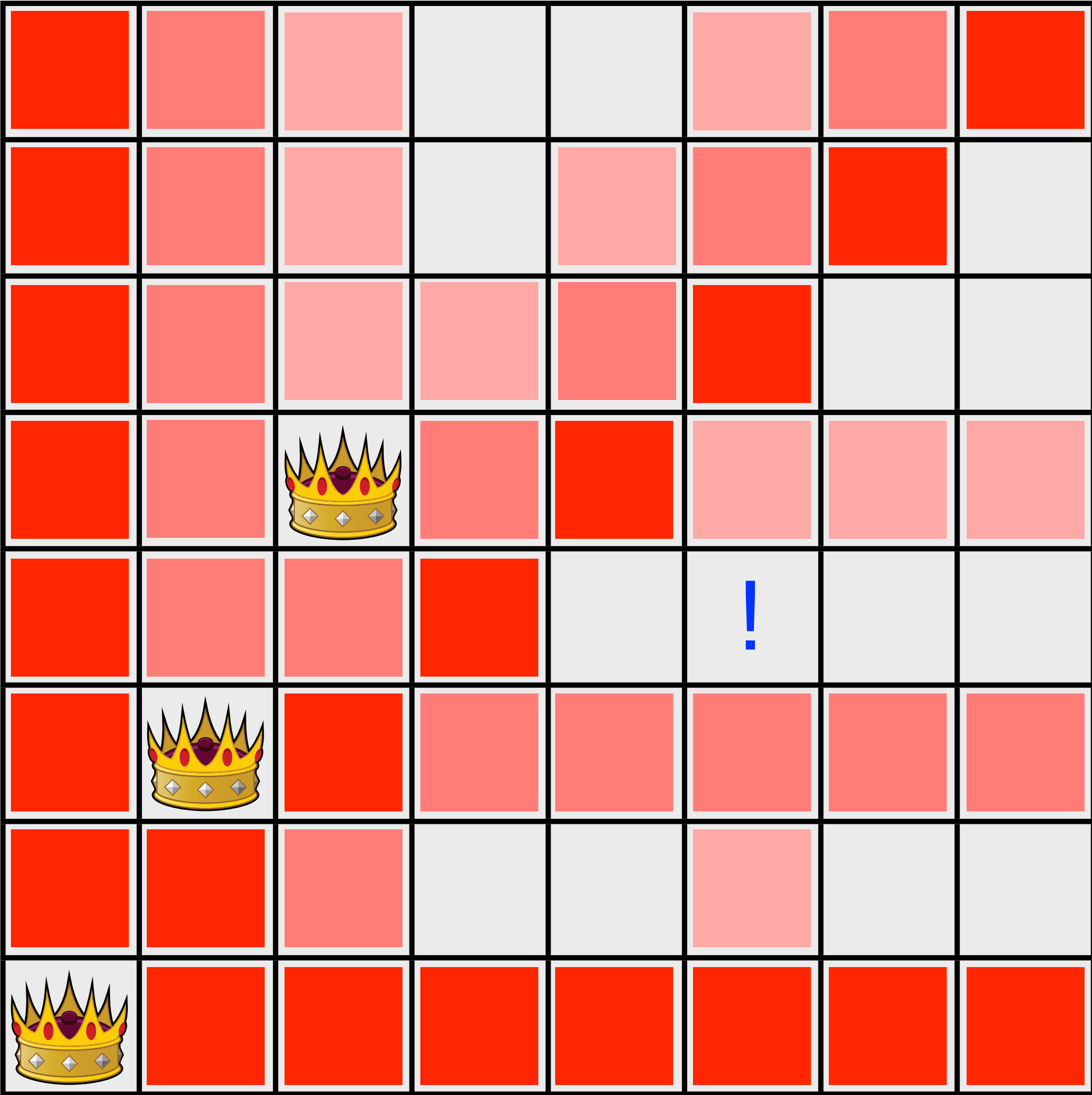
The 8-Queens Problem



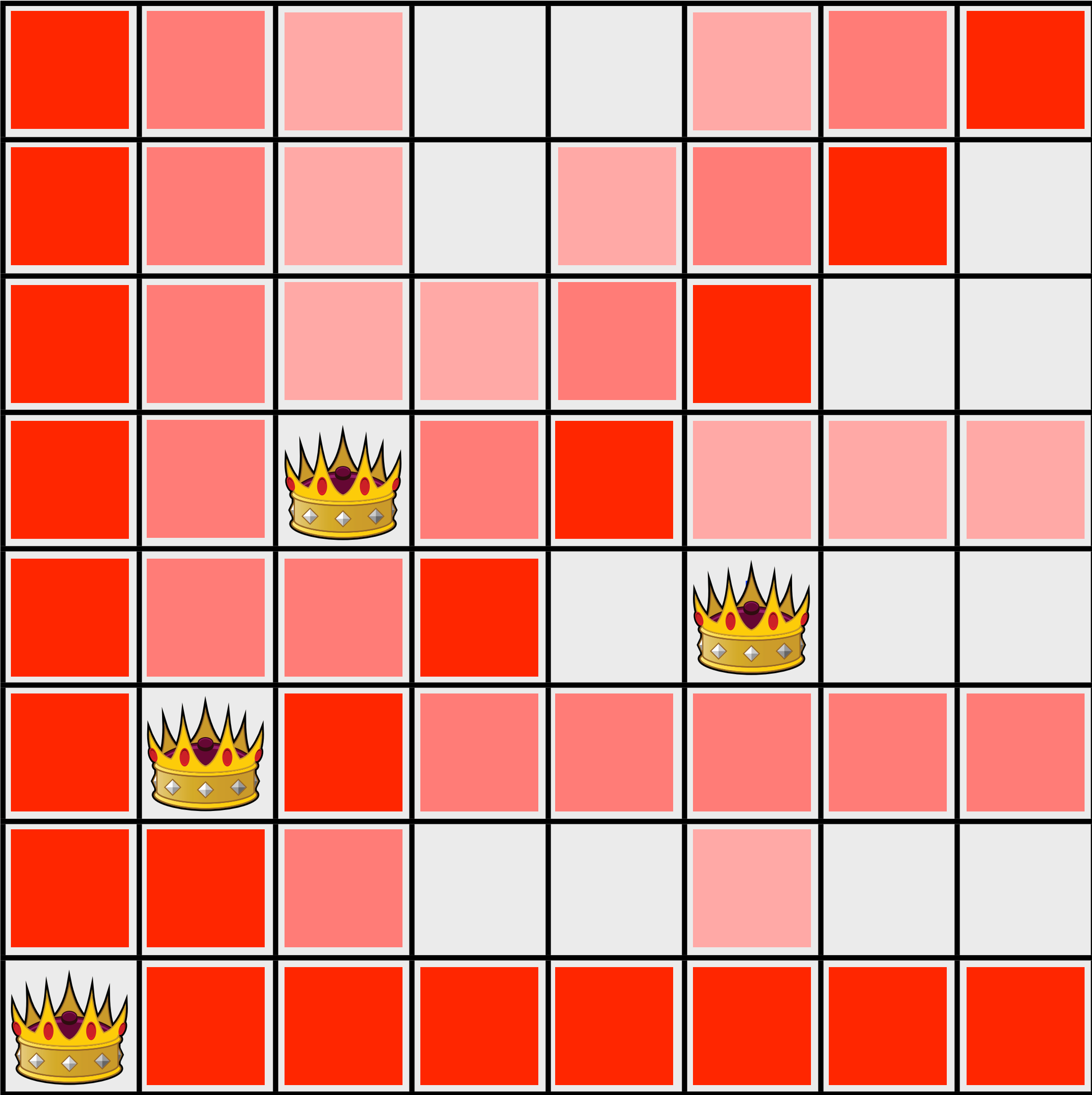
The 8-Queens Problem



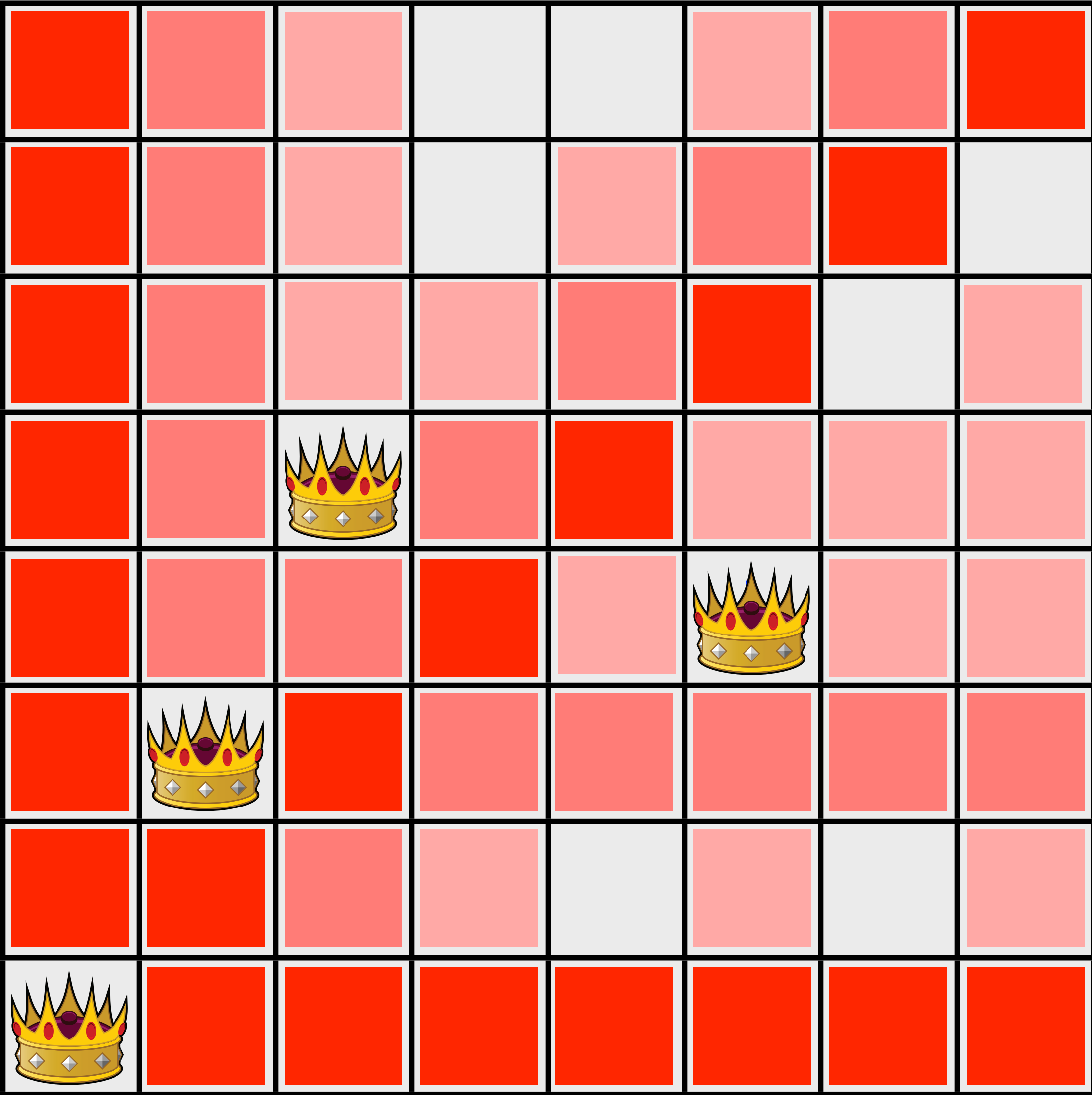
The 8-Queens Problem



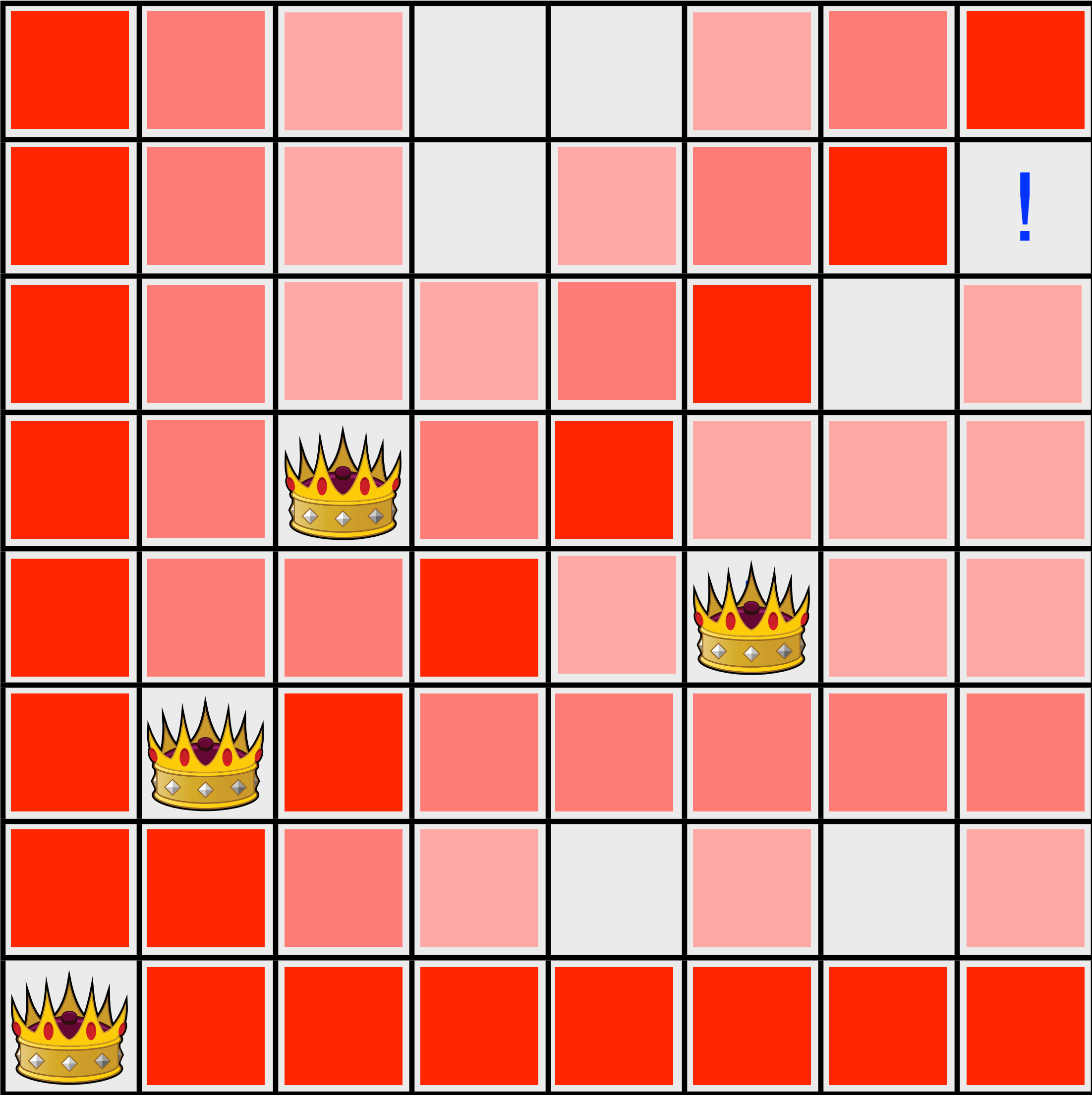
The 8-Queens Problem



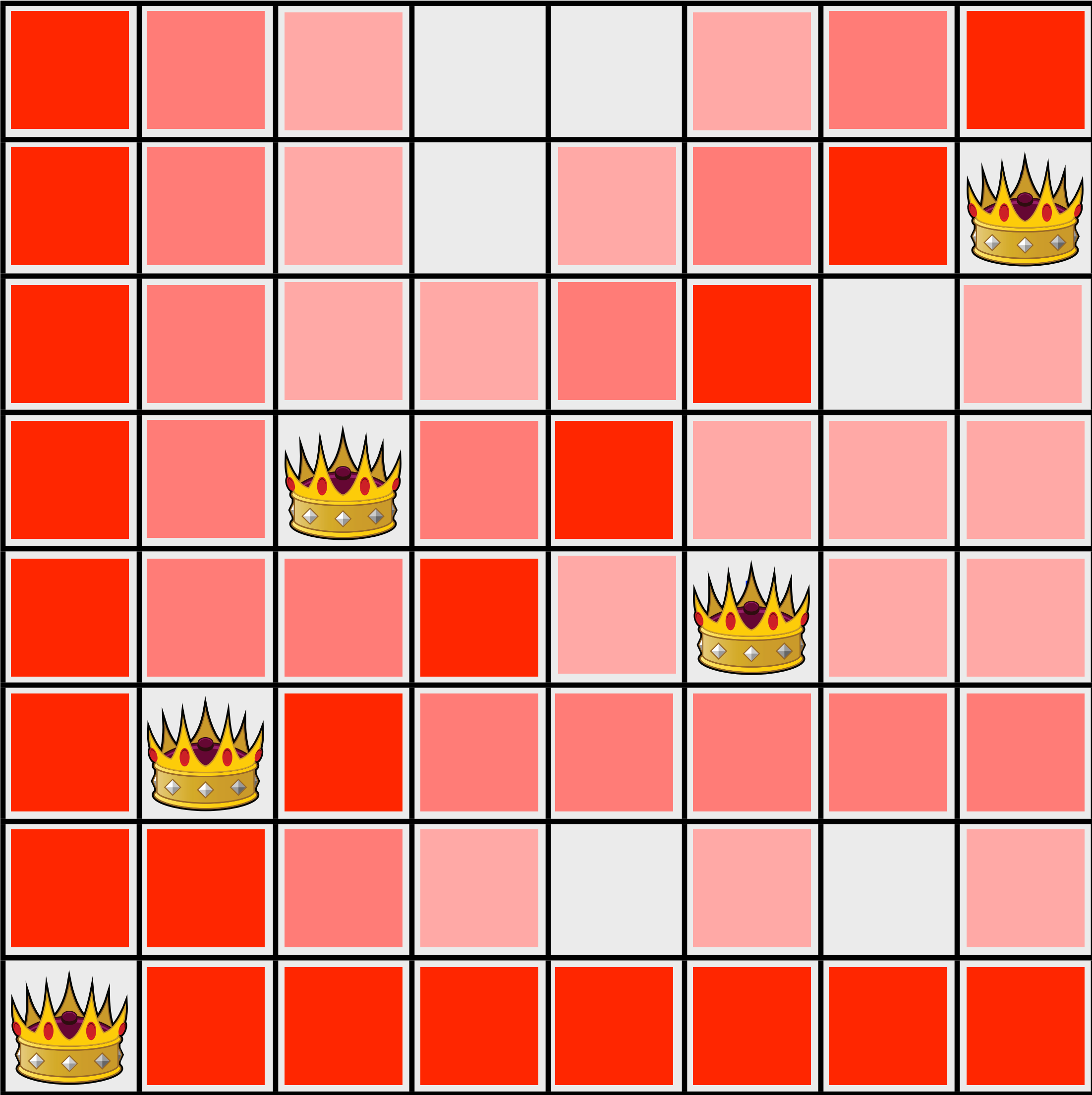
The 8-Queens Problem



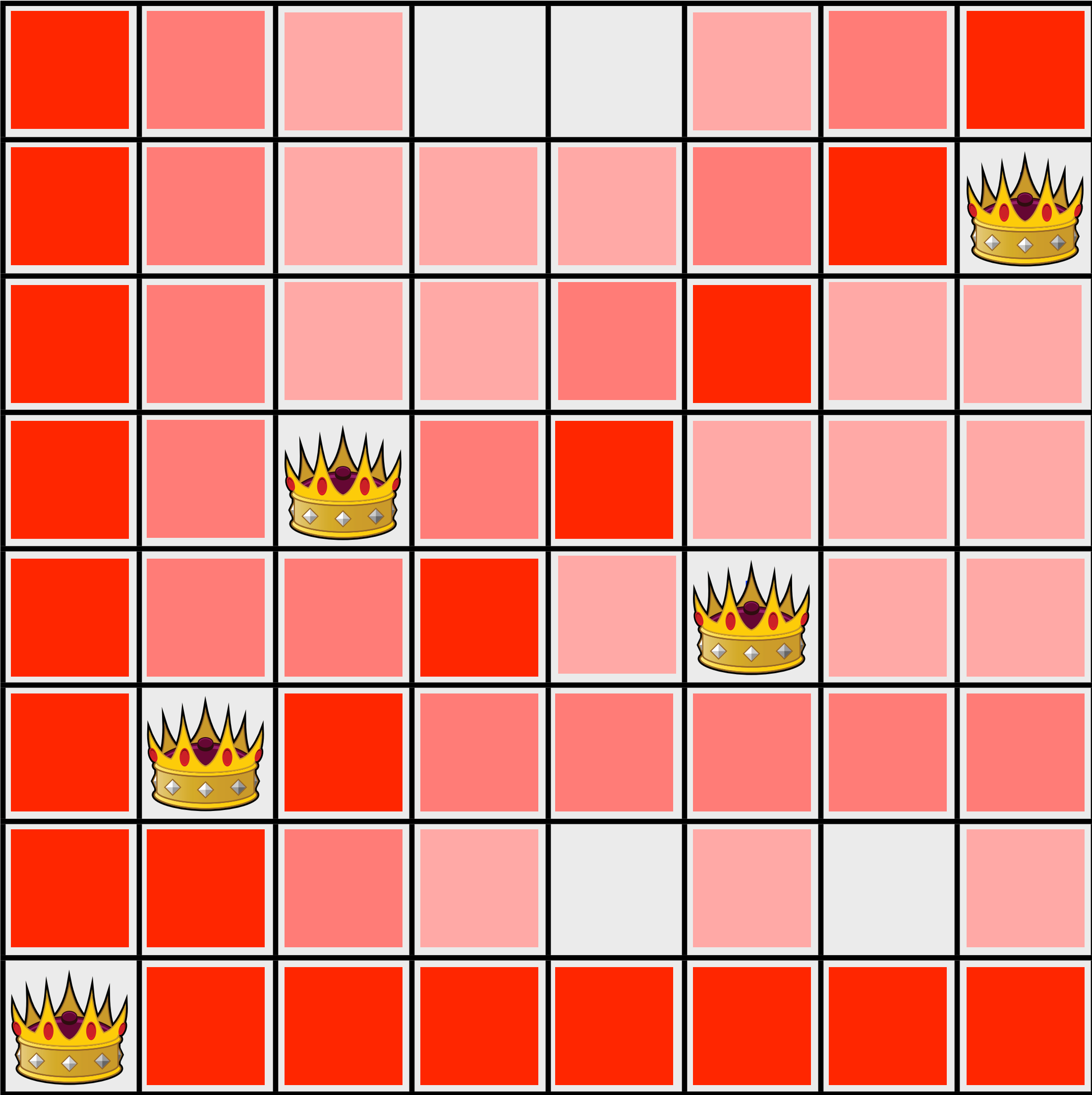
The 8-Queens Problem



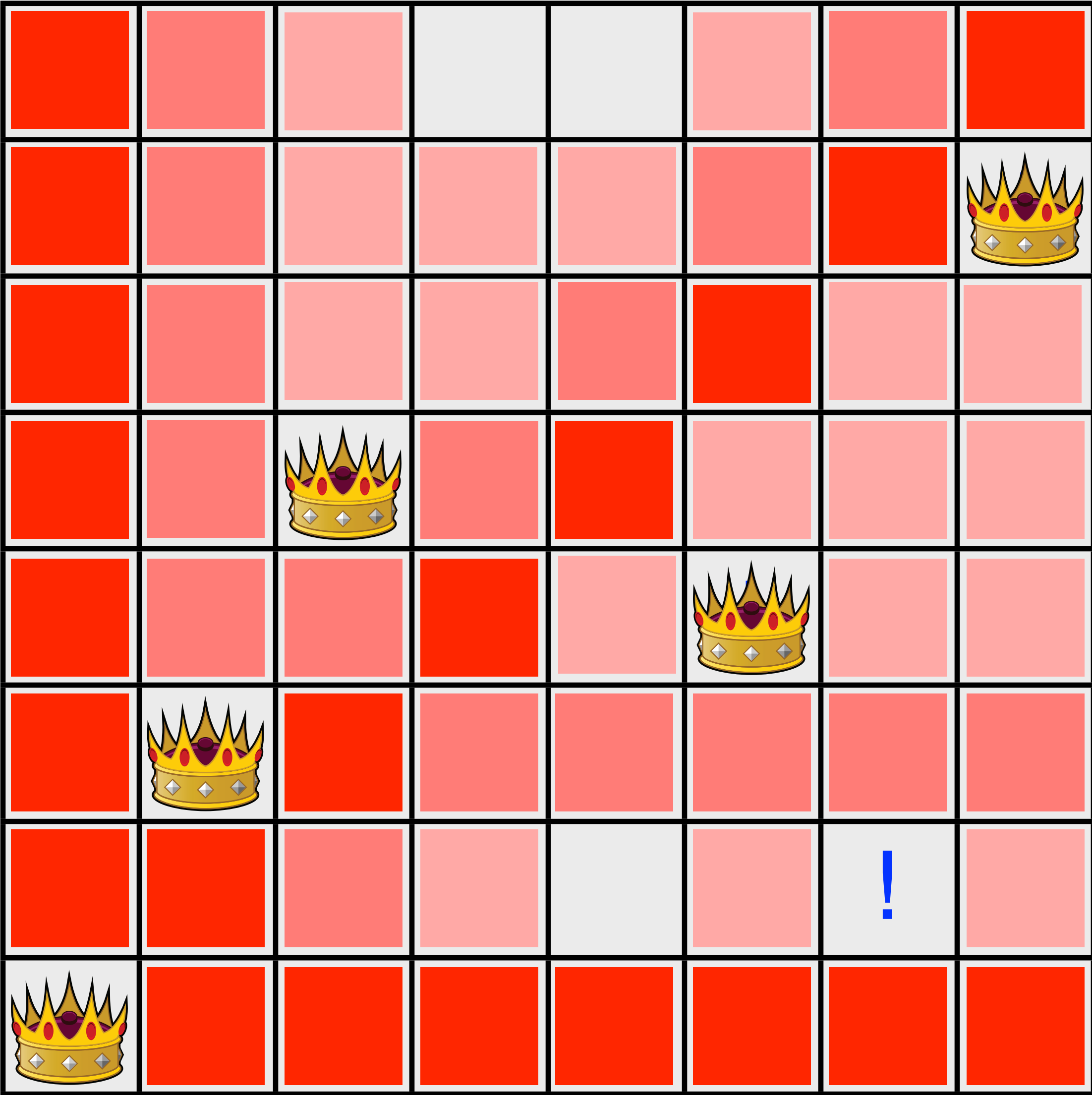
The 8-Queens Problem



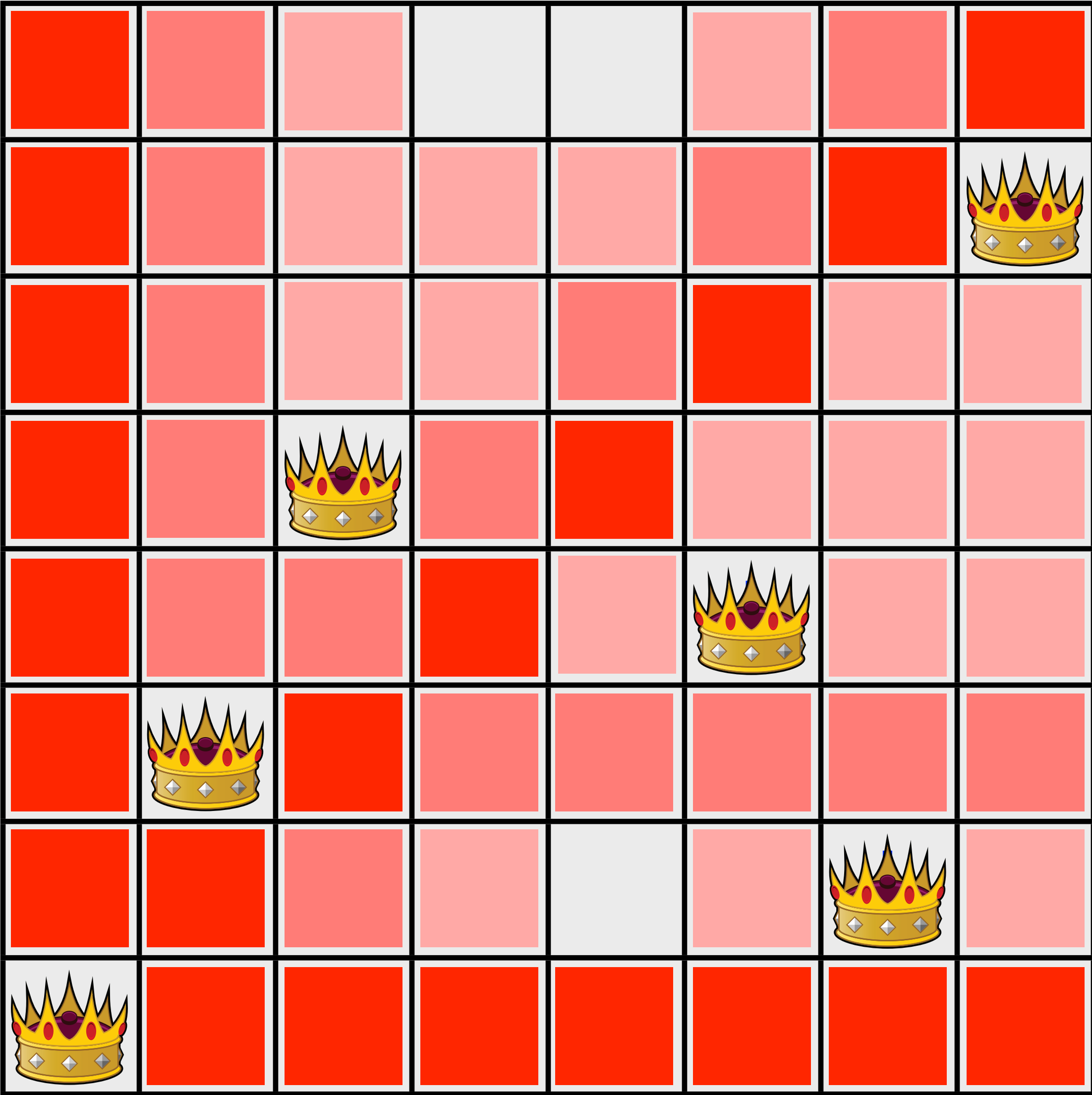
The 8-Queens Problem



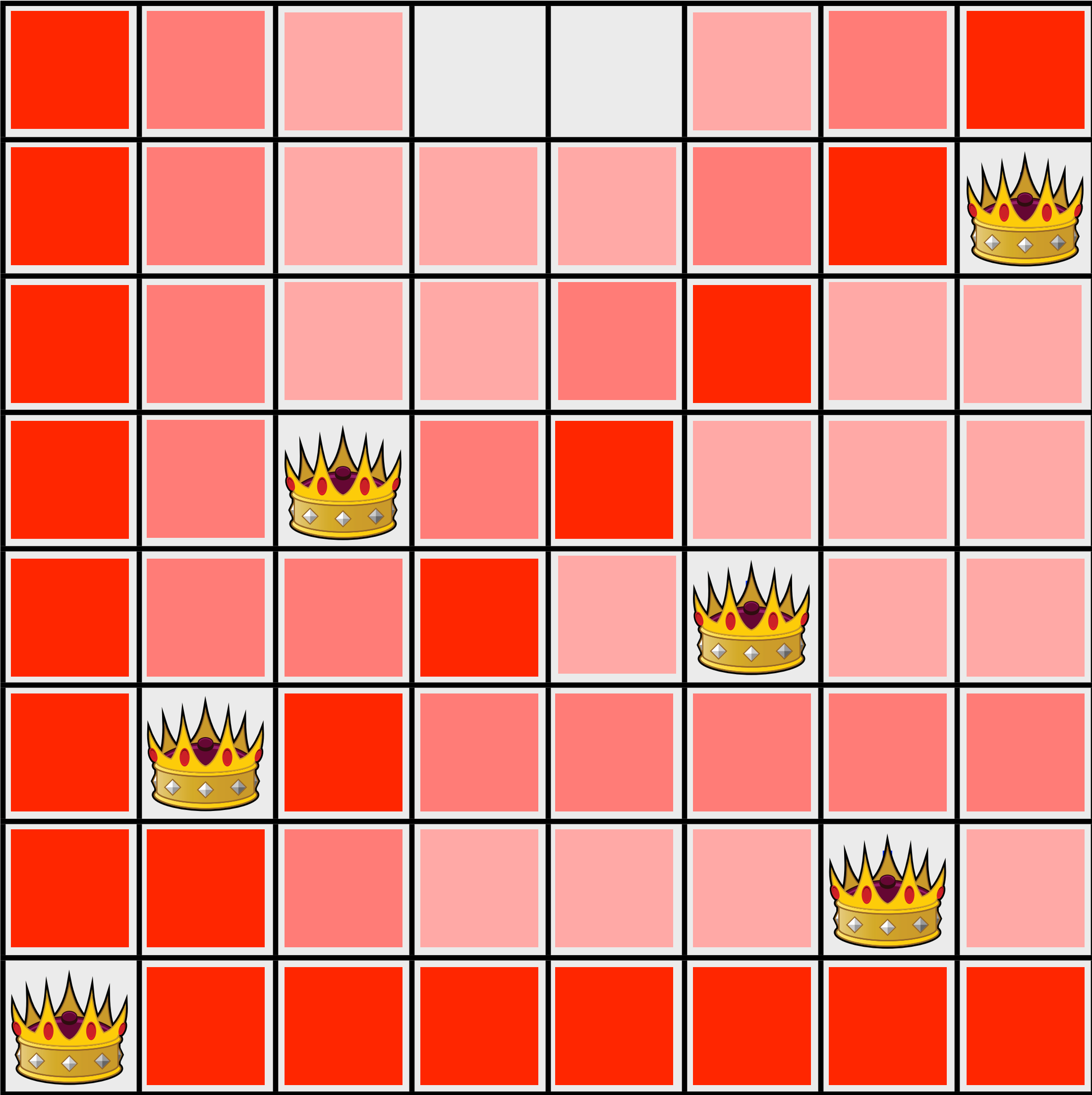
The 8-Queens Problem



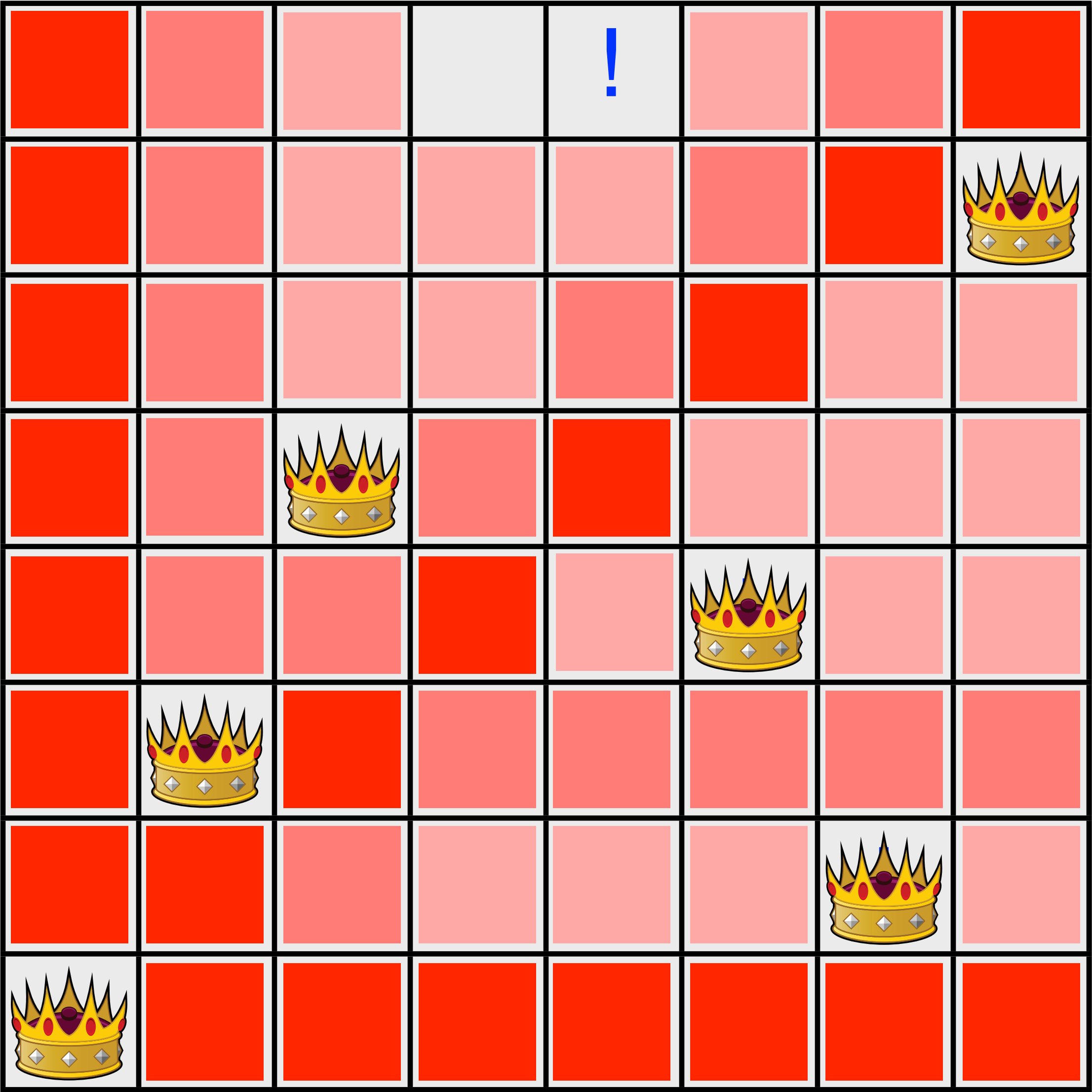
The 8-Queens Problem



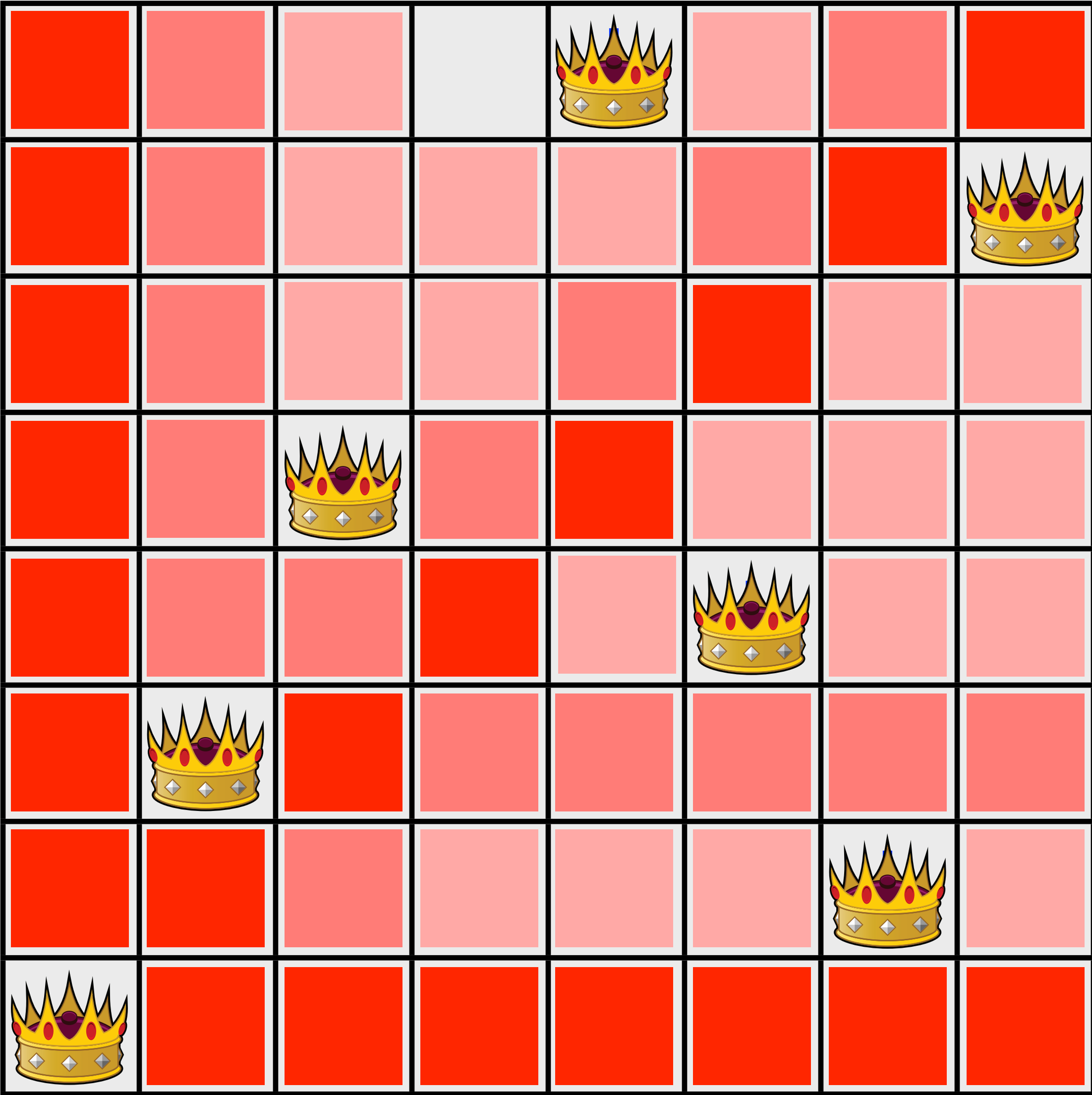
The 8-Queens Problem



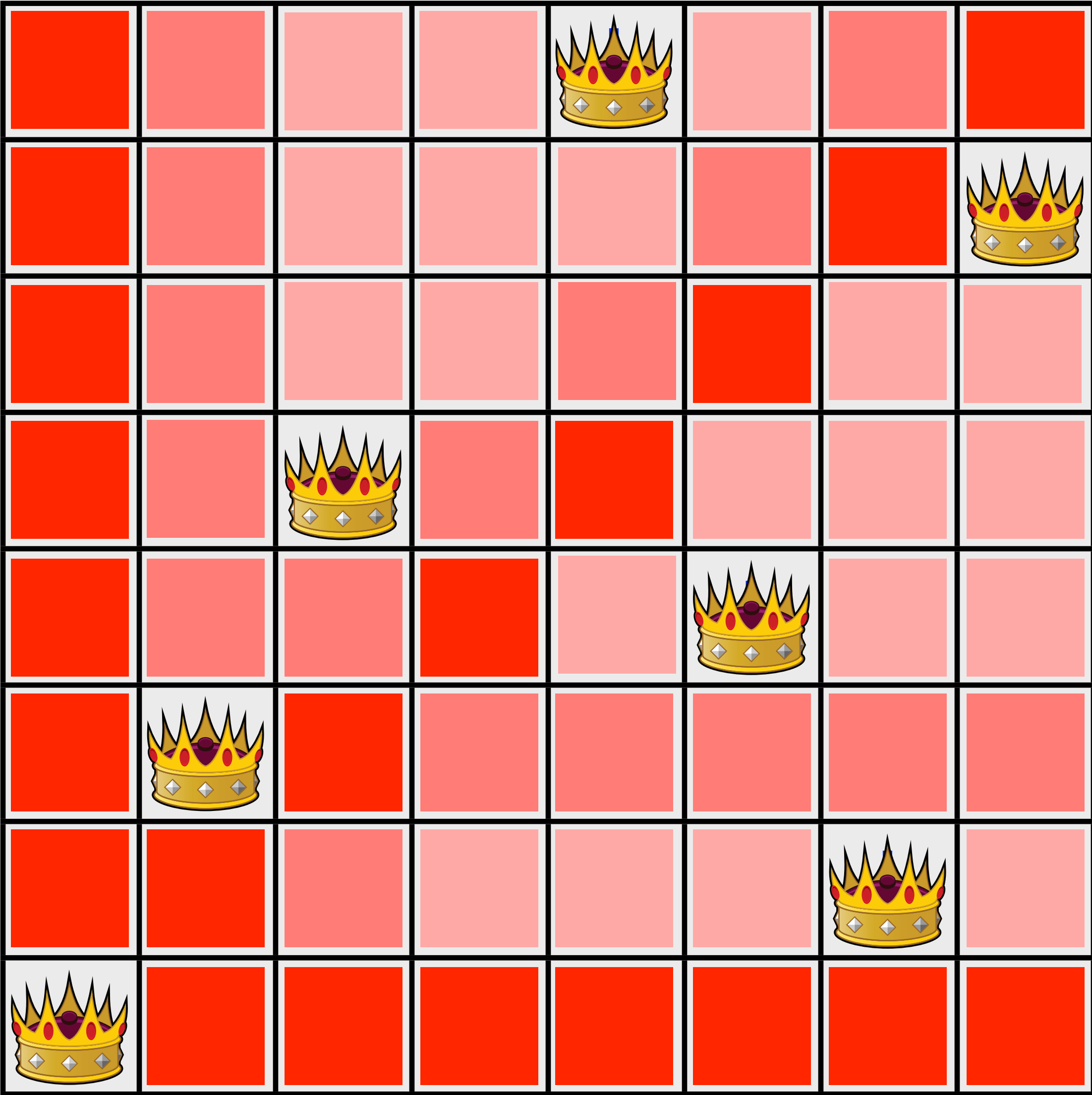
The 8-Queens Problem



The 8-Queens Problem

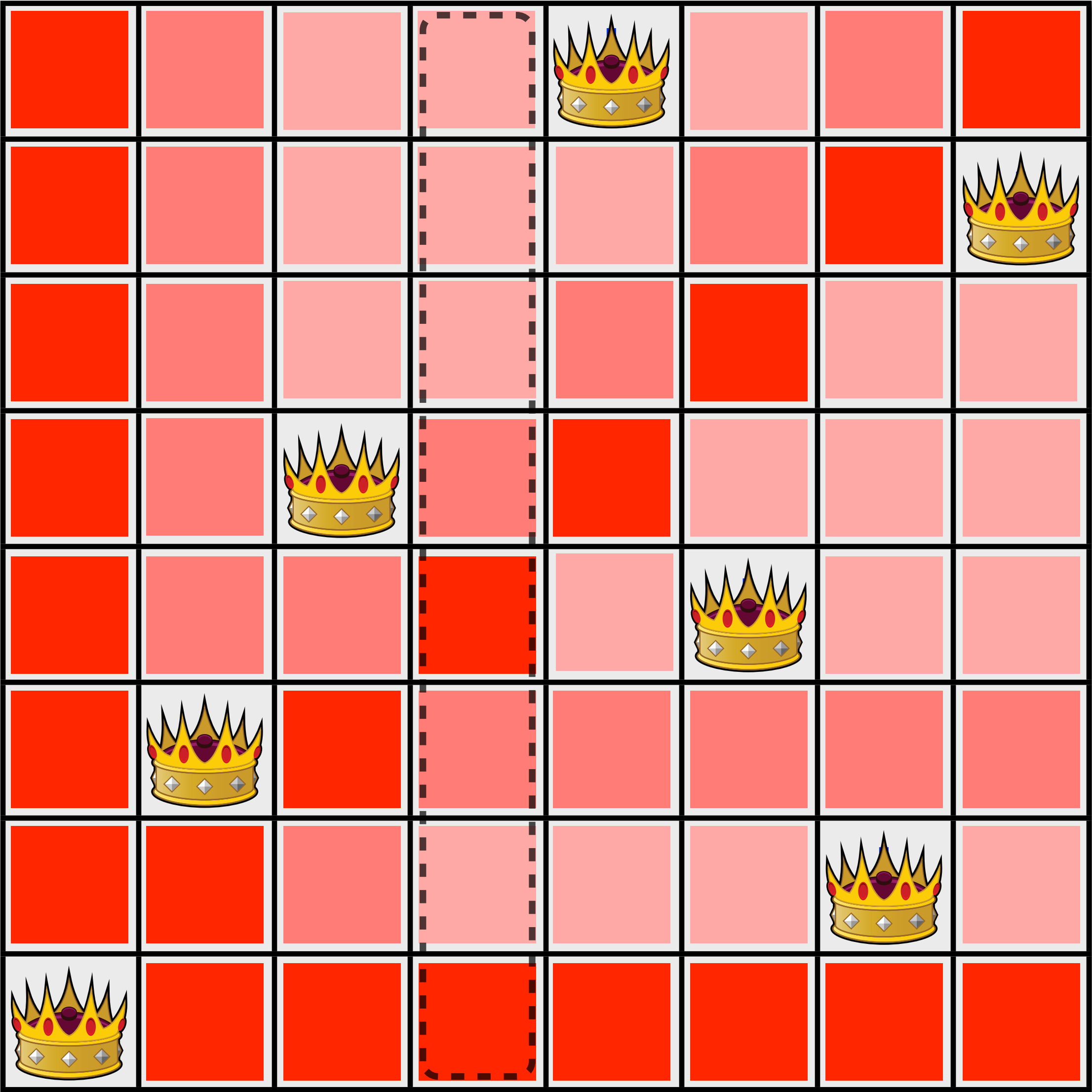


The 8-Queens Problem



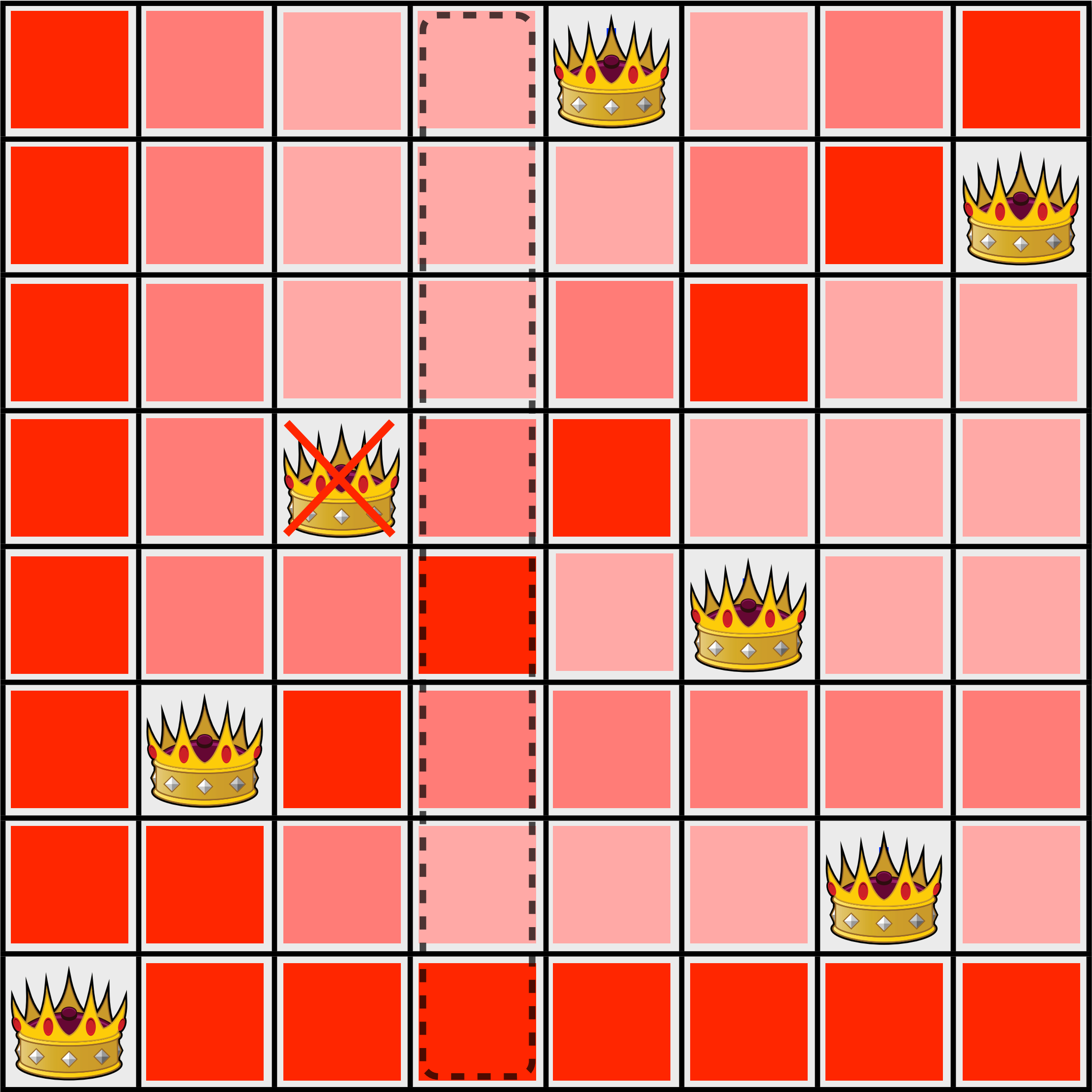
The 8-Queens Problem

Failure!

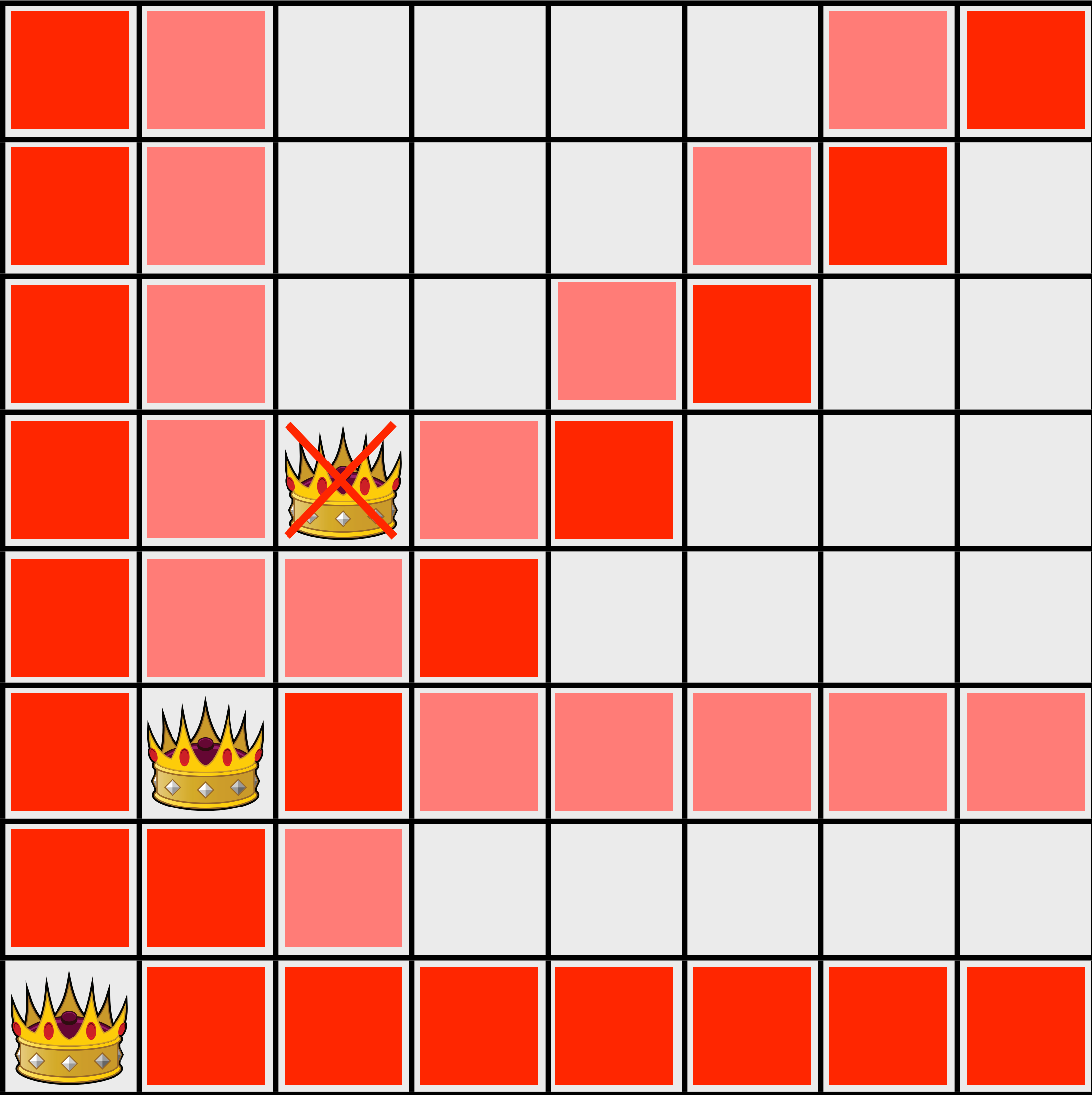


The 8-Queens Problem

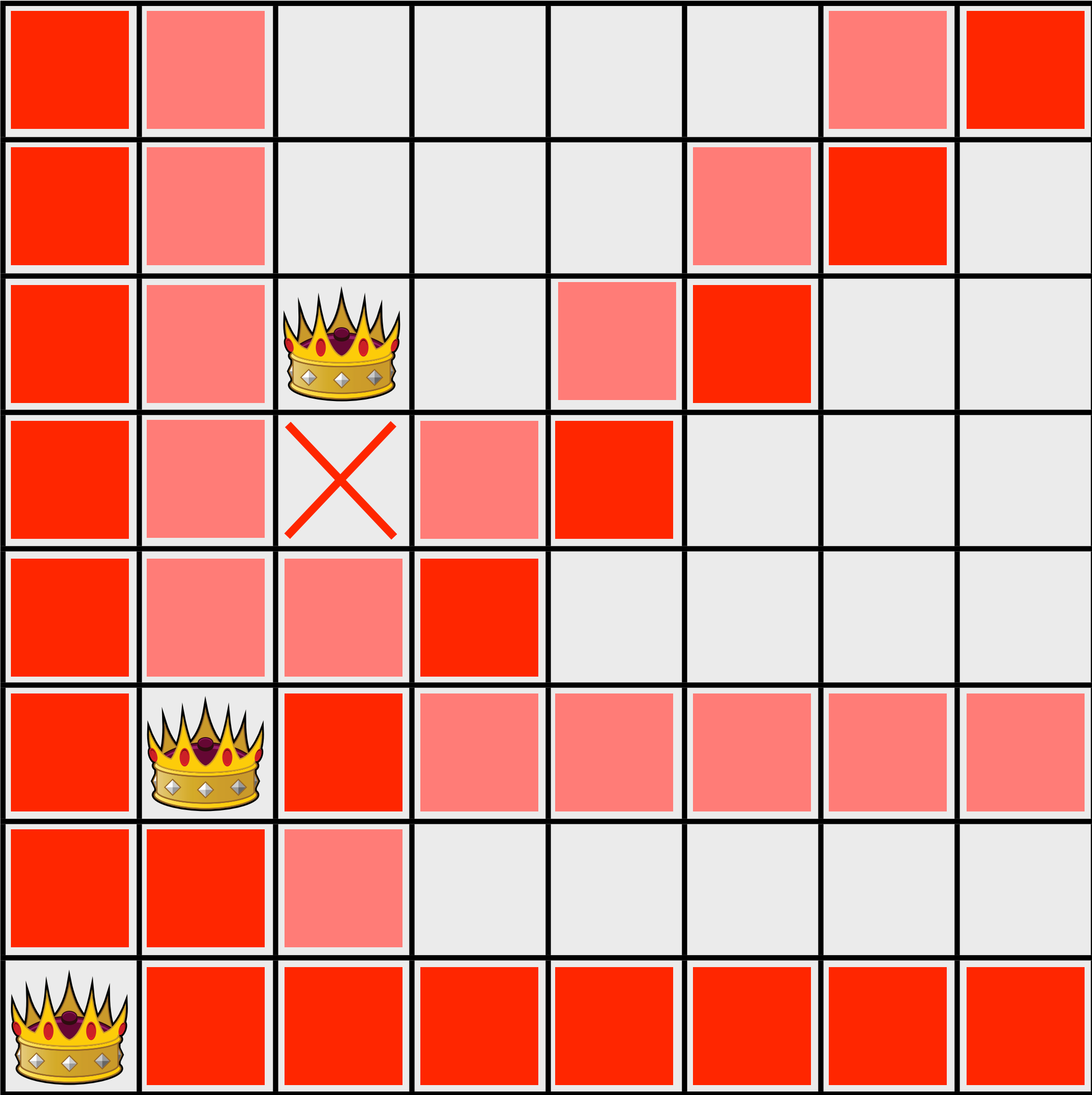
Failure!



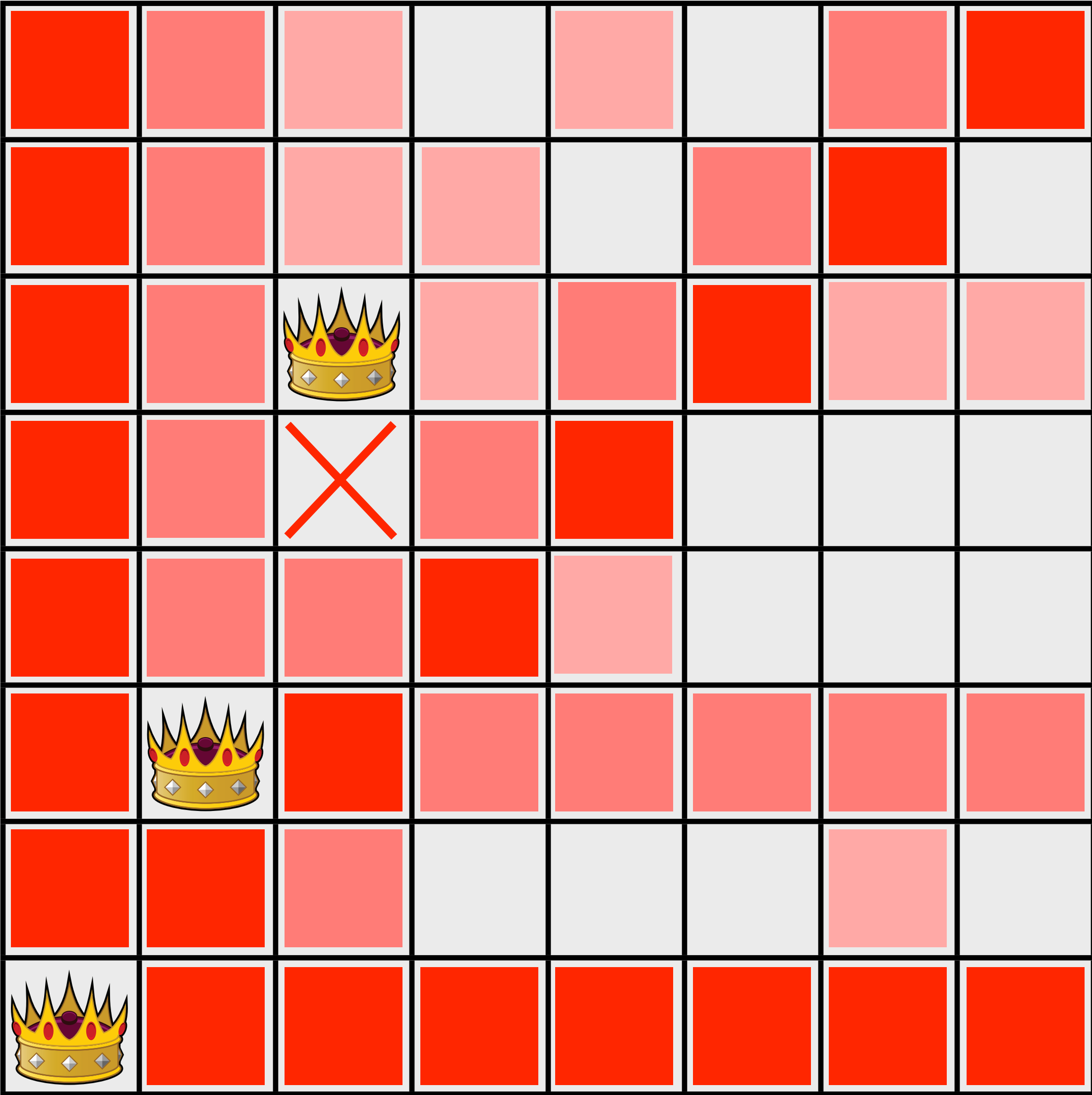
The 8-Queens Problem



The 8-Queens Problem



The 8-Queens Problem



Constraint Programming

► Computational paradigm

- use constraints to reduce the set of values that each variable can take
- make a choice when no more deduction can be performed

Constraint Programming

► Computational paradigm

- use constraints to reduce the set of values that each variable can take
- make a choice when no more deduction can be performed

► What is a choice?

- there are many choices!
- for the moment, assume a choice assigns a value to a variable

Constraint Programming

► Computational paradigm

- use constraints to reduce the set of values that each variable can take
- make a choice when no more deduction can be performed

► What is a choice?

- there are many choices!
- for the moment, assume a choice assigns a value to a variable

► Choices can be wrong

- in optimization, they are often wrong :-)
- the solver then backtracks, i.e., it tries another value

Coloring a Map

- ▶ How to color this map with constraint programming?
 - choose the decision variables
 - express the constraints in terms of the decision variables

Coloring a Map

- ▶ How to color this map with constraint programming?
 - choose the decision variables
 - express the constraints in terms of the decision variables
- ▶ What are the decision variables?
 - the color given to each country

Coloring a Map

- ▶ How to color this map with constraint programming?
 - choose the decision variables
 - express the constraints in terms of the decision variables
- ▶ What are the decision variables?
 - the color given to each country
- ▶ What are the domains of the decision variables?
 - the domain is the set of values that a variable can take
 - four different colors

Coloring a Map

- ▶ How to color this map with constraint programming?
 - choose the decision variables
 - express the constraints in terms of the decision variables
- ▶ What are the decision variables?
 - the color given to each country
- ▶ What are the domains of the decision variables?
 - the domain is the set of values that a variable can take
 - four different colors
- ▶ How do you express the constraints?
 - specify that two adjacent countries cannot be given the same color

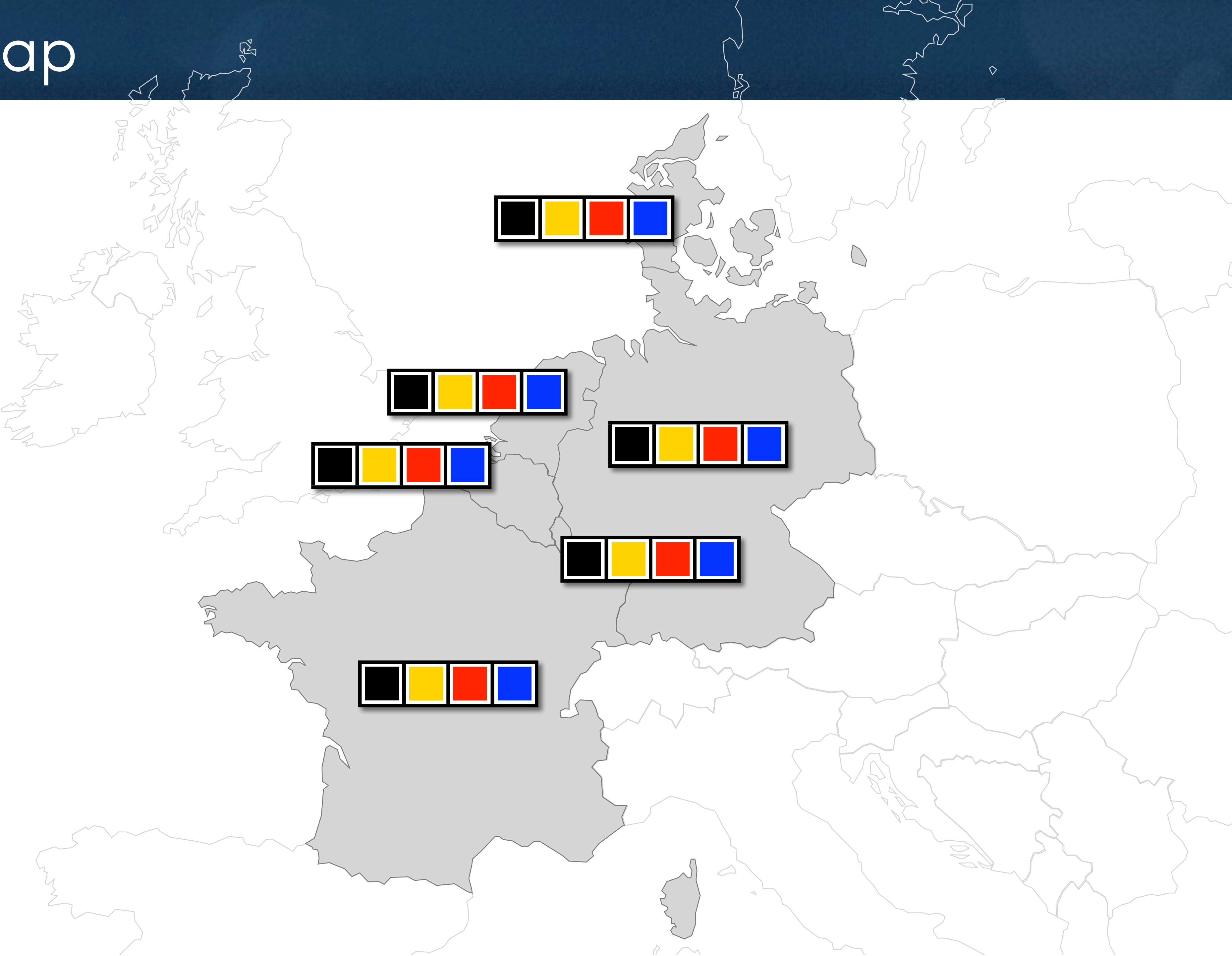
Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,  
                  Netherlands, Luxembourg };  
enum Colors = { black, yellow, red, blue };  
var{Colors} color[Countries];  
  
solve {  
    color[Belgium] ≠ color[France];  
    color[Belgium] ≠ color[Germany];  
    color[Belgium] ≠ color[Netherlands];  
    color[Belgium] ≠ color[Luxembourg];  
    color[Denmark] ≠ color[Germany];  
    color[France] ≠ color[Germany];  
    color[France] ≠ color[Luxembourg];  
    color[Germany] ≠ color[Netherlands];  
    color[Germany] ≠ color[Luxembourg];  
}
```

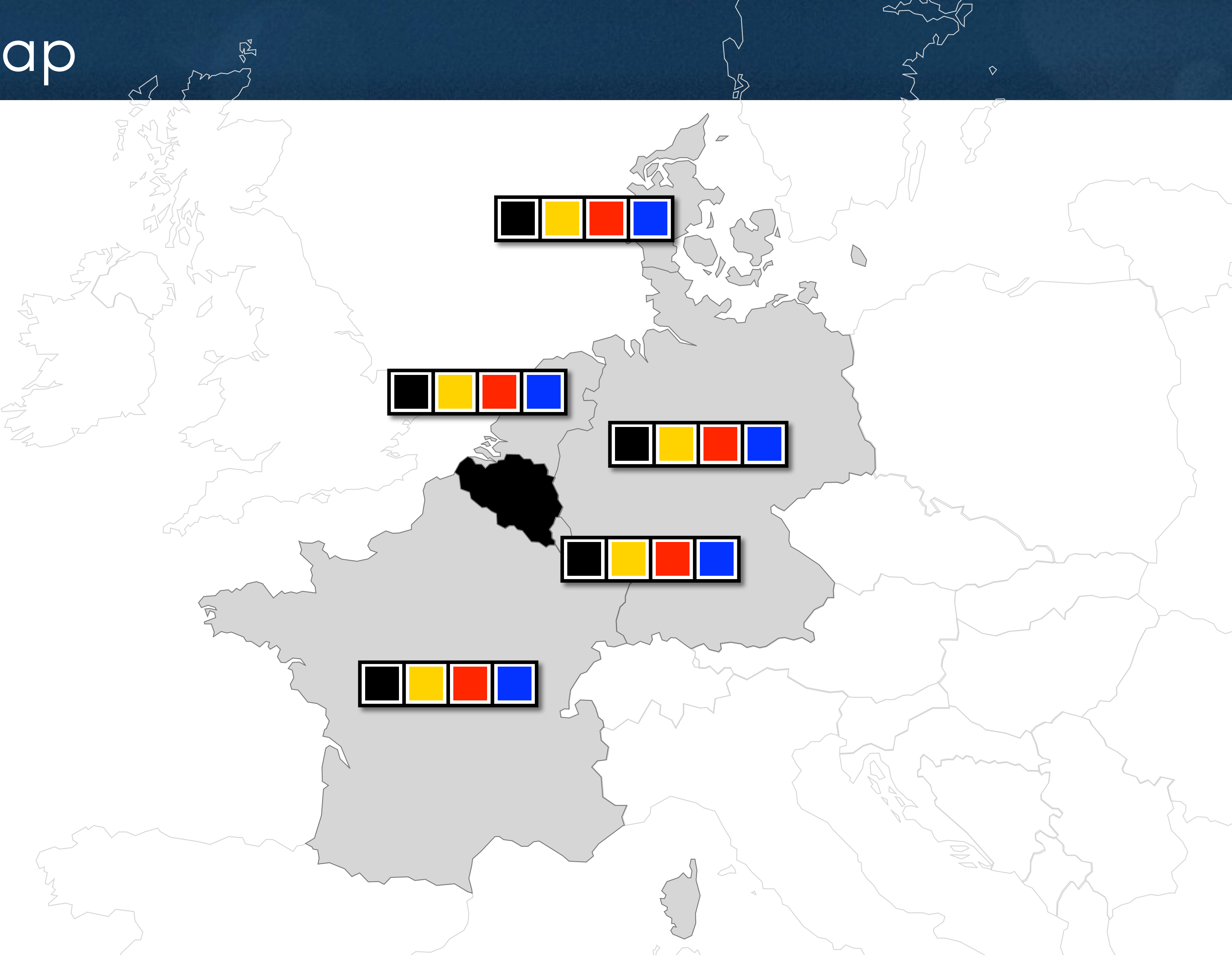
Coloring a Map



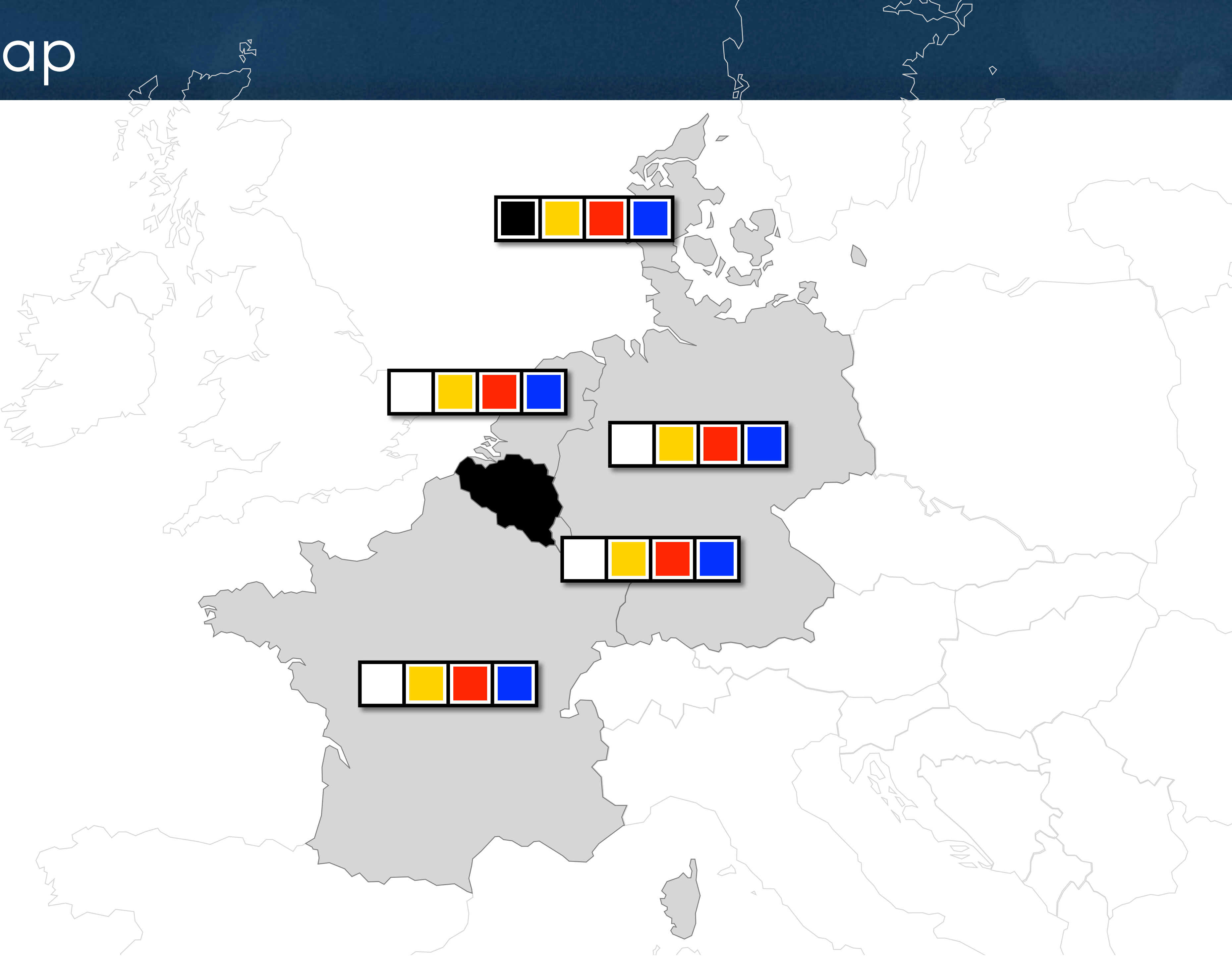
Coloring a Map



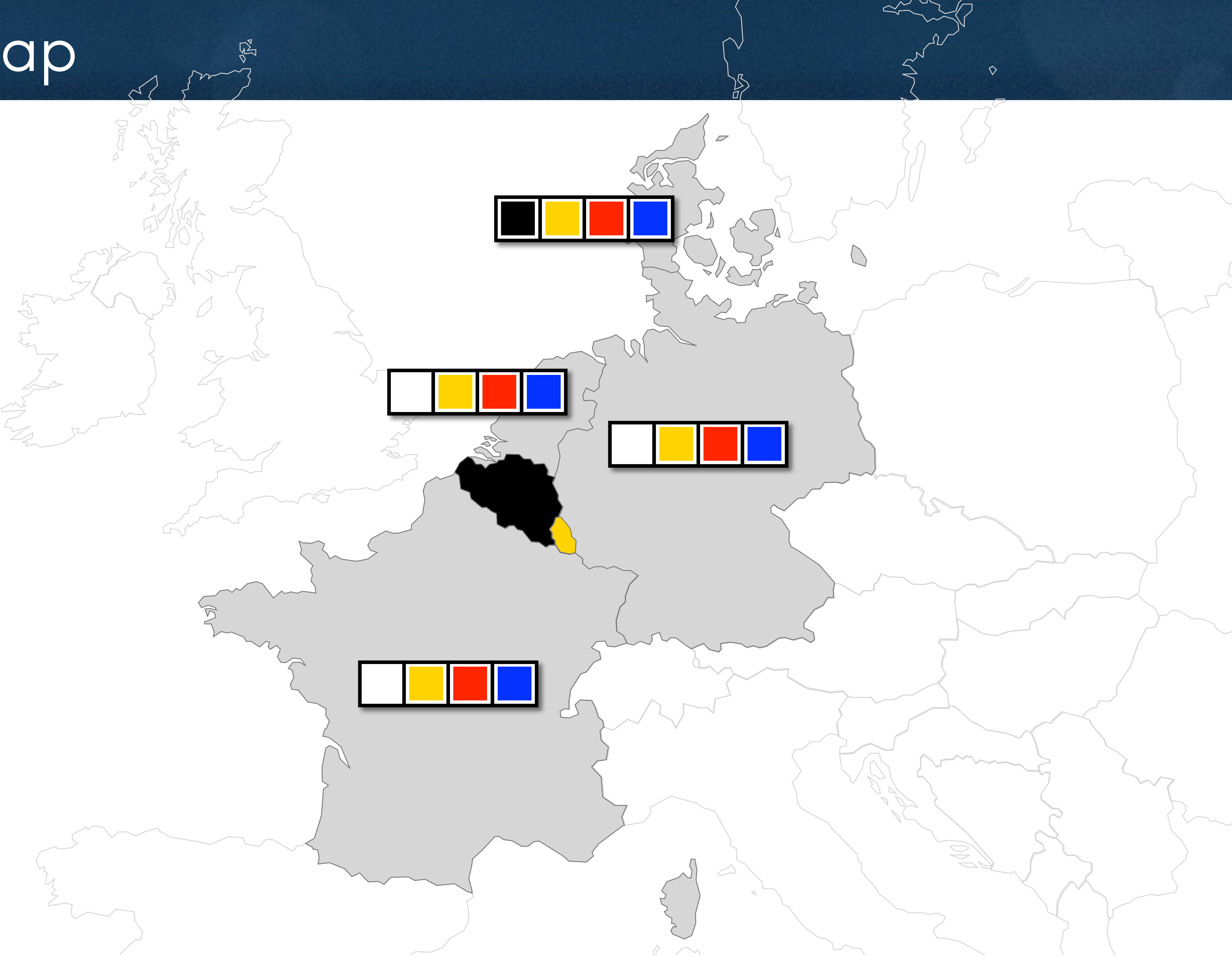
Coloring a Map



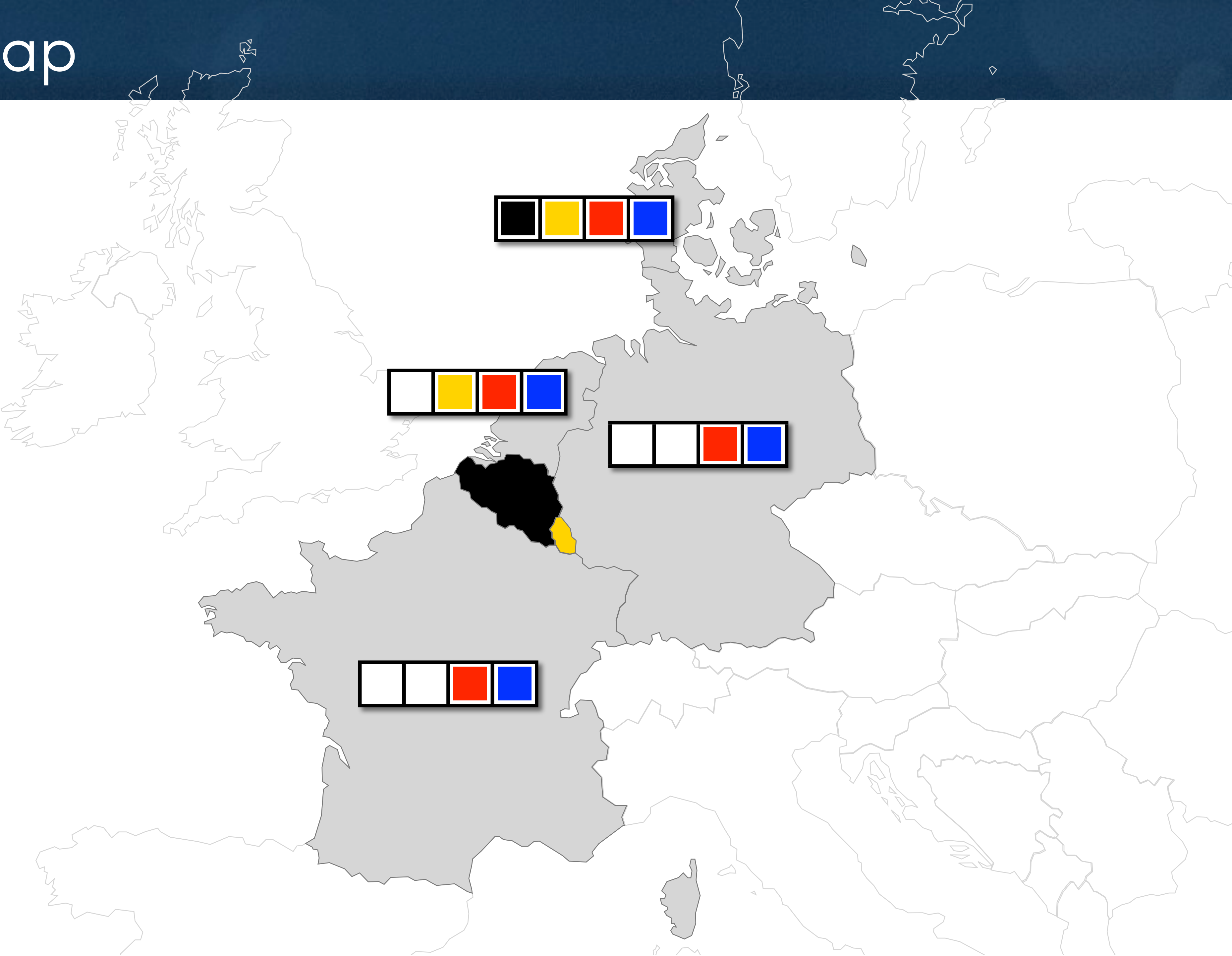
Coloring a Map



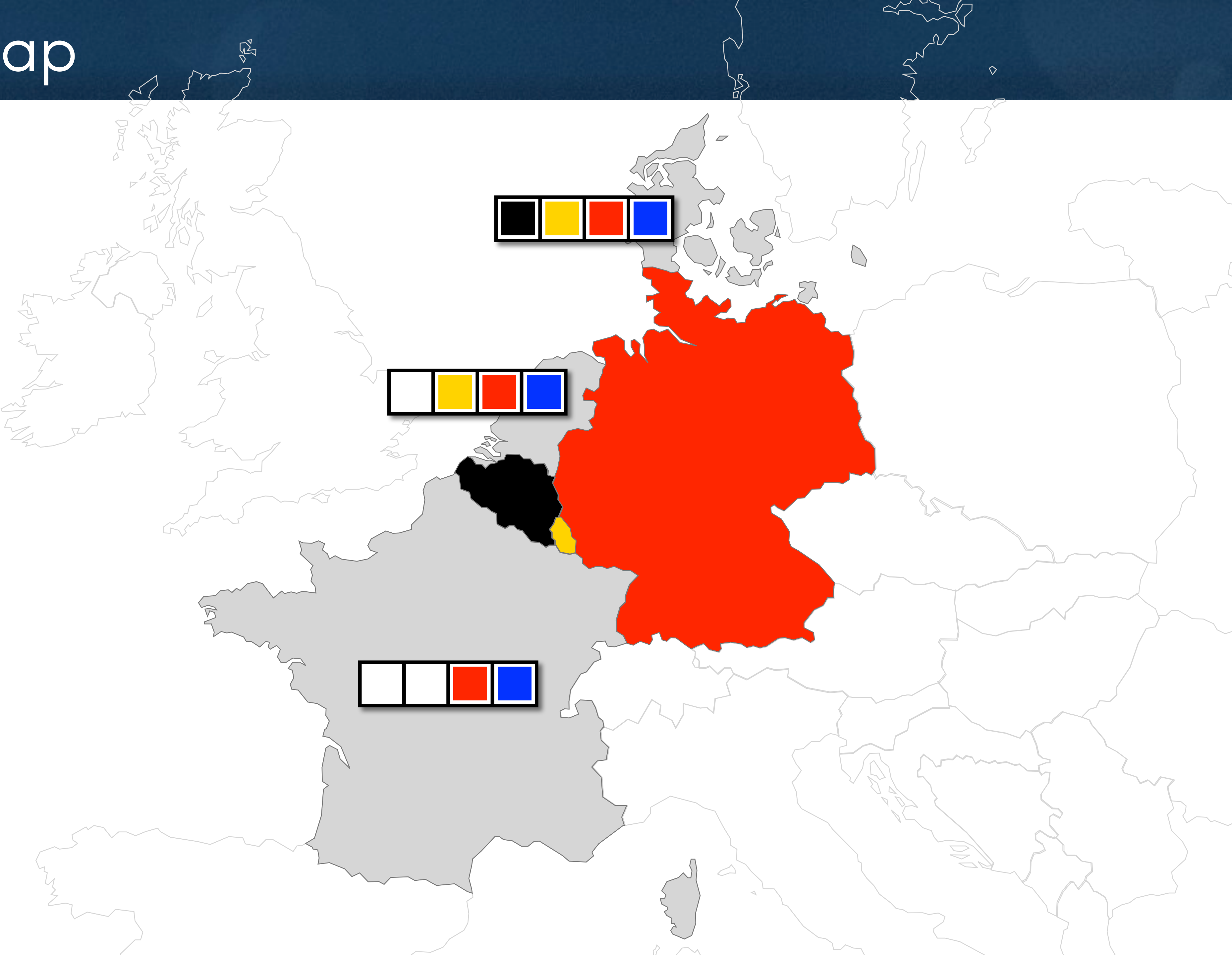
Coloring a Map



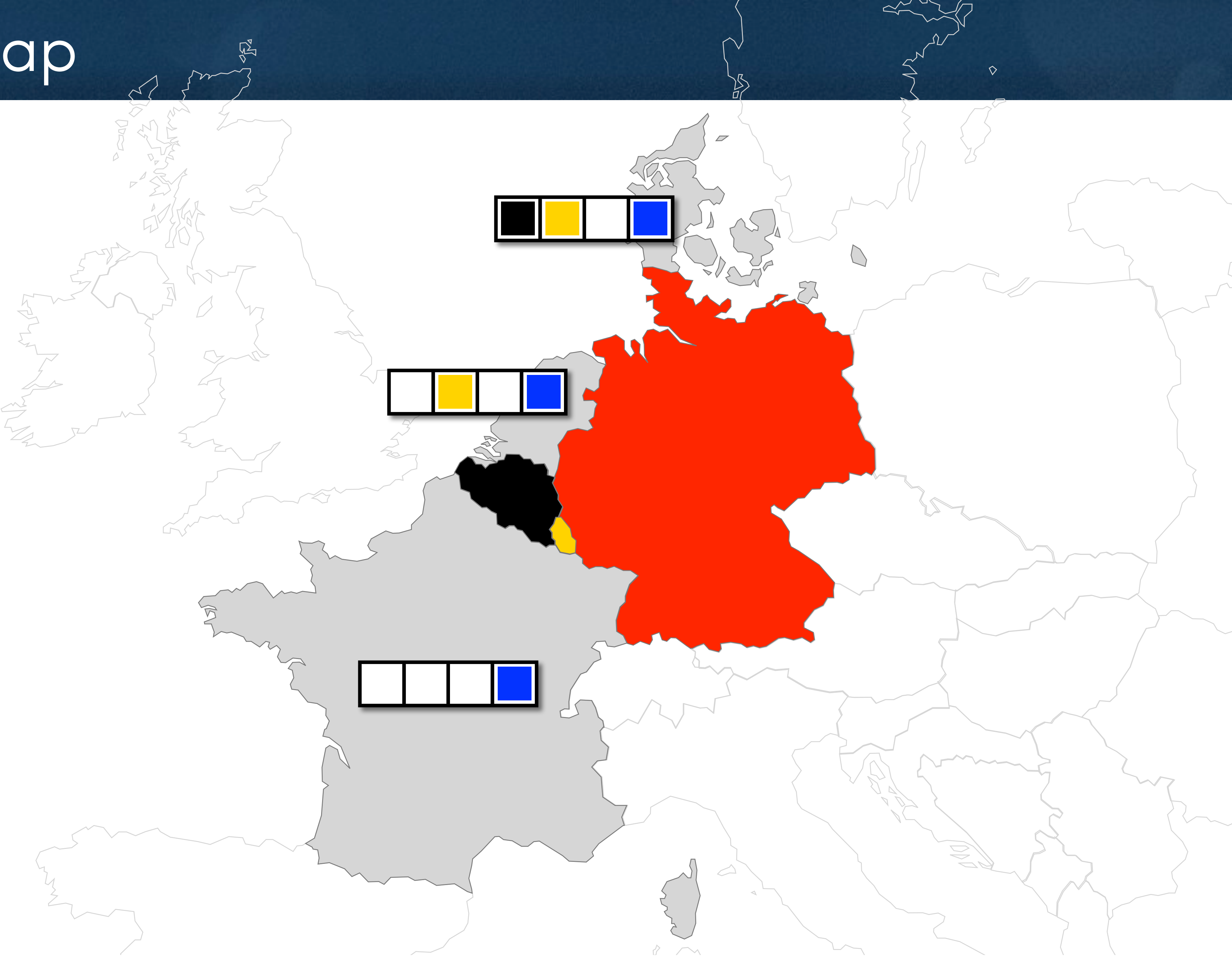
Coloring a Map



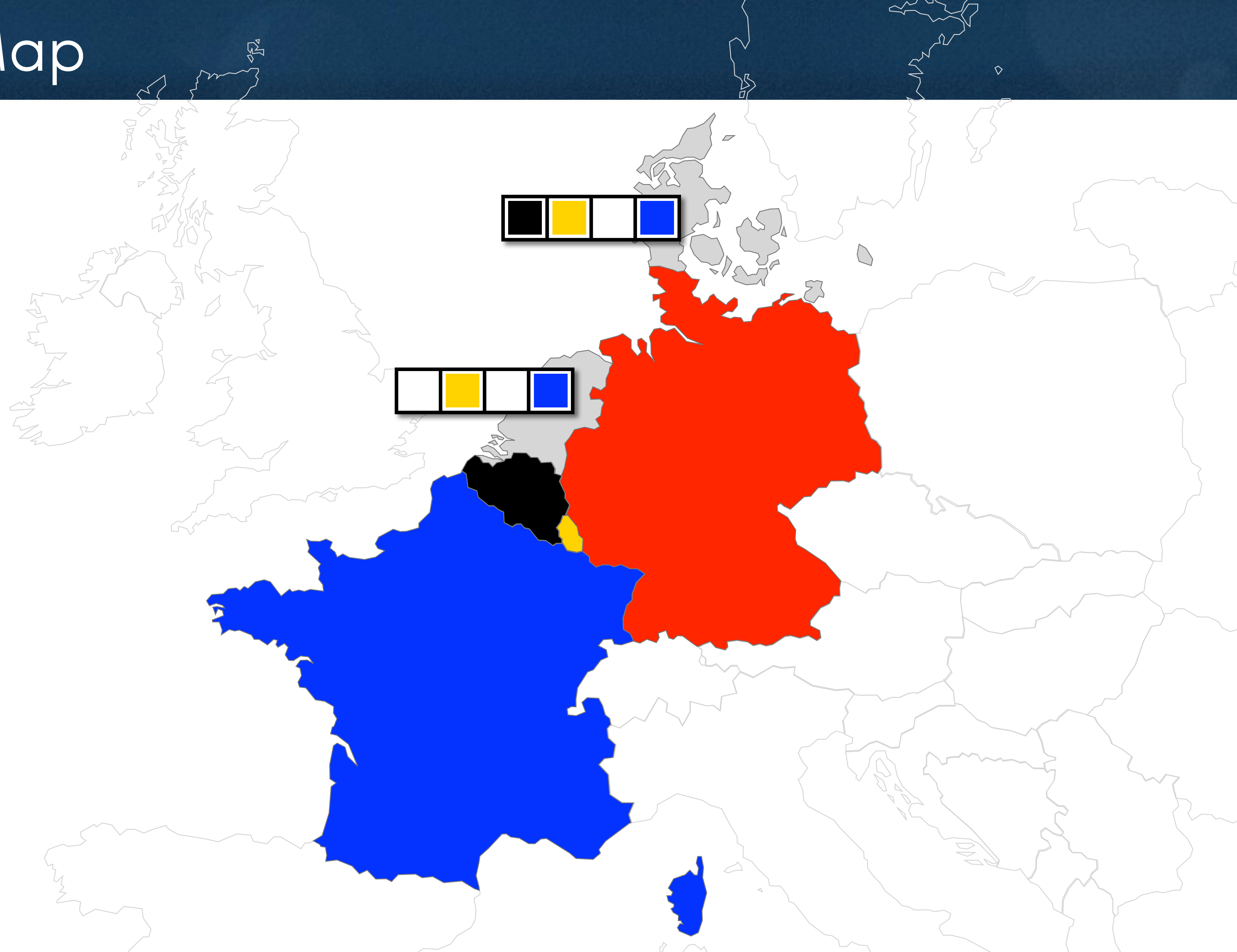
Coloring a Map



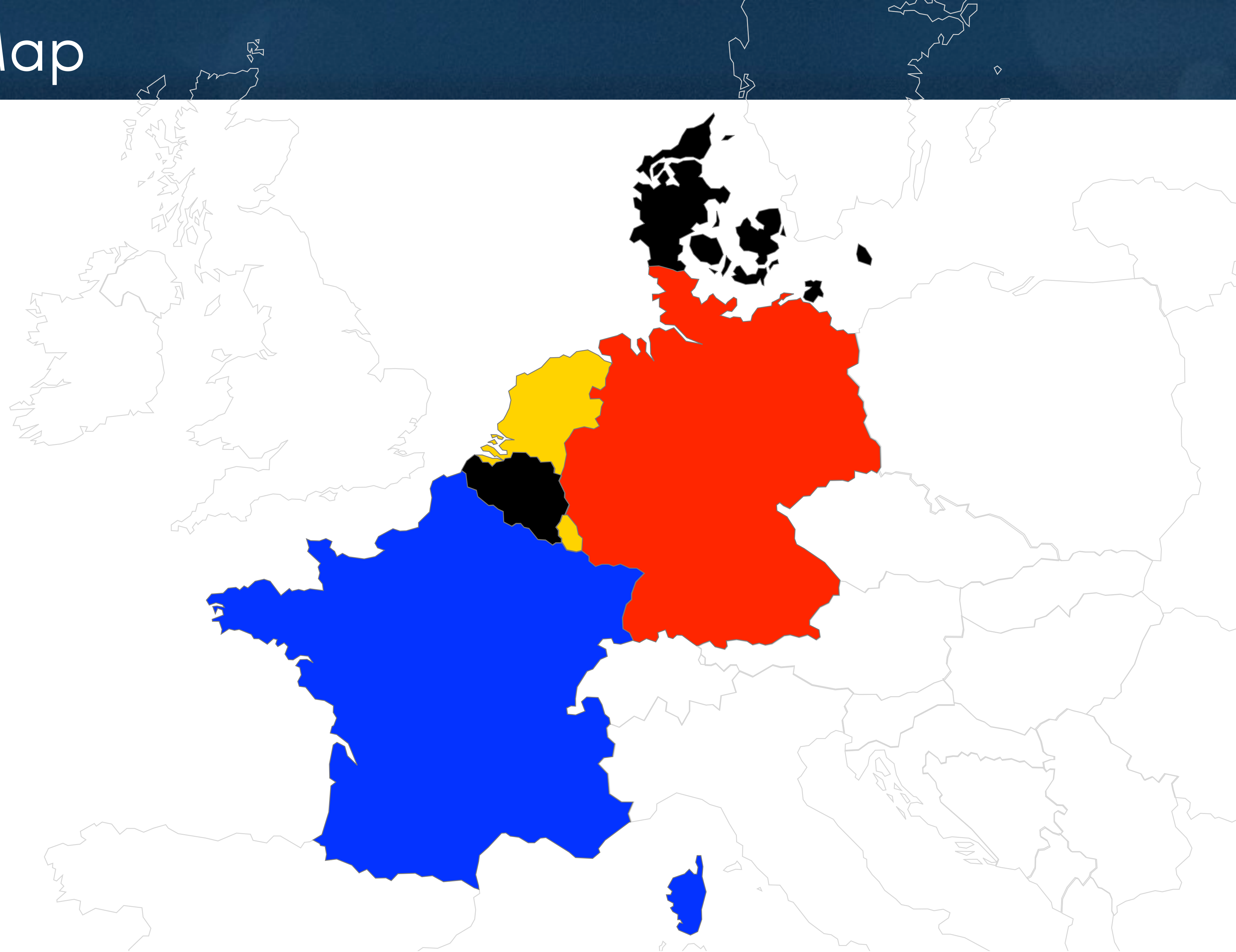
Coloring a Map



Coloring a Map



Coloring a Map



Constraint Programming

► Computational paradigm

- use constraints to reduce the set of values that each variable can take
- make a choice when no more deduction can be performed

Constraint Programming

- ▶ Computational paradigm
 - use constraints to reduce the set of values that each variable can take
 - make a choice when no more deduction can be performed
- ▶ What does this mean for the coloring problem?

Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,  
                  Netherlands, Luxembourg };  
enum Colors = { black, yellow, red, blue };  
var{Colors} color[Countries];  
  
solve {  
    color[Belgium] ≠ color[France];  
    color[Belgium] ≠ color[Germany];  
    color[Belgium] ≠ color[Netherlands];  
    color[Belgium] ≠ color[Luxembourg];  
    color[Denmark] ≠ color[Germany];  
    color[France] ≠ color[Germany];  
    color[France] ≠ color[Luxembourg];  
    color[Germany] ≠ color[Netherlands];  
    color[Germany] ≠ color[Luxembourg];  
}
```

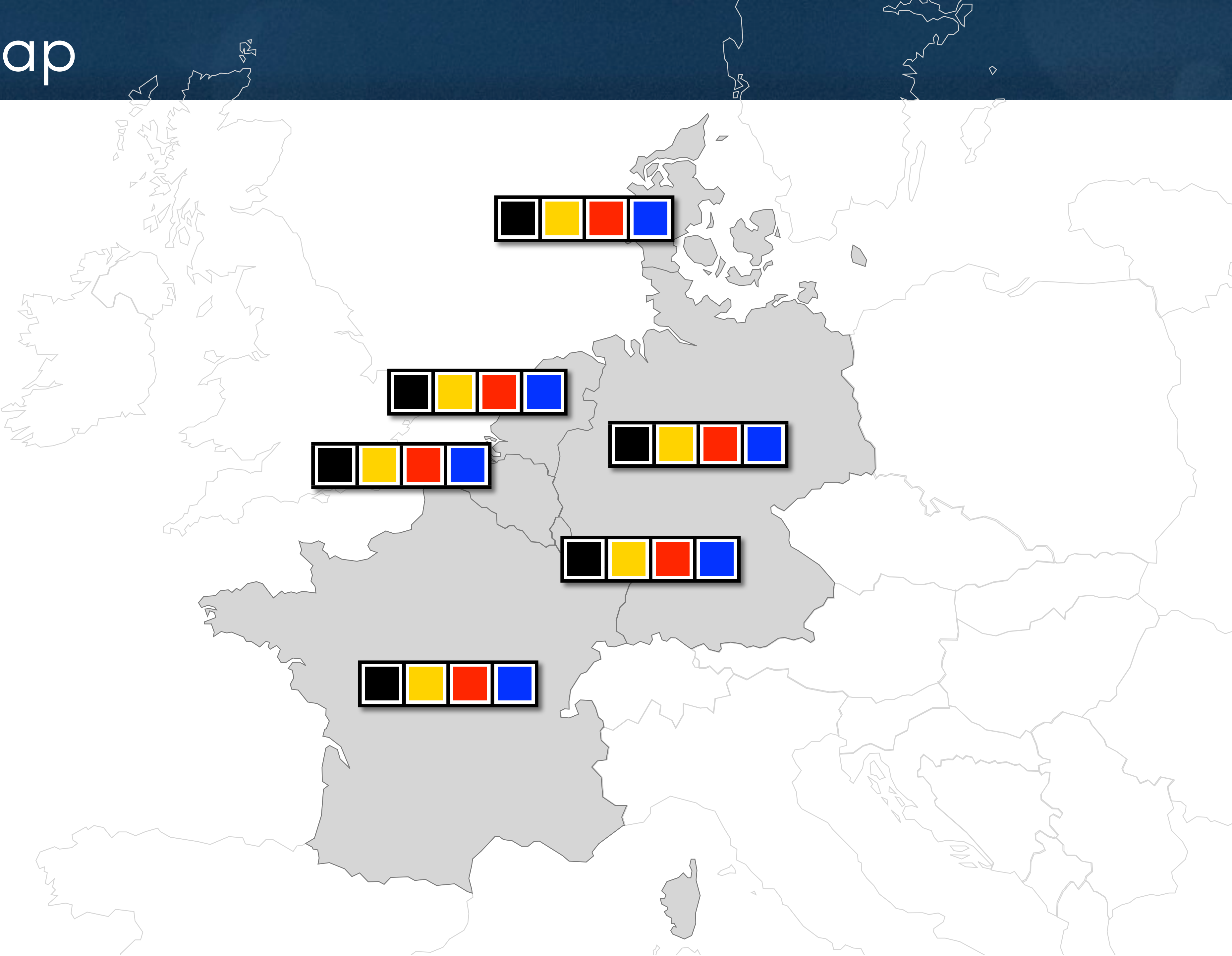

Constraint Programming

- ▶ Computational paradigm
 - use constraints to reduce the set of values that each variable can take
 - make a choice when no more deduction can be performed

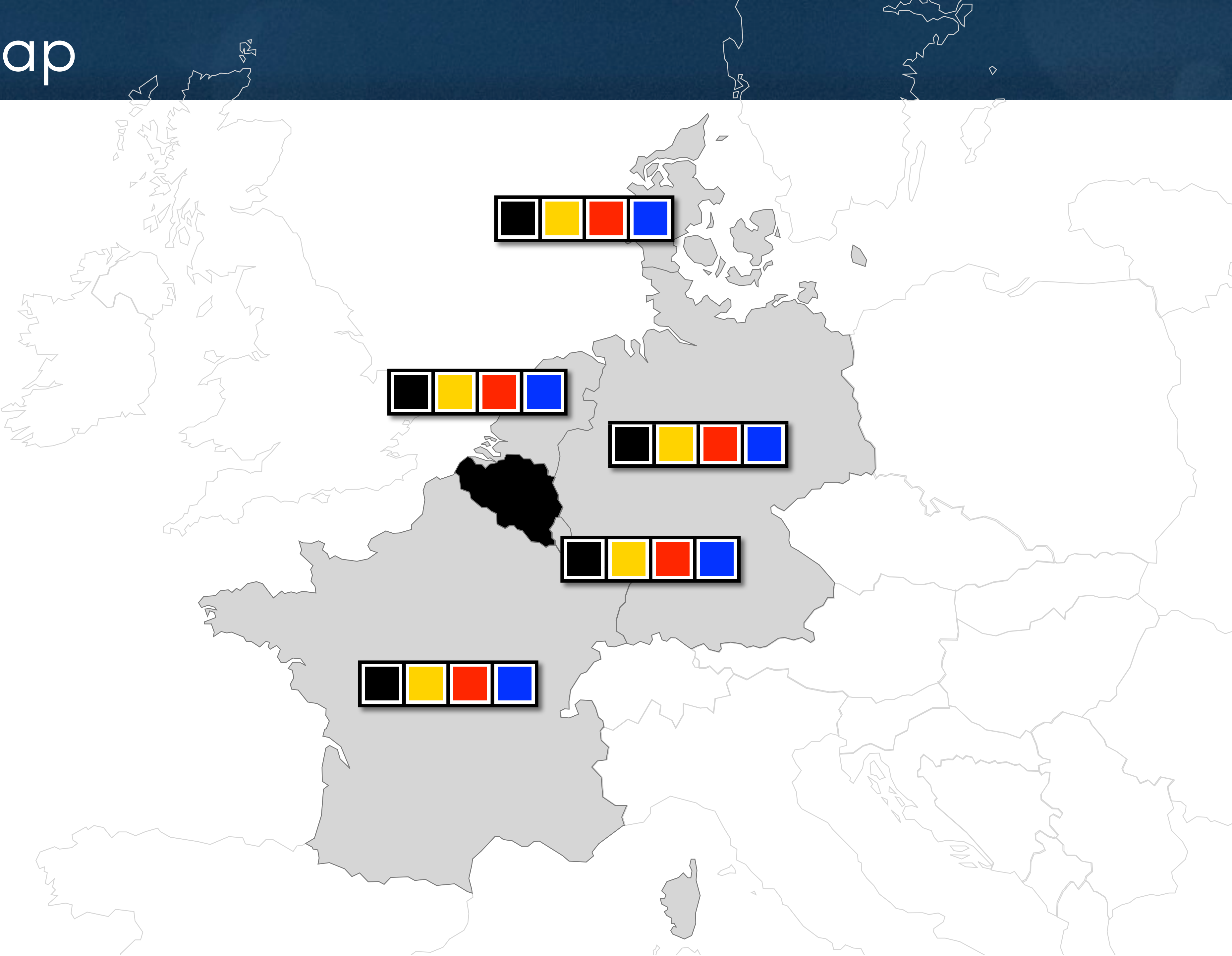
Constraint Programming

- ▶ Computational paradigm
 - use constraints to reduce the set of values that each variable can take
 - make a choice when no more deduction can be performed
- ▶ What does this mean for the coloring problem?
 - no value can be removed initially, so the system must make a choice

Coloring a Map



Coloring a Map



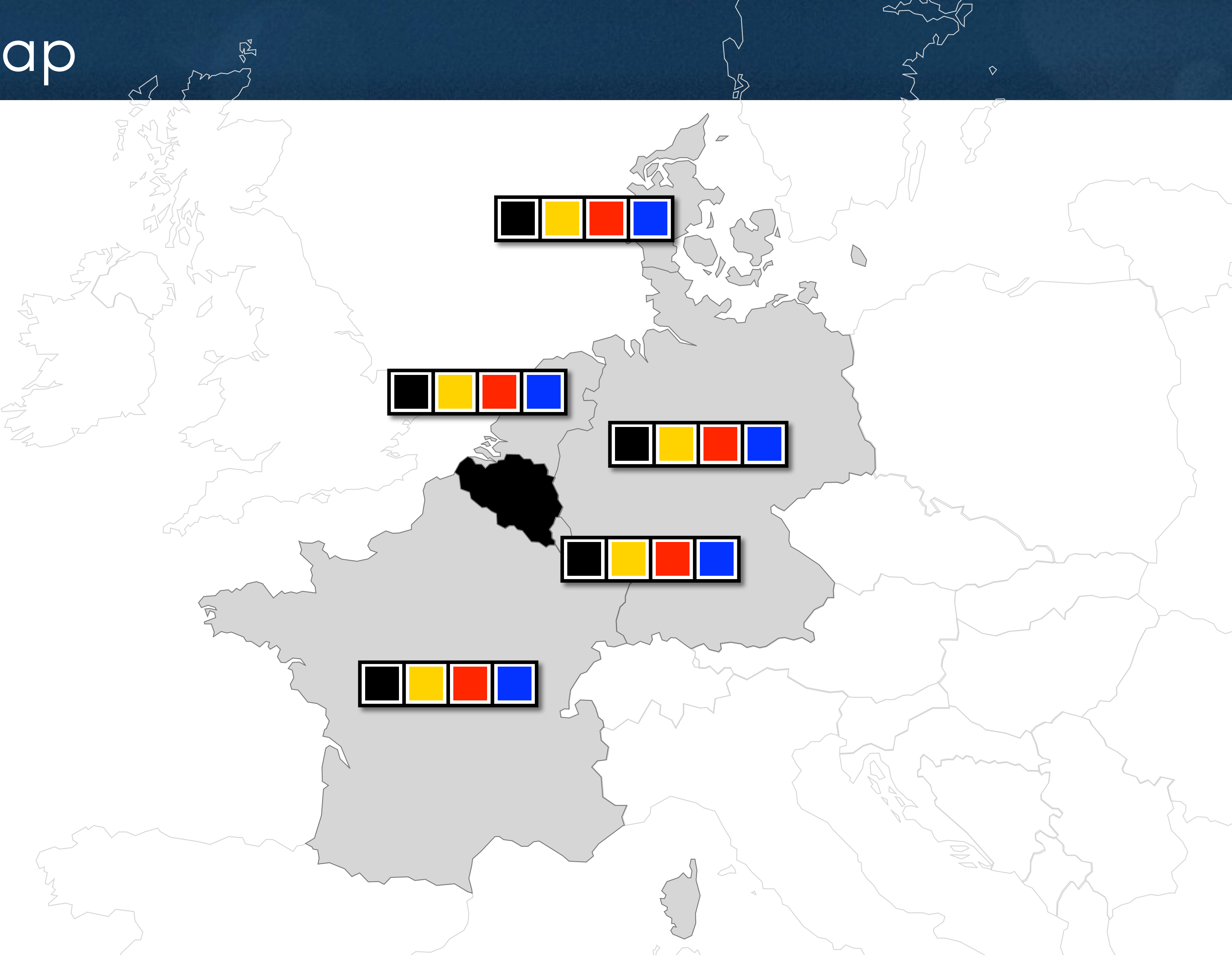
Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,  
                  Netherlands, Luxembourg };  
enum Colors = { black, yellow, red, blue };  
var{Colors} color[Countries];  
  
solve {  
  color[Belgium] ≠ color[France];  
  color[Belgium] ≠ color[Germany];  
  color[Belgium] ≠ color[Netherlands];  
  color[Belgium] ≠ color[Luxembourg];  
  color[Denmark] ≠ color[Germany];  
  color[France] ≠ color[Germany];  
  color[France] ≠ color[Luxembourg];  
  color[Germany] ≠ color[Netherlands];  
  color[Germany] ≠ color[Luxembourg];  
}
```

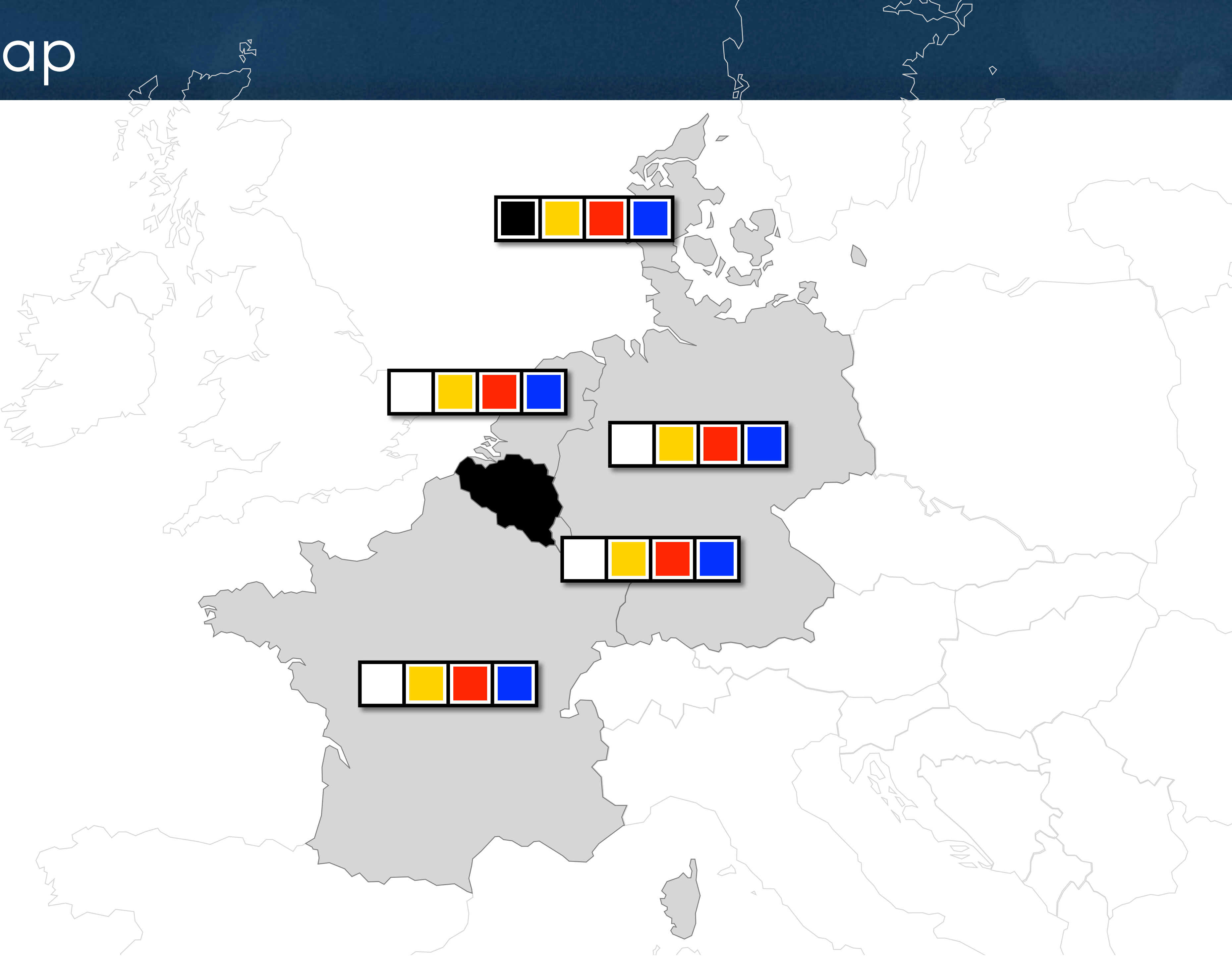
Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,  
                  Netherlands, Luxembourg };  
enum Colors = { black, yellow, red, blue };  
var{Colors} color[Countries];  
  
solve {  
    black ≠ color[France];  
    black ≠ color[Germany];  
    black ≠ color[Netherlands];  
    black ≠ color[Luxembourg];  
    color[Denmark] ≠ color[Germany];  
    color[France] ≠ color[Germany];  
    color[France] ≠ color[Luxembourg];  
    color[Germany] ≠ color[Netherlands];  
    color[Germany] ≠ color[Luxembourg];  
}
```


Coloring a Map



Coloring a Map



Computational Paradigm

► Branch and prune

– pruning

- reduce the search space as much as possible

– branching

- decompose the problem into subproblems and explore the subproblems

Computational Paradigm

► Branch and prune

– pruning

- reduce the search space as much as possible

– branching

- decompose the problem into subproblems and explore the subproblems

► Pruning

- use constraints to remove, from the variable domains, values that cannot belong to any solution

Computational Paradigm

► Branch and prune

– pruning

- reduce the search space as much as possible

– branching

- decompose the problem into subproblems and explore the subproblems

► Pruning

- use constraints to remove, from the variable domains, values that cannot belong to any solution

► Branching

- e.g., try all the possible values of a variable until a solution is found or it can be proven that no solution exists

Computational Paradigm

- ▶ Complete method, not a heuristic
 - given enough time, it will find a solution to a satisfaction problem
 - given enough time, it will find an optimal solution to an optimization problem

Computational Paradigm

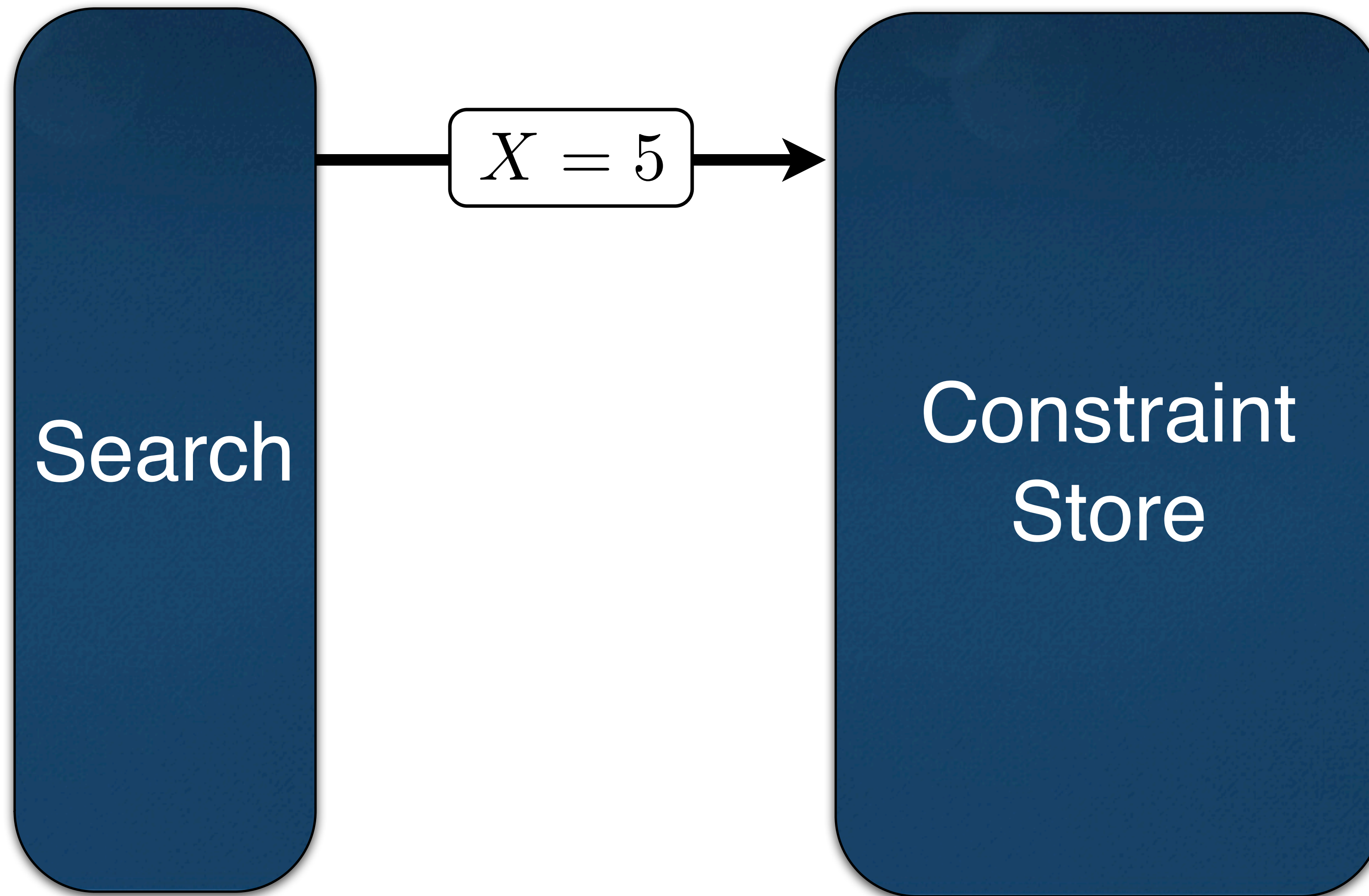
- ▶ **Complete method, not a heuristic**
 - given enough time, it will find a solution to a satisfaction problem
 - given enough time, it will find an optimal solution to an optimization problem
- ▶ **Focus on feasibility**
 - how to use constraints to prune the search space by eliminating values that cannot belong to any solution

Computational Paradigm

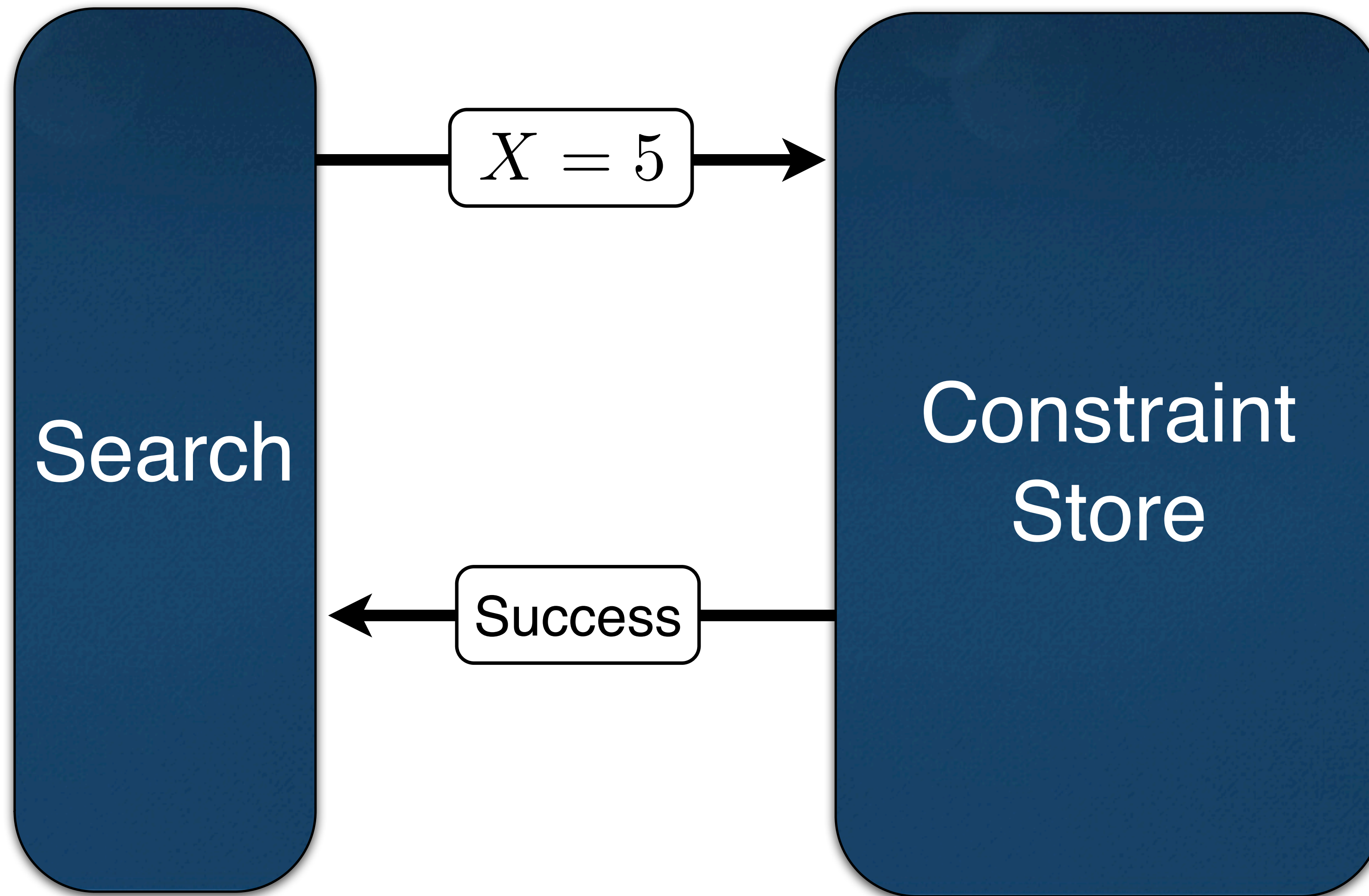
Search

Constraint
Store

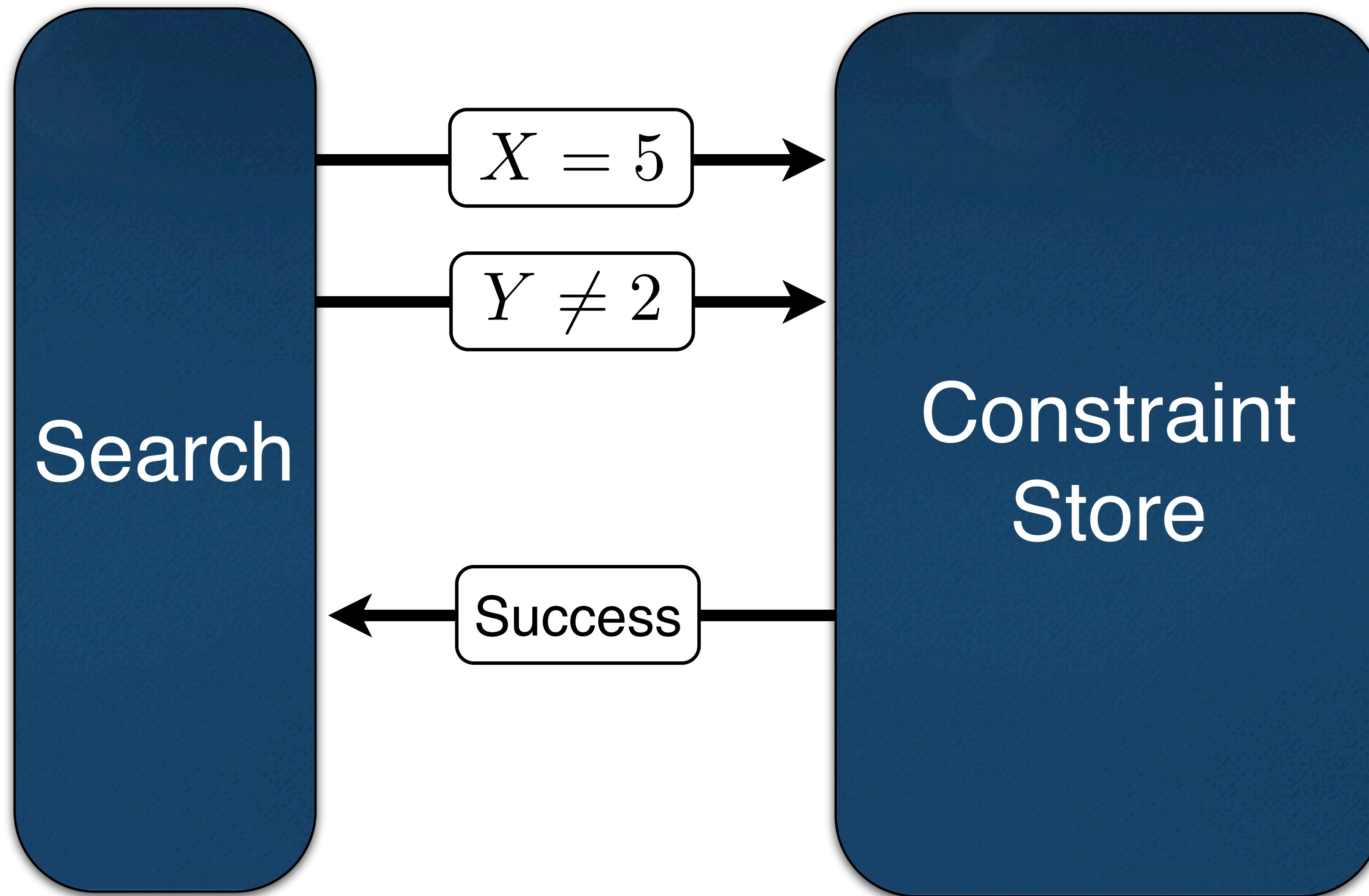
Computational Paradigm



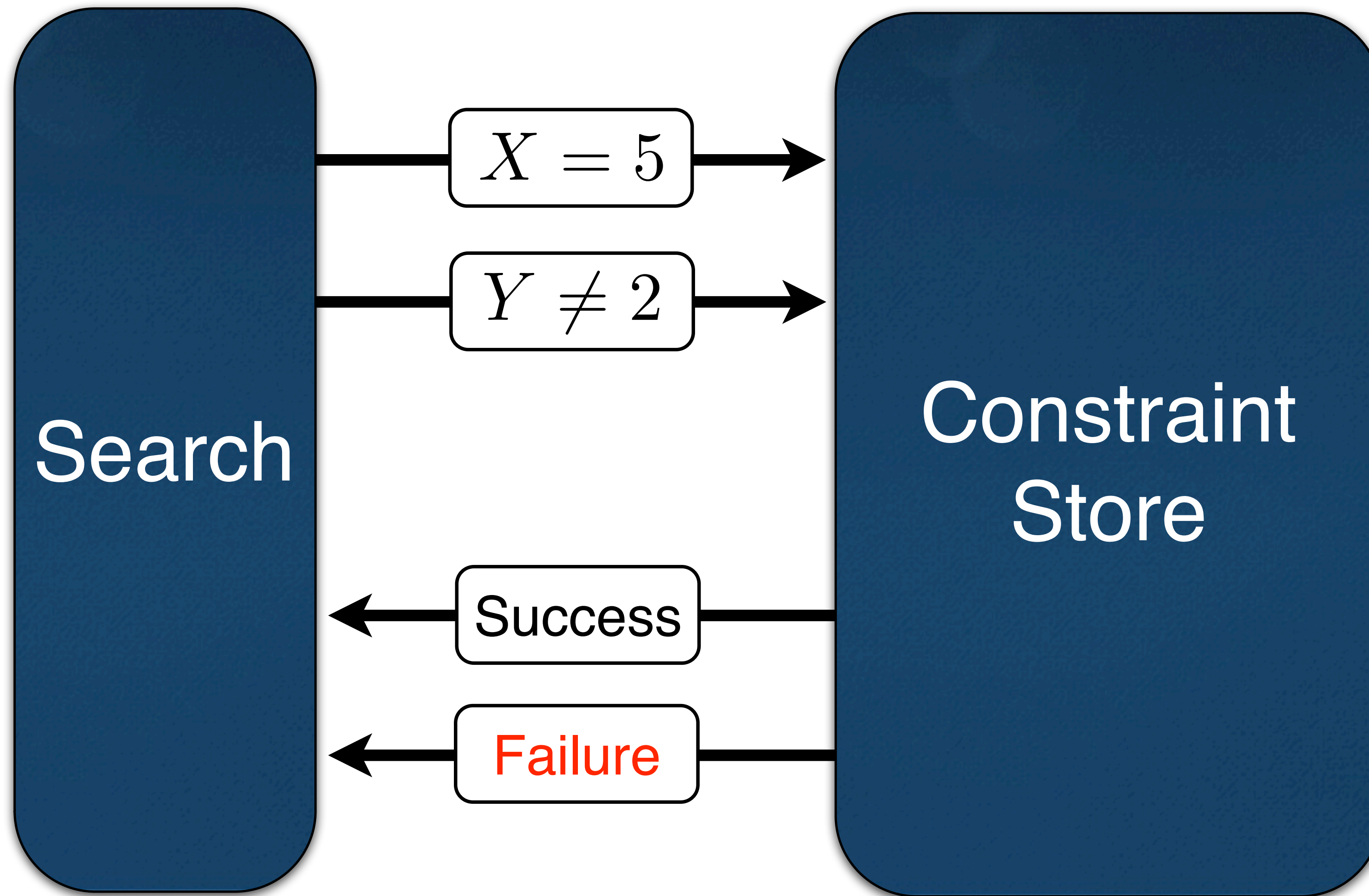
Computational Paradigm



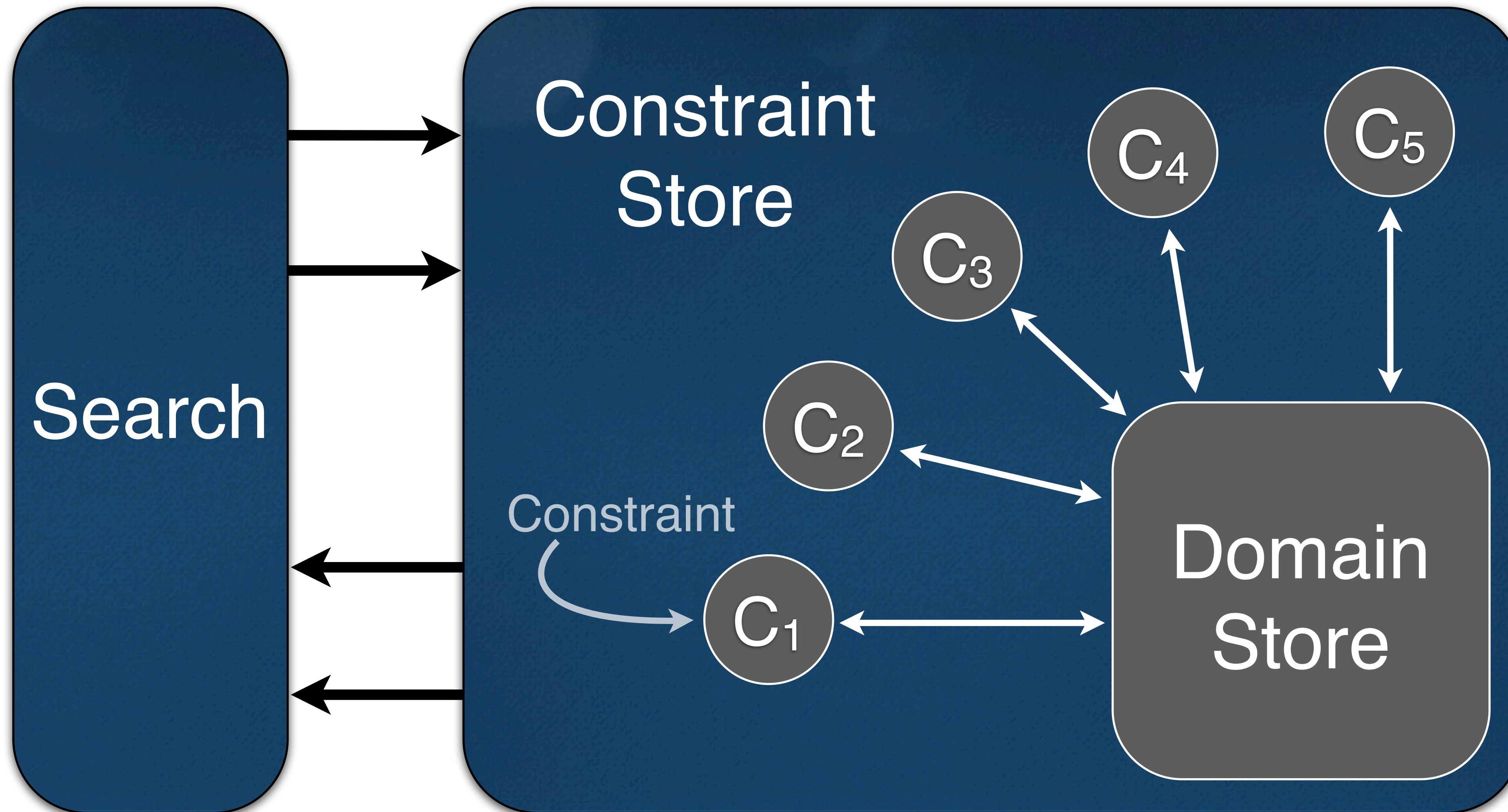
Computational Paradigm



Computational Paradigm



Computational Paradigm



Computational Paradigm

- ▶ What does a constraint do?
 - feasibility checking
 - pruning

Computational Paradigm

- ▶ What does a constraint do?
 - feasibility checking
 - pruning
- ▶ Feasibility checking
 - a constraint checks if it can be satisfied given the values in the domains of its variables

Computational Paradigm

- ▶ What does a constraint do?
 - feasibility checking
 - pruning
- ▶ Feasibility checking
 - a constraint checks if it can be satisfied given the values in the domains of its variables
- ▶ Pruning
 - if satisfiable, a constraint determines which values in the domains cannot be part of any solution

Computational Paradigm

- ▶ The propagation engine
 - this is the core of any constraint-programming solver
 - a simple (fixpoint) algorithm

```
propagate()  
{  
  repeat  
    select a constraint c;  
    if c is infeasible given the domain store then  
      return failure;  
    else  
      apply the pruning algorithm associated with c;  
  until no constraint can remove any value from the  
  domain of its variables;  
  return success;  
}
```

Back to the 8-Queens Problem

- ▶ What are the decision variables?
 - many possible modelings
 - this is what makes optimization problems interesting :-)

Back to the 8-Queens Problem

- ▶ What are the decision variables?
 - many possible modelings
 - this is what makes optimization problems interesting :-)
- ▶ Here is one modeling
 - associate a decision variable with each column
 - the variable denotes the row of the queens placed in this column
 - no two queens can be placed on the same column so this is valid

Back to the 8-Queens Problem

- ▶ What are the decision variables?
 - many possible modelings
 - this is what makes optimization problems interesting :-)
- ▶ Here is one modeling
 - associate a decision variable with each column
 - the variable denotes the row of the queens placed in this column
 - no two queens can be placed on the same column so this is valid
- ▶ What are the constraints?
 - the queens cannot be placed on the same,
 - row
 - upward diagonal
 - downward diagonal

A Constraint Program for the 8-Queens

► Constraints

- the queens cannot be placed on the same,
 - row
 - upward diagonal
 - downward diagonal

A Constraint Program for the 8-Queens

► Constraints

– the queens cannot be placed on the same,

- row
- upward diagonal
- downward diagonal

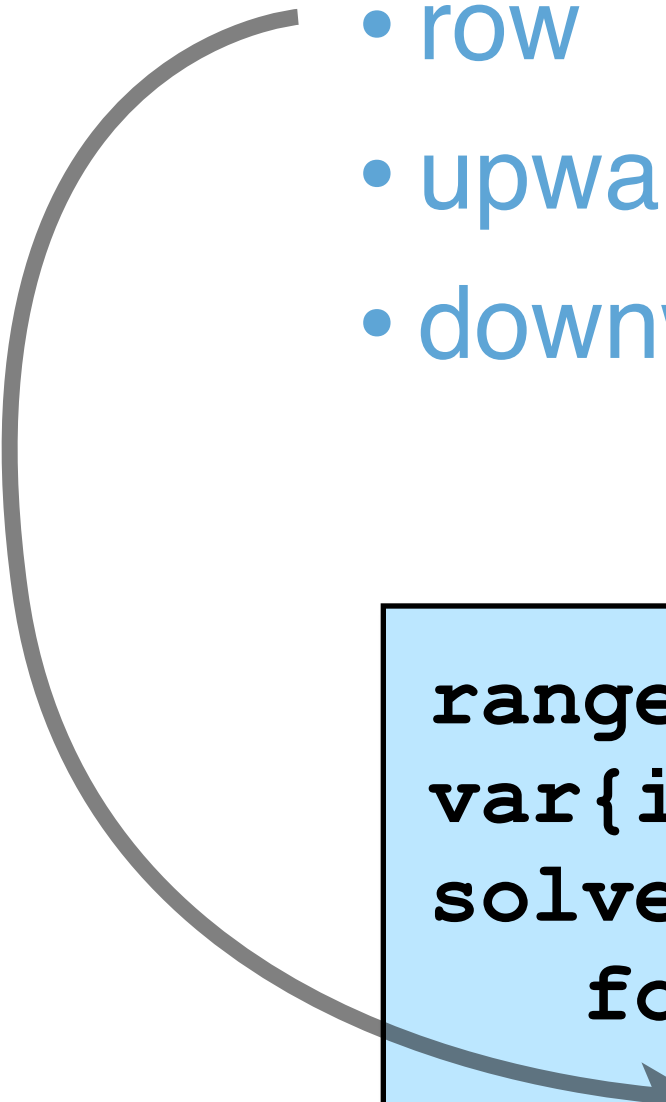
```
range R = 1..8;
var{int} row[R] in R;
solve {
    forall(i in R, j in R: i < j) {
        row[i] ≠ row[j];
        row[i] ≠ row[j] + (j - i);
        row[i] ≠ row[j] - (j - i);
    }
}
```

A Constraint Program for the 8-Queens

► Constraints

– the queens cannot be placed on the same,

- row
- upward diagonal
- downward diagonal



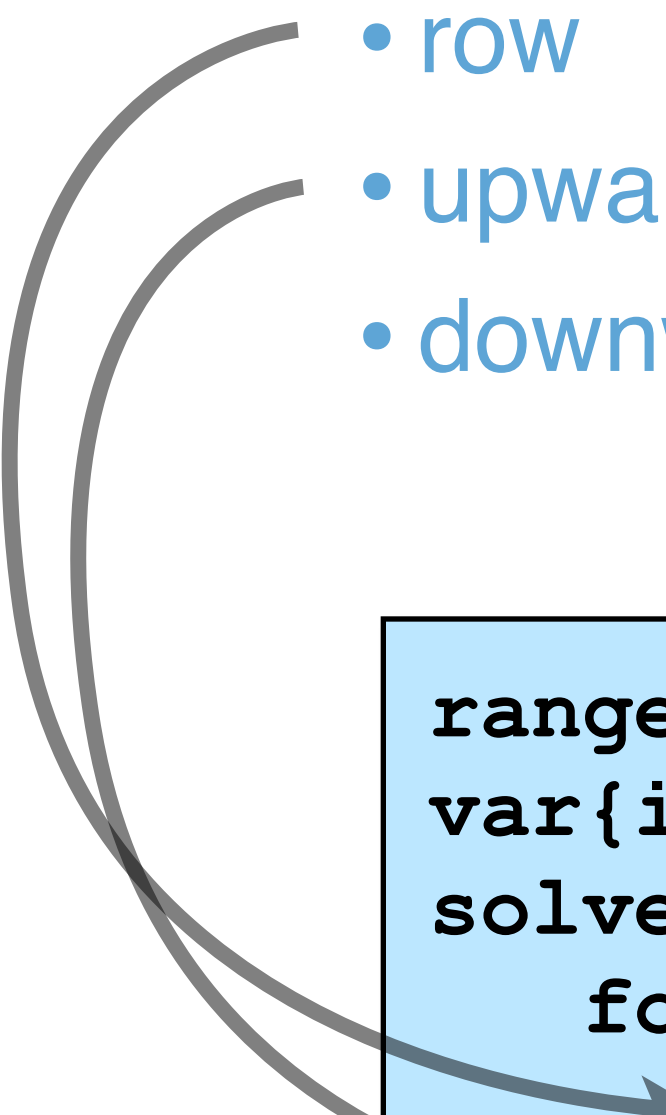
```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
  forall(i in R, j in R: i < j) {  
    row[i] != row[j];  
    row[i] != row[j] + (j - i);  
    row[i] != row[j] - (j - i);  
  }  
}
```

A Constraint Program for the 8-Queens

► Constraints

– the queens cannot be placed on the same,

- row
- upward diagonal
- downward diagonal



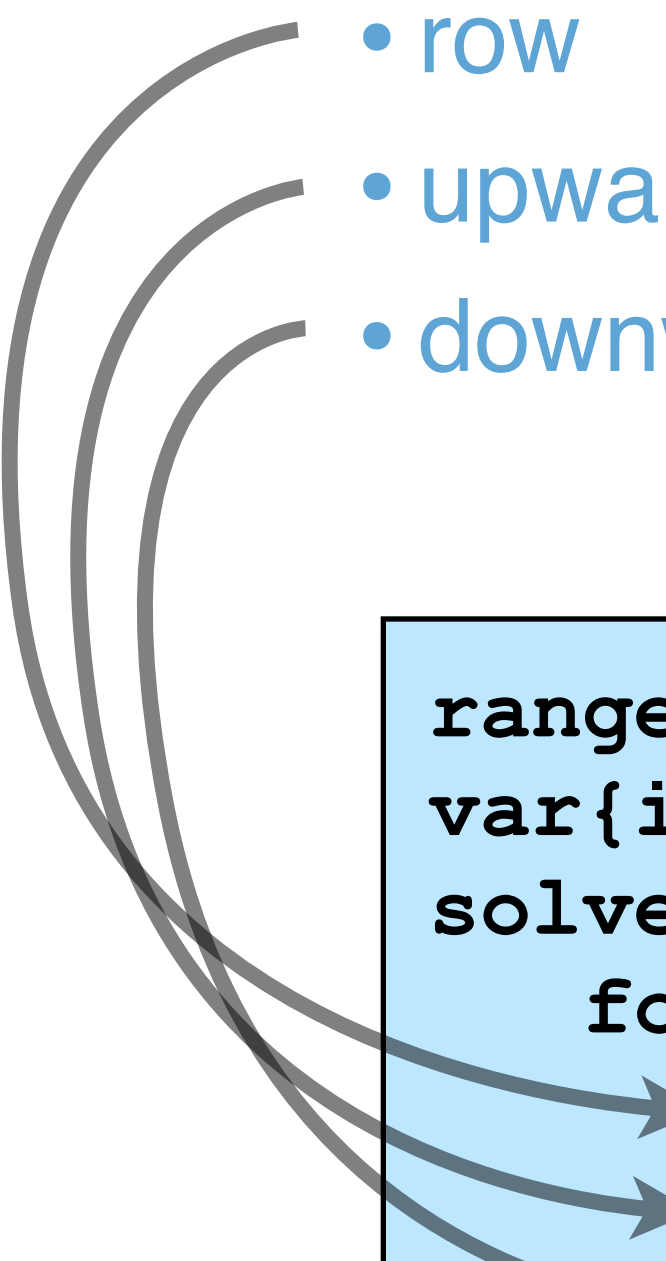
```
range R = 1..8;
var{int} row[R] in R;
solve {
  forall(i in R, j in R: i < j) {
    → row[i] ≠ row[j];
    → row[i] ≠ row[j] + (j - i);
    row[i] ≠ row[j] - (j - i);
  }
}
```


A Constraint Program for the 8-Queens

► Constraints

– the queens cannot be placed on the same,

- row
- upward diagonal
- downward diagonal



```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        → row[i] ≠ row[j];  
        → row[i] ≠ row[j] + (j - i);  
        → row[i] ≠ row[j] - (j - i);  
    }  
}
```

A Constraint Program for the 8-Queens

- A simple model for the 8-queens problem

```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        row[i] ≠ row[j];  
        row[i] ≠ row[j] + (j - i);  
        row[i] ≠ row[j] - (j - i);  
    }  
}
```

A Constraint Program for the 8-Queens

- ▶ A simple model for the 8-queens problem
- ▶ What happens when the queen in column 1 is assigned the value 1

```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        row[i] ≠ row[j];  
        row[i] ≠ row[j] + (j - i);  
        row[i] ≠ row[j] - (j - i);  
    }  
}
```

A Constraint Program for the 8-Queens

- ▶ A simple model for the 8-queens problem
- ▶ What happens when the queen in column 1 is assigned the value 1

```
row[1] ≠ row[2];  
...  
row[1] ≠ row[8];  
  
row[1] ≠ row[2] + 1;  
...  
row[1] ≠ row[8] + 7;  
  
row[1] ≠ row[2] - 1;  
...  
row[1] ≠ row[8] - 7;
```

```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        row[i] ≠ row[j];  
        row[i] ≠ row[j] + (j - i);  
        row[i] ≠ row[j] - (j - i);  
    }  
}
```


A Constraint Program for the 8-Queens

- ▶ A simple model for the 8-queens problem
- ▶ What happens when the queen in column 1 is assigned the value 1

```
row[1] ≠ row[2];  
...  
row[1] ≠ row[8];  
  
row[1] ≠ row[2] + 1;  
...  
row[1] ≠ row[8] + 7;  
  
row[1] ≠ row[2] - 1;  
...  
row[1] ≠ row[8] - 7;
```

```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        row[i] ≠ row[j];  
        row[i] ≠ row[j] + (j - i);  
        row[i] ≠ row[j] - (j - i);  
    }  
}
```

A Constraint Program for the 8-Queens

- ▶ A simple model for the 8-queens problem
- ▶ What happens when the queen in column 1 is assigned the value 1

```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        row[i] ≠ row[j];  
        row[i] ≠ row[j] + (j - i);  
        row[i] ≠ row[j] - (j - i);  
    }  
}
```

```
row[1] ≠ row[2];  
...  
row[1] ≠ row[8];  
  
row[1] ≠ row[2] + 1;  
...  
row[1] ≠ row[8] + 7;  
  
row[1] ≠ row[2] - 1;  
...  
row[1] ≠ row[8] - 7;
```

A Constraint Program for the 8-Queens

- ▶ A simple model for the 8-queens problem
- ▶ What happens when the queen in column 1 is assigned the value 1

```
range R = 1..8;  
var{int} row[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        row[i] ≠ row[j];  
        row[i] ≠ row[j] + (j - i);  
        row[i] ≠ row[j] - (j - i);  
    }  
}
```

```
row[1] ≠ row[2];  
...  
row[1] ≠ row[8];  
  
row[1] ≠ row[2] + 1;  
...  
row[1] ≠ row[8] + 7;  
  
row[1] ≠ row[2] - 1;  
...  
row[1] ≠ row[8] - 7;
```

Computational Paradigm

- ▶ What does a constraint do?
 - feasibility checking
 - pruning

Computational Paradigm

- ▶ What does a constraint do?
 - feasibility checking
 - pruning
- ▶ Feasibility checking
 - a constraint checks if it can be satisfied given the values in the domains of its variables

Computational Paradigm

- ▶ What does a constraint do?
 - feasibility checking
 - pruning
- ▶ Feasibility checking
 - a constraint checks if it can be satisfied given the values in the domains of its variables
- ▶ Pruning
 - if satisfiable, a constraint determines which values in the domains cannot be part of any solution

Computational Paradigm

- ▶ Consider two variables X , Y
 - X can take the values 0,1,2
 - Y can take the values 1,2,3
- ▶ The domain of X is the set of values it can take
- ▶ A short hand for ranges of integers

Computational Paradigm

- ▶ Consider two variables X, Y

- X can take the values 0,1,2

- Y can take the values 1,2,3

X

$\in \{0, 1, 2\}$

Y

$\in \{1, 2, 3\}$

- ▶ The domain of X is the set of values it can take
- ▶ A short hand for ranges of integers

Computational Paradigm

- ▶ Consider two variables X , Y
 - X can take the values 0,1,2
 - Y can take the values 1,2,3
- ▶ The domain of X is the set of values it can take

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$D(X) = \{0, 1, 2\}$$

Computational Paradigm

- ▶ Consider two variables X, Y
 - X can take the values 0,1,2
 - Y can take the values 1,2,3

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

- ▶ The domain of X is the set of values it can take

$$D(X) = \{0, 1, 2\}$$

- ▶ A short hand for ranges of integers

$$[1..5] = \{1, 2, 3, 4, 5\}$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Feasibility checking

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Feasibility checking

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$|D(X) \cup D(Y)| \geq 2$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Feasibility checking

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$|D(X) \cup D(Y)| \geq 2$$

$$|\{0, 1, 2\} \cup \{1, 2, 3\}| \geq 2$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Feasibility checking

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$|D(X) \cup D(Y)| \geq 2$$

$$|\{0, 1, 2\} \cup \{1, 2, 3\}| \geq 2$$

$$|\{0, 1, 2, 3\}| \geq 2$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Feasibility checking

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$|D(X) \cup D(Y)| \geq 2$$

$$|\{0, 1, 2\} \cup \{1, 2, 3\}| \geq 2$$

$$|\{0, 1, 2, 3\}| \geq 2$$

$$4 \geq 2$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Pruning

X

Y

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Pruning

$$\boxed{X} \in \{1\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$D(X) = \{1\}$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Pruning

$$\boxed{X} \in \{1\}$$

$$\boxed{Y} \in \{1, 2, 3\}$$

$$\begin{aligned} D(X) &= \{1\} \\ \Rightarrow D(Y) &\setminus \{1\} \end{aligned}$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Pruning

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{2\}$$

$$\begin{aligned} D(X) &= \{1\} \\ \Rightarrow D(Y) \setminus \{1\} \end{aligned}$$

$$D(Y) = \{2\}$$

Computational Paradigm

- ▶ Consider constraint, $X \neq Y$
- ▶ Pruning

$$\boxed{X} \in \{0, 1, 2\}$$

$$\boxed{Y} \in \{2\}$$

$$\begin{aligned} D(X) &= \{1\} \\ &\Rightarrow D(Y) \setminus \{1\} \end{aligned}$$

$$\begin{aligned} D(Y) &= \{2\} \\ &\Rightarrow D(X) \setminus \{2\} \end{aligned}$$

Until Next Time