CSE 143
Lecture 19

More Recursive Backtracking

reading: "Appendix R" on course web site

slides adapted from Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/

Exercise: solve maze

- Write a method `solveMaze` that accepts a `Maze` and a starting row/column as parameters and tries to find a path out of the maze starting from that position.
  - If you find a solution:
    - Your code should **stop** exploring.
    - You should **mark** the path out of the maze on your way back out of the recursion, using backtracking.
  - (As you explore the maze, squares you set as 'explored' will be printed with a dot, and squares you 'mark' will display an X.)

```
##########
#    xx  #
# ###x## #
# # xx # #
# # x# # #
# ##x#####
# #.xx   #
# #.#x # #
#####x####
#...#xxxx?
#.#..xx#.#
##########
```

Maze class

```
##########
#        #
# ### ## #
# #    # #
# #  # # #
# ## #####
# #      #
# # #  # #
##### ####
#   #
# #    # #
##########
```

- Suppose we have a `Maze` class with these methods:

Method/Constructor	Description
`public` **`Maze`**`(String text)`	construct a given maze
`public int` **`getHeight`**`(),` **`getWidth`**`()`	get maze dimensions
`public boolean` **`isExplored`**`(int r, int c)` `public void` **`setExplored`**`(int r, int c)`	get/set whether you have visited a location
`public void` **`isWall`**`(int r, int c)`	whether given location is blocked by a wall
`public void` **`mark`**`(int r, int c)` `public void` **`isMarked`**`(int r, int c)`	whether given location is marked in a path
`public String` **`toString`**`()`	text display of maze

Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore(**choices**):

– if there are no more **choices** to make: stop.

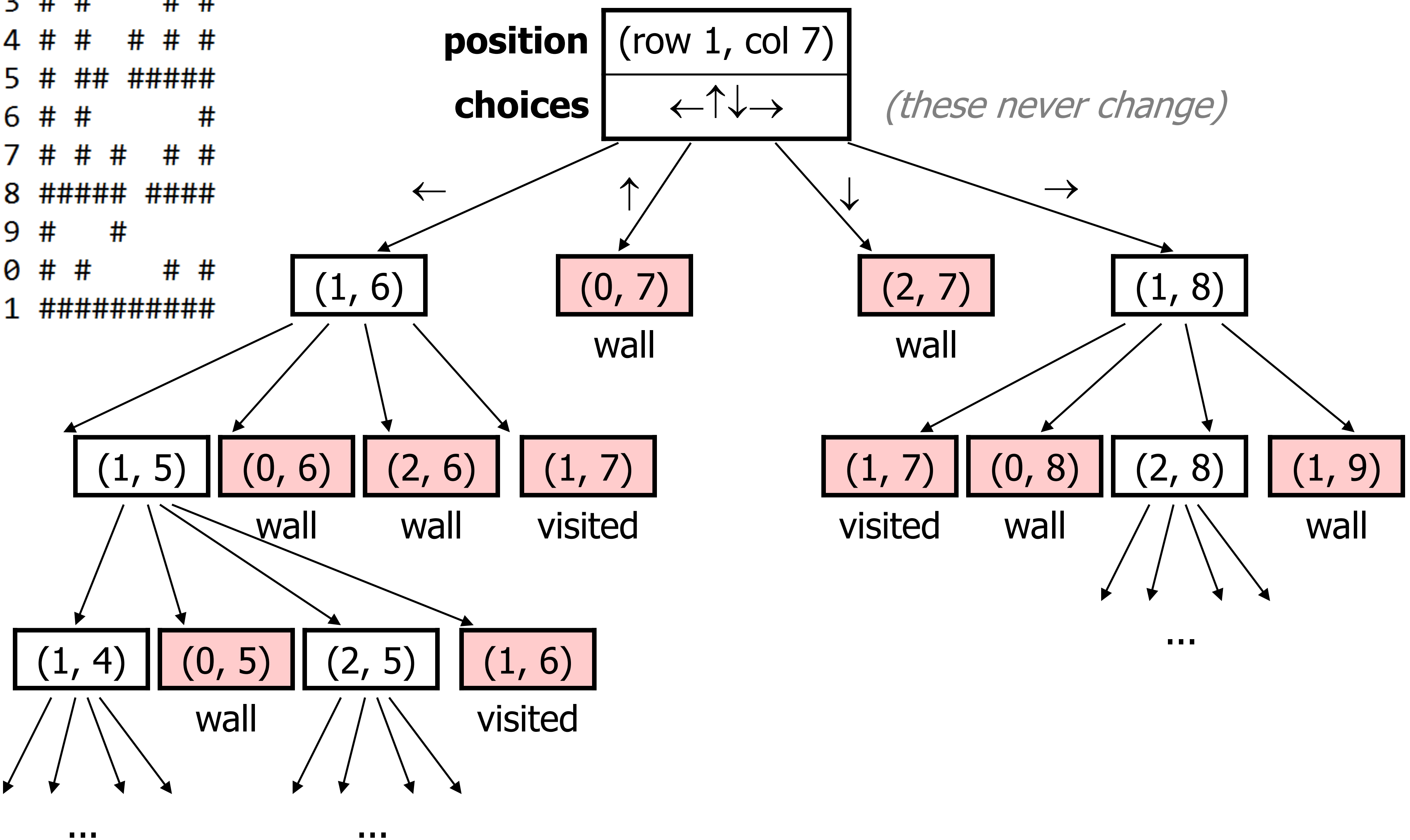
– else, for each available choice **C**:

  - Choose **C**.
  - Explore the remaining **choices**.
  - Un-choose **C**, if necessary. (backtrack!)

What are the choices in this problem?

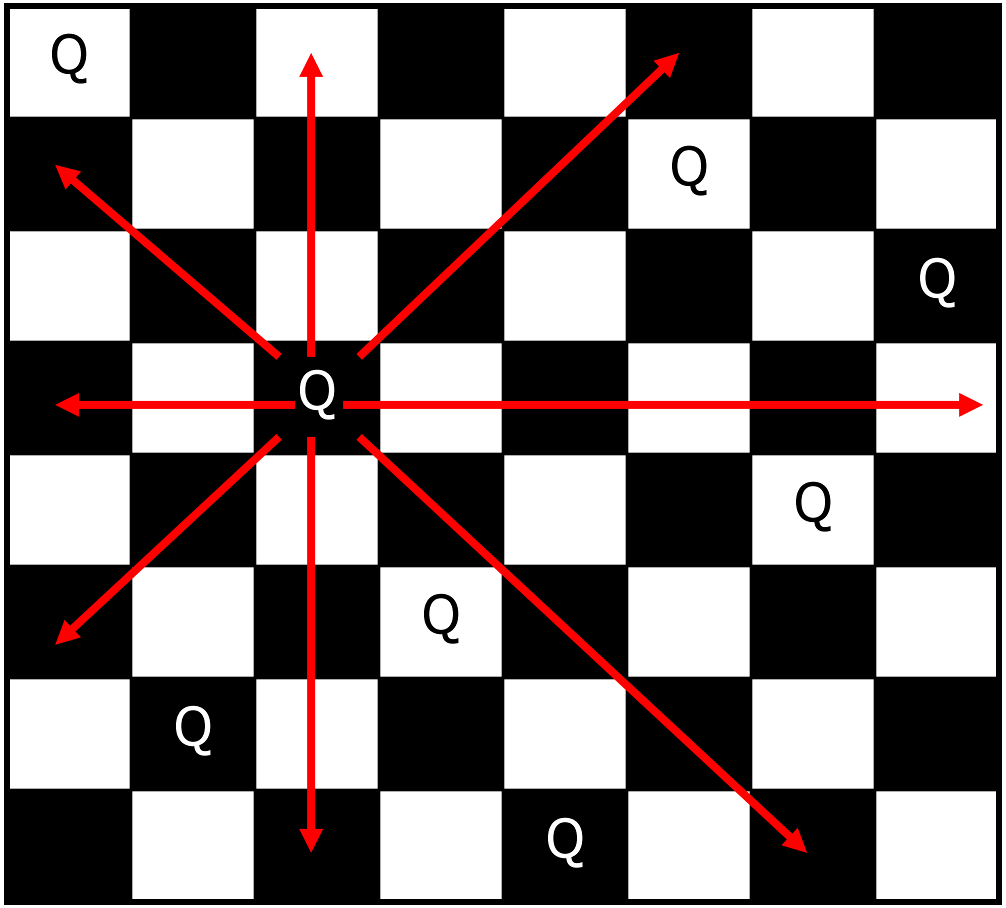
Decision tree

```
  0123456789
0 ##########
1 #        #
2 # ### ## #
3 # #    # #
4 # #  # # #
5 # ## #####
6 # #      #
7 # # #  # #
8 ##### ####
9 #   #
0 # #    # #
1 ##########
```



The "8 Queens" problem

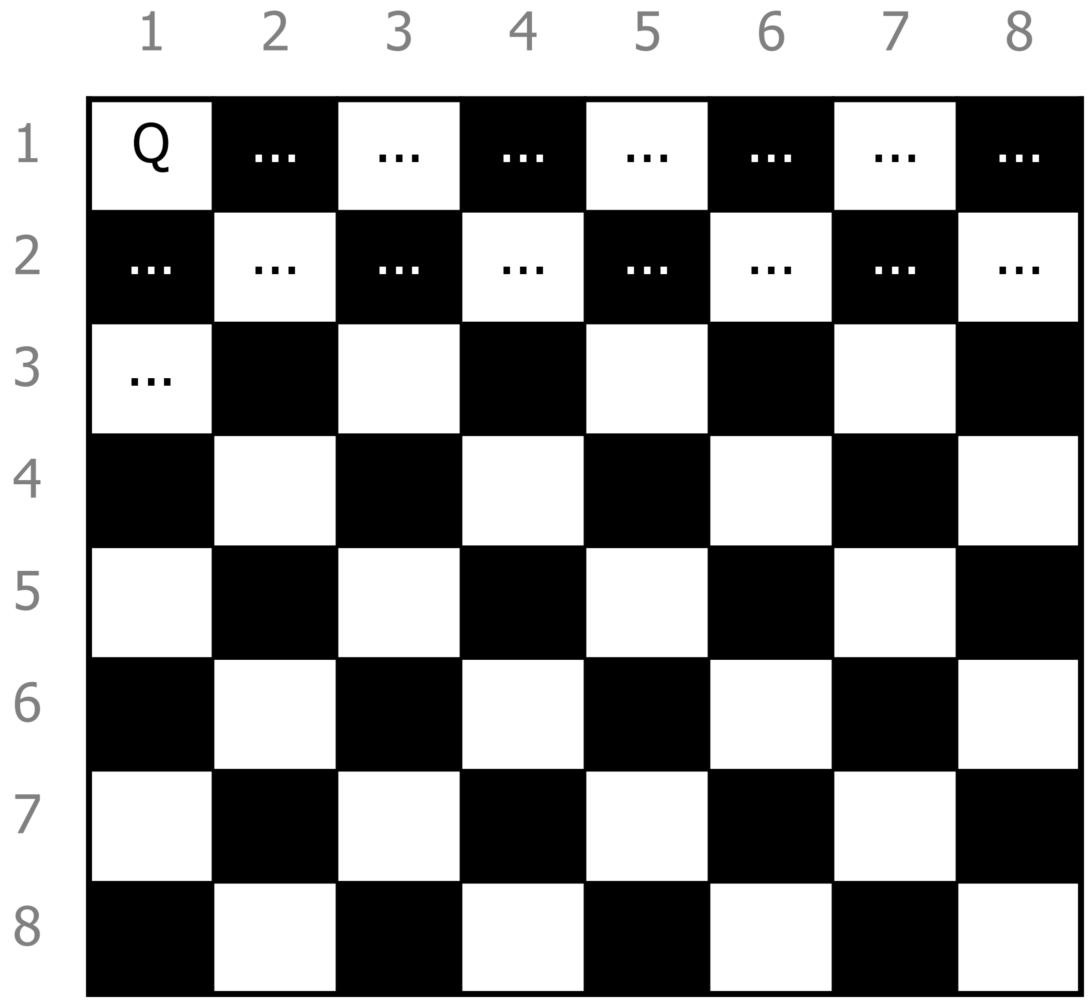
- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



Naive algorithm

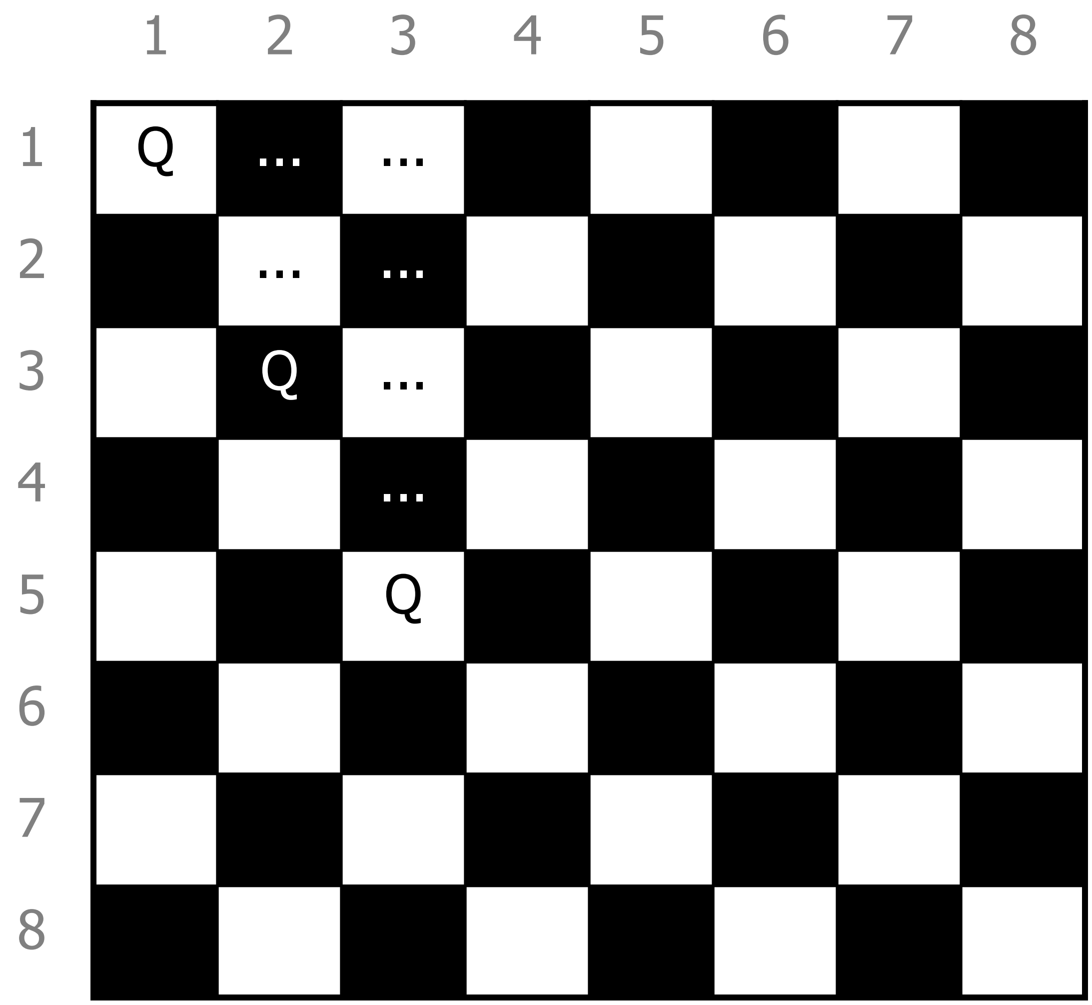
- for (each square on board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.

  - 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

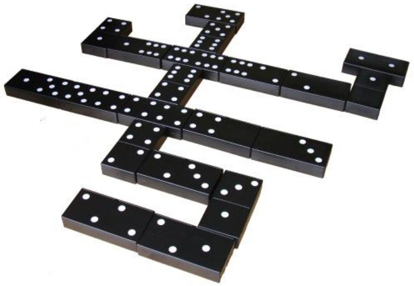
```
// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
public static void solveQueens(Board board) {
    if (solveQueens(board, 1)) {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    } else {
     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

...
```

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 solveQueens(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
    }
}
```

Exercise: Dominoes



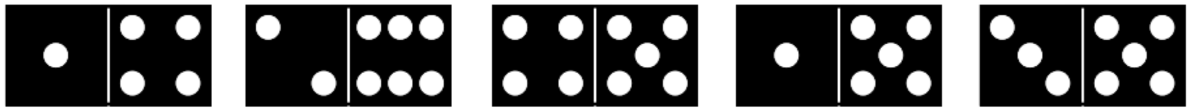
- The game of dominoes is played with small black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.
- Given a class `Domino` with the following public methods:

```
int first()                  // first dots value
int second()                 // second dots value
void flip()                  // inverts 1st/2nd
boolean contains(int n)      // true if 1st/2nd == n
String toString()            // e.g. "(3|5)"
```

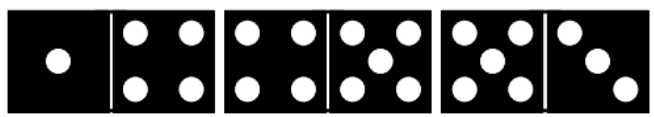
- Write a method `hasChain` that takes a `List` of dominoes and a starting/ending dot value, and returns whether the dominoes can be made into a chain that starts/ends with those values.

Domino chains

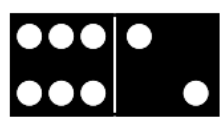
- Suppose we have the following dominoes:



- We can link them into a chain from 1 to 3 as follows:
  – Notice that the 3|5 domino had to be flipped.



- We can "link" one domino into a "chain" from 6 to 2 as follows:



Exercise client code

```
import java.util.*;    // for ArrayList

public class SolveDominoes {
    public static void main(String[] args) {
        // [(1|4), (2|6), (4|5), (1|5), (3|5)]
        List<Domino> dominoes = new ArrayList<Domino>();
        dominoes.add(new Domino(1, 4));
        dominoes.add(new Domino(2, 6));
        dominoes.add(new Domino(4, 5));
        dominoes.add(new Domino(1, 5));
        dominoes.add(new Domino(3, 5));
        System.out.println(hasChain(dominoes, 5, 5));    // true
        System.out.println(hasChain(dominoes, 1, 5));    // true
        System.out.println(hasChain(dominoes, 1, 3));    // true
        System.out.println(hasChain(dominoes, 1, 6));    // false
        System.out.println(hasChain(dominoes, 1, 2));    // false
    }

    public static boolean hasChain(List<Domino> dominoes,
                                   int start, int end) {
        ...
    }
}
```

Exercise solution

```
public boolean hasChain(List<Domino> dominoes, int start, int end) {
    if (start == end) {
        for (Domino d : dominoes) {
            if (d.contains(start)) { return true; }
        }
        return false;                              // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);         // choose
            if (d.first() == start) {              // explore
                if (hasChain(dominoes, d.second(), end)) {
                    return true;
                }
            } else if (d.second() == start) {
                if (hasChain(dominoes, d.first(), end)) {
                    return true;
                }
            }
            dominoes.add(i, d);                    // un-choose
        }
        return false;
    }
}
```

Exercise: Print chain

- Write a variation of your `hasChain` method that also prints the chain of dominoes that it finds, if any.

```
hasChain(dominoes, 1, 3);
```

```
[(1|4), (4|5), (5|3)]
```