CSE 143
Lecture 17

Recursive Backtracking

slides created by Marty Stepp

http://www.cs.washington.edu/143/

Exercise

- Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.
  - Example: `permute("MARTY")` outputs the following sequence of lines:

```
MARTY | MYRAT | ATYMR | RTMAY | TARMY | YMTAR
MARYT | MYRTA | ATYRM | RTMYA | TARYM | YMTRA
MATRY | MYTAR | AYMRT | RTAMY | TAYMR | YAMRT
MATYR | MYTRA | AYMTR | RTAYM | TAYRM | YAMTR
MAYRT | AMRTY | AYRMT | RTYMA | TRMAY | YARMT
MAYTR | AMRYT | AYRTM | RTYAM | TRMYA | YARTM
MRATY | AMTRY | AYTMR | RYMAT | TRAMY | YATMR
MRAYT | AMTYR | AYTRM | RYMTA | TRAYM | YATRM
MRTAY | AMYRT | RMATY | RYAMT | TRYMA | YRMAT
MRTYA | AMYTR | RMAYT | RYATM | TRYAM | YRMTA
MRYAT | ARMTY | RMTAY | RYTMA | TYMAR | YRAMT
MRYTA | ARMYT | RMTYA | RYTAM | TYMRA | YRATM
MTARY | ARTMY | RMYAT | TMARY | TYAMR | YRTMA
MTAYR | ARTYM | RMYTA | TMAYR | TYARM | YRTAM
MTRAY | ARYMT | RAMTY | TMRAY | TYRMA | YTMAR
MTRYA | ARYTM | RAMYT | TMRYA | TYRAM | YTMRA
MTYAR | ATMRY | RATMY | TMYAR | YMART | YTAMR
MTYRA | ATMYR | RATYM | TMYRA | YMATR | YTARM
MYART | ATRMY | RAYMT | TAMRY | YMRAT | YTRMA
MYATR | ATRYM | RAYTM | TAMYR | YMRTA | YTRAM
```

Examining the problem

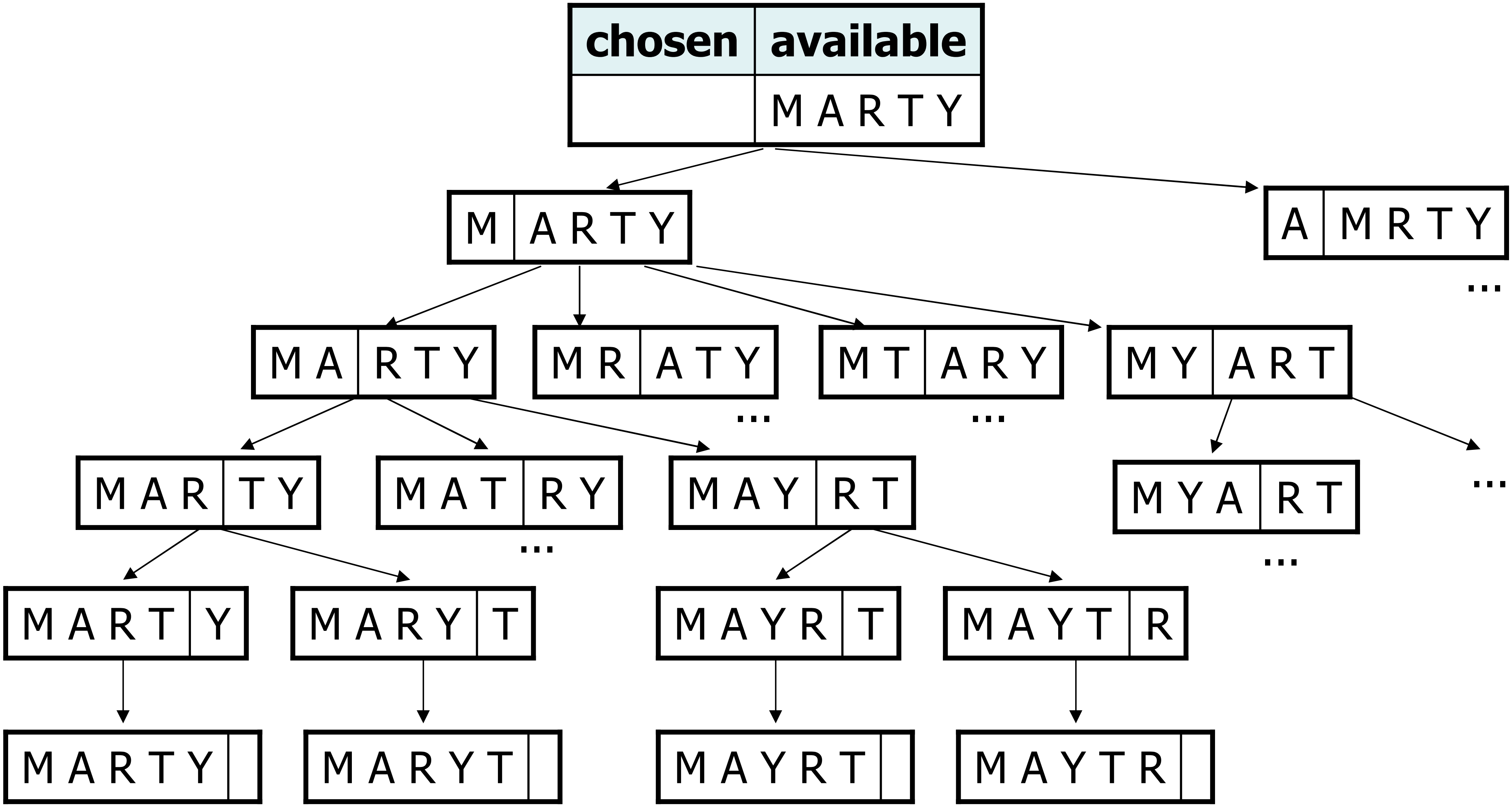
- Think of each permutation as a set of choices or **decisions**:
  - Which character do I want to place first?
  - Which character do I want to place second?
  - ...
  - **solution space**: set of all possible sets of decisions to explore

- We want to generate all possible sequences of decisions.

```
for (each possible first letter):
    for (each possible second letter):
        for (each possible third letter):
            ...
               print!
```

  - This is called a **depth-first search**

Decision trees



chosen	available
	M A R T Y

M | A R T Y

A | M R T Y

...

M A | R T Y

M R | A T Y

...

M T | A R Y

...

M Y | A R T

M A R | T Y

M A T | R Y

...

M A Y | R T

M Y A | R T

...

...

M A R T | Y

M A R Y | T

M A Y R | T

M A Y T | R

M A R T Y |

M A R Y T |

M A Y R T |

M A Y T R |

Backtracking

- **backtracking**: A general algorithm for finding solution(s) to a computational problem by trying partial solutions and then abandoning them ("backtracking") if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths; not clever)
  - often (but not always) implemented recursively

Applications:

- producing all permutations of a set of values
- parsing languages
- games: anagrams, crosswords, word jumbles, 8 queens
- combinatorics and logic programming

Backtracking algorithms

A general pseudo-code algorithm for backtracking problems:

```
explore(choices):
  – if there are no more choices to make:  stop.
  – else:
      • Make a single choice C from the set of choices.
          – Remove C from the set of choices.
      • explore the remaining choices.
      • Un-make choice C.
          – Backtrack!
```

Backtracking strategies

- When solving a backtracking problem, ask these questions:
  - What are the "choices" in this problem?
    - What is the "base case"? (How do I know when I'm out of choices?)

  - How do I "make" a choice?
    - Do I need to create additional variables to remember my choices?
    - Do I need to modify the values of existing variables?

  - How do I explore the rest of the choices?
    - Do I need to remove the made choice from the list of choices?

  - Once I'm done exploring the rest, what should I do?

  - How do I "un-make" a choice?

Permutations revisited

- Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.
  - Example:
    `permute("MARTY")`
    outputs the following sequence of lines:

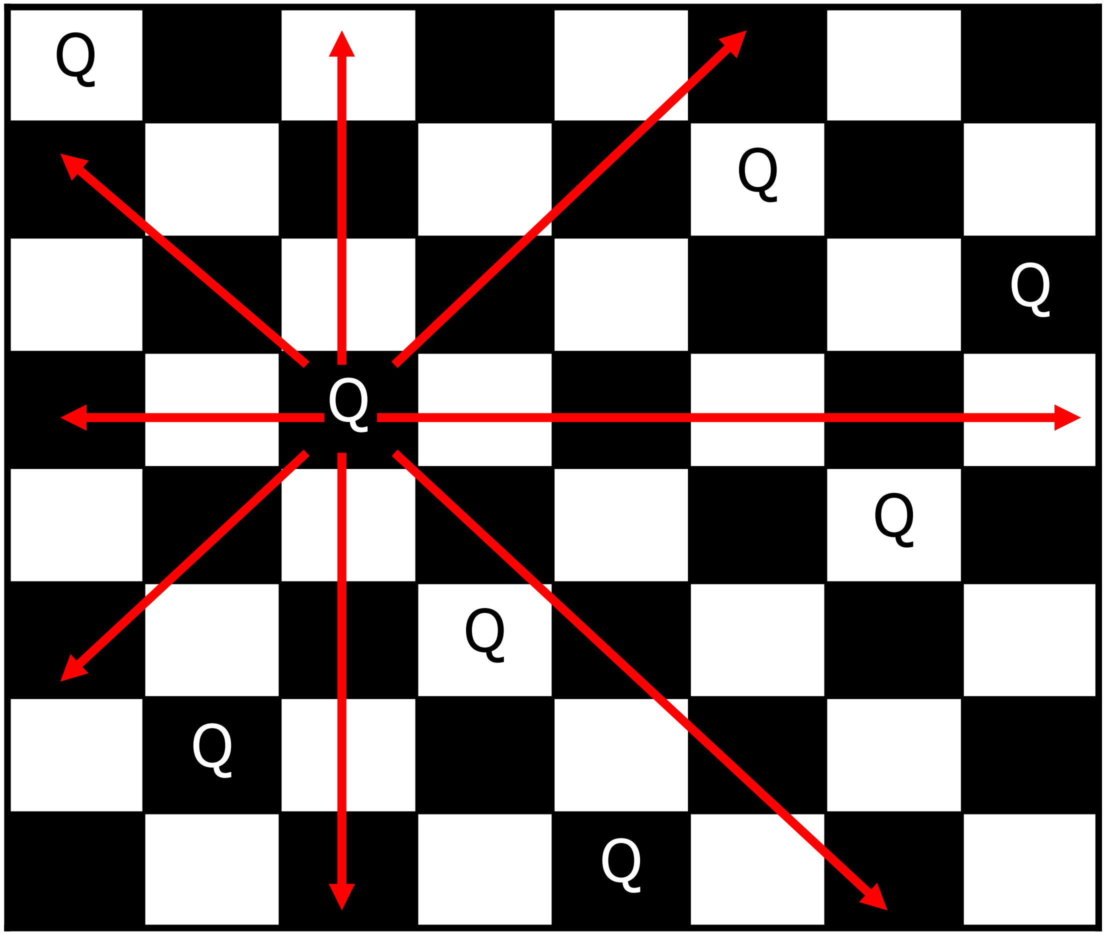
```
MARTY  MYRAT  ATYMR  RTMAY  TARMY  YMTAR
MARYT  MYRTA  ATYRM  RTMYA  TARYM  YMTRA
MATRY  MYTAR  AYMRT  RTAMY  TAYMR  YAMRT
MATYR  MYTRA  AYMTR  RTAYM  TAYRM  YAMTR
MAYRT  AMRTY  AYRMT  RTYMA  TRMAY  YARMT
MAYTR  AMRYT  AYRTM  RTYAM  TRMYA  YARTM
MRATY  AMTRY  AYTMR  RYMAT  TRAMY  YATMR
MRAYT  AMTYR  AYTRM  RYMTA  TRAYM  YATRM
MRTAY  AMYRT  RMATY  RYAMT  TRYMA  YRMAT
MRTYA  AMYTR  RMAYT  RYATM  TRYAM  YRMTA
MRYAT  ARMTY  RMTAY  RYTMA  TYMAR  YRAMT
MRYTA  ARMYT  RMTYA  RYTAM  TYMRA  YRATM
MTARY  ARTMY  RMYAT  TMARY  TYAMR  YRTMA
MTAYR  ARTYM  RMYTA  TMAYR  TYARM  YRTAM
MTRAY  ARYMT  RAMTY  TMRAY  TYRMA  YTMAR
MTRYA  ARYTM  RAMYT  TMRYA  TYRAM  YTMRA
MTYAR  ATMRY  RATMY  TMYAR  YMART  YTAMR
MTYRA  ATMYR  RATYM  TMYRA  YMATR  YTARM
MYART  ATRMY  RAYMT  TAMRY  YMRAT  YTRMA
MYATR  ATRYM  RAYTM  TAMYR  YMRTA  YTRAM
```

Exercise solution

```
// Outputs all permutations of the given string.
public static void permute(String s) {
    permute(s, "");
}

private static void permute(String s, String s2) {
    if (s.length() == 0) {
        // base case: no choices left to be made
        System.out.println(s2);
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            String ch = s.substring(i, i + 1);  // choose

            String rest = s.substring(0, i) +   // remove
                          s.substring(i + 1);

            permute(rest, soFar + ch);          // explore
        }
    }
}
```

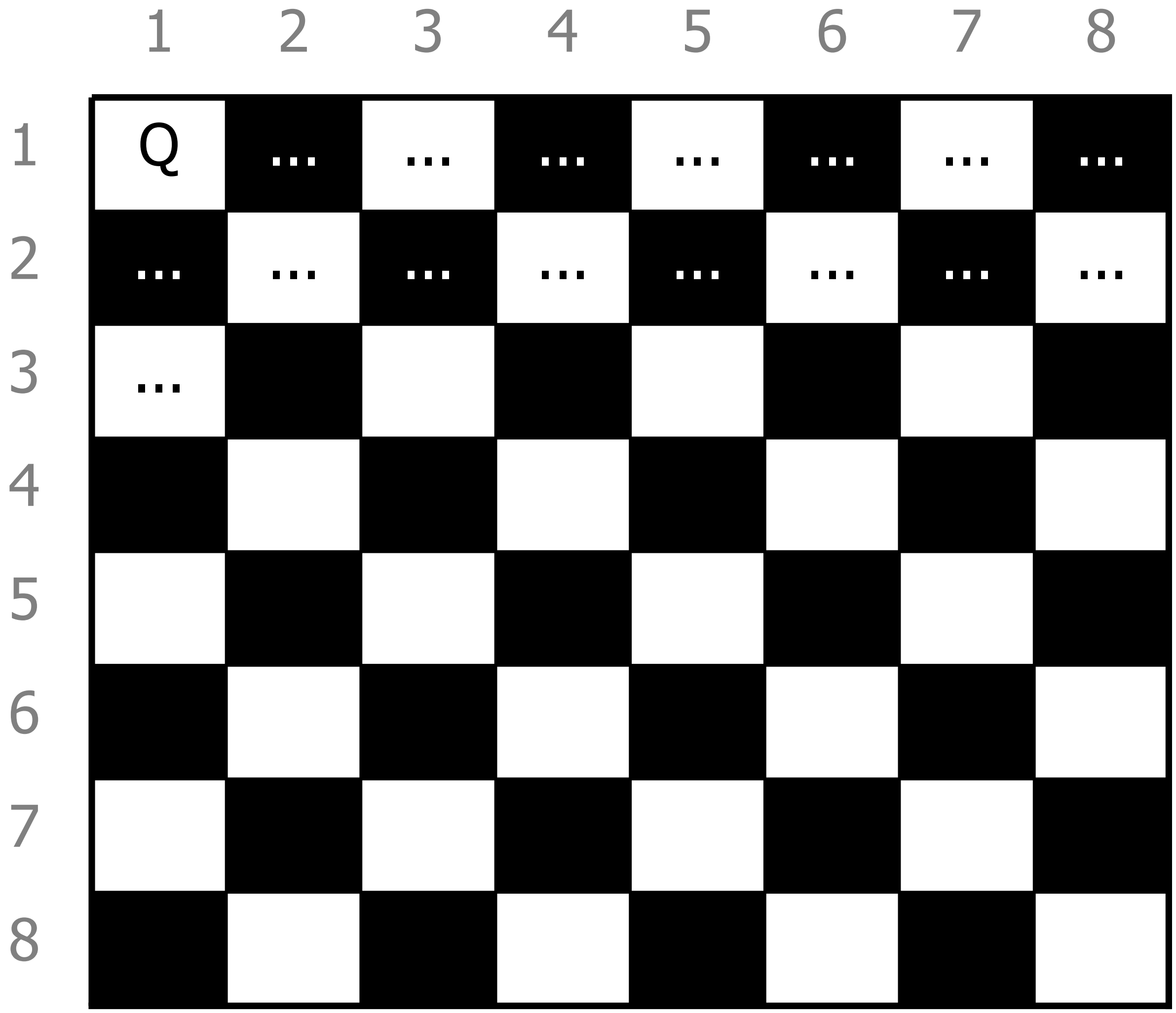
The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



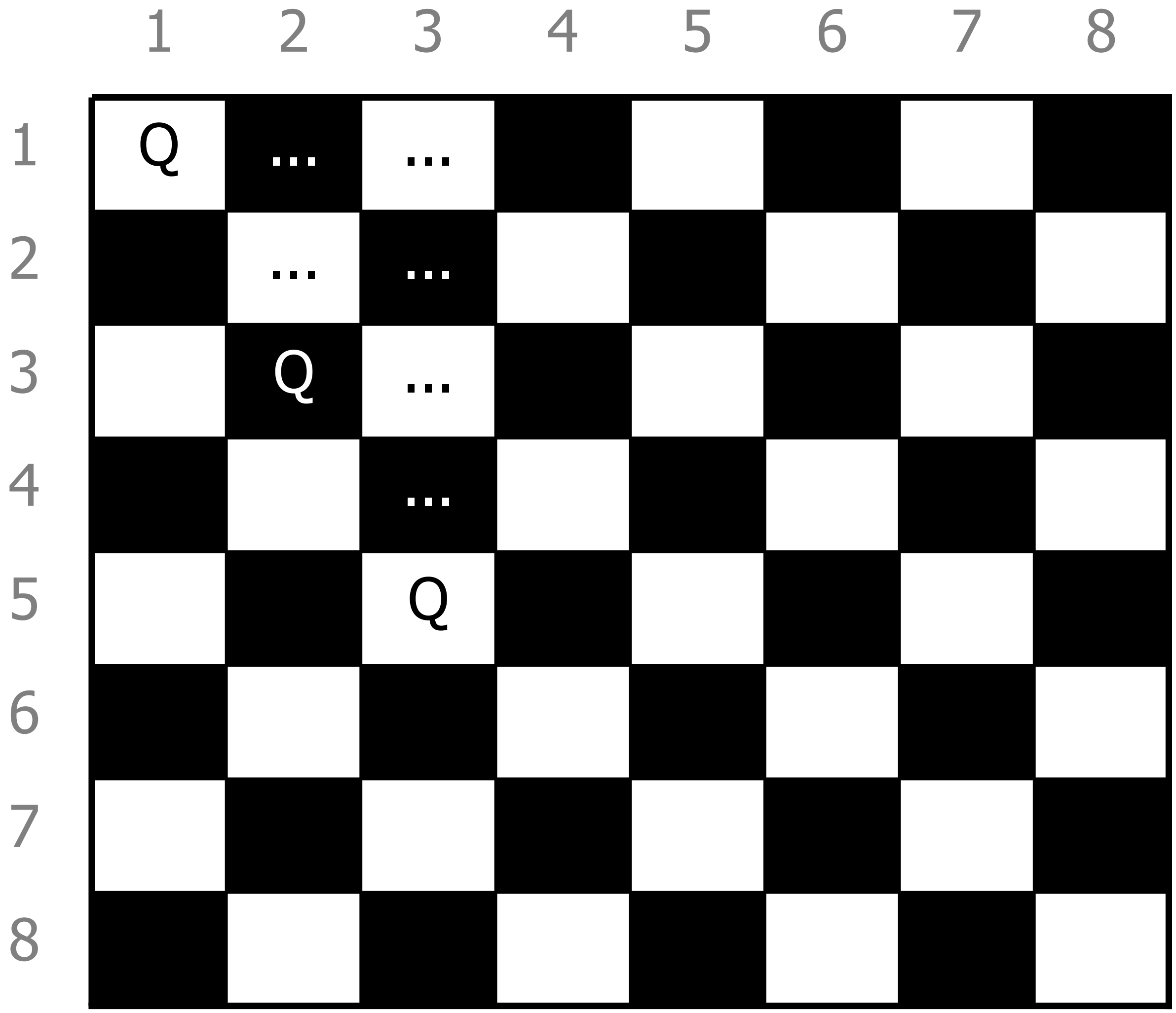
Naive algorithm

- for (each square on the board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.
  - How large is the solution space for this algorithm?
    - 64 * 63 * 62 * …



Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.
  - Redefine a "choice" to be valid placement of a queen in a particular column.
  - How large is the solution space now?
    - 8 * 8 * 8 * …



Exercise

- Suppose we have a `Board` class with the following methods:

Method/Constructor	Description
`public` **`Board`**`(int size)`	construct empty board
`public boolean` **`isSafe`**`(int row, int column)`	`true` if queen can be safely placed here
`public void` **`place`**`(int row, int column)`	place queen here
`public void` **`remove`**`(int row, int column)`	remove queen from here
`public String` **`toString`**`()`	text display of board

- Write a method `solveQueens` that accepts a `Board` as a parameter and tries to place 8 queens on it safely.
  - Your method should stop exploring if it finds a solution.

Exercise solution

```
// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
public static void solveQueens(Board board) {
    if (!explore(board, 1)) {
        System.out.println("No solution found.");
    } else {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    }
}

...
```

Exercise solution, cont'd.

```
// Recursively searches for a solution to 8 queens on this
// board, starting with the given column, returning true if a
// solution is found and storing that solution in the board.
// PRE: queens have been safely placed in columns 1 to (col-1)
public static boolean explore(Board board, int col) {
    if (col > board.size()) {
        return true;   // base case: all columns are placed
    } else {
        // recursive case: place a queen in this column
        for (int row = 1; row <= board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col);              // choose
                if (explore(board, col + 1)) {   // explore
                    return true;   // solution found
                }
                b.remove(row, col);                 // un-choose
            }
        }
        return false;   // no solution found
    }
}
```