Week 8 - Monday

COMP 2000

Last time

- What did we talk about last time?
- Finished Tower of Hanoi
- Merge sort

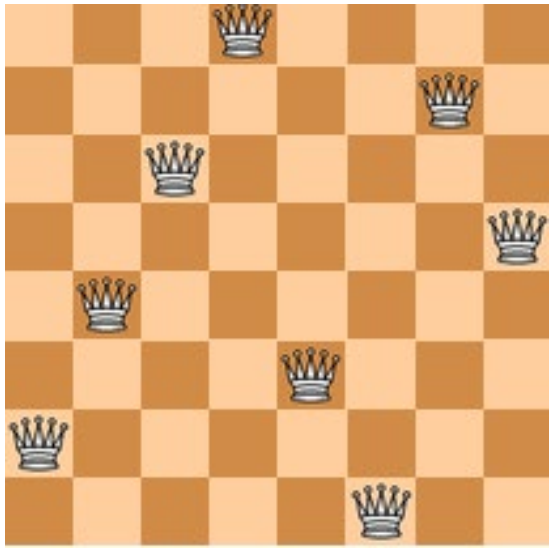
Questions?

Project 3

N-Queens Example

N-Queens

- Given an ***N*** x ***N*** chess board, where ***N*** ≥ 4 it is possible to place ***N*** queens on the board so that none of them are able to attack each other in a given move
- Write a method that, given a value of ***N***, will return the total number of ways that the ***N*** queens can be placed



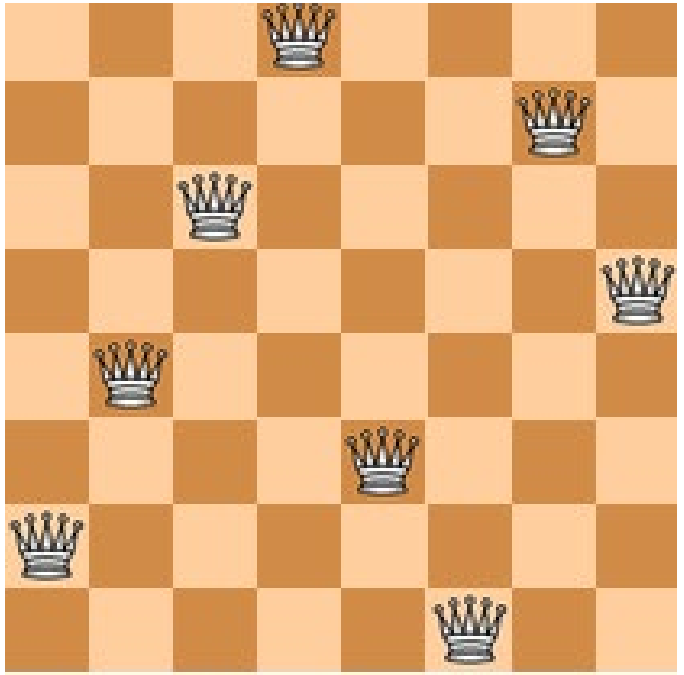
Problem solving approach

- We will use recursion to place queens on the board, one row at a time
- To save typing, we will use a loop to place the queen at each different column within the row and then recurse
  - Egad! A loop inside recursion!
  - It happens.
- If we have placed queens on all the rows, we return 1 (a successful placement)
- We sum up all the successful placements that our recursive children make

Key observations

- We can never have more than one queen in a given row
- Instead of using a 2D array, we can just use a 1D array
- The array will record which column a queen on a given row uses
- Thus, it will be an array of **`int`** values
- The array for the placement to the right would look like:

```
{3, 6, 2, 7, 1, 4, 0, 5}
```



N-Queens algorithm (recursive)

- Base case: (***row*** = 8)
  - You have placed queens on rows 0-7
  - Return 1 (a successful placement)
- Recursive case: (***row*** < 8)
  - Keep a sum of the successful placements made by placing in future rows, initially 0
  - Try to place a queen on columns 0-7
    - For each successful column placement, recursively try to place queens on the next row and add those successful placements to your sum
  - Return sum

Helper method

- As you place a queen on a row, you'll need a method to check if it's safe
  - If it isn't safe, there's no reason to recurse
- We have set up our program so that no queens can ever be on the same row
- We still have to check previous rows to see if they have the same column or diagonal
- Checking the column simply means seeing if the number inside the row is the same
- Checking the diagonal requires more thought
- Use a method with the following signature, where **`board`** is the 1D array of **`int`** values giving column locations and **`row`** is the row you're currently adding to

```
public static boolean isSafe(int[] board, int row)
```

- You only need to look at the locations before row

Files

Files in Java

- At the end of COMP 1600, we briefly mentioned general file I/O
- Think of a file as a stream of bytes
- It's possible to read data from and write data to files
- Files are great because they exist after the program is done executing
- Reading from and writing to text files is very similar to reading and writing to the command line (using **`Scanner`** and **`System.out`**)

Text files

- First, we're going to talk about **text files**
- All files are sequences of bytes stored in binary, but in text files, those bytes form human-readable text like words and numbers
- Unlike files storing data in binary, working with text files is similar to the command-line I/O we've been doing since before COMP 2000
- Examples of text files:
  - Source code for most programming languages (`.c`, `.java`, `.py` files, etc.)
  - Plain text files (often with a `.txt` extension)
  - Many configuration and log files

File extensions

- **File extensions have no real meaning**
- Extensions are part of the name of a file
- A file filled with binary data could end in **`.txt`**
- An audio file could end with **`.jpg`**
- The OS uses extensions to *guess* about which program should open a file
- Changing the extension changes nothing about a file
  - **Caveat:** I've heard that changing an image or audio file extension in macOS will sometimes change the format of the file
- In Windows, you should never hide extensions, since doing so allows the OS to lie to you about the file's real name

Reading

- Reading from a text file is straightforward
- We use **`Scanner`**, just like reading from the command line
- We just have to create a new **`File`** object that gives the file path we want to read from

```
Scanner in = new Scanner(new File("input.txt"));
```

- This code will read from some file called **`input.txt`**, as if someone were typing its contents into the command line

Scanner methods

- Recall that we can read correctly formatted text with a **`Scanner`** using the following methods
  - **`nextInt()`** Reads an **`int`** value
  - **`nextDouble()`** Reads a **`double`** value
  - **`next()`** Reads a white-space delimited **`String`**
  - **`nextLine()`** Reads a **`String`** up to the next newline (which can cause problems if there's a newline left over from previous reads)
- These methods are usually what you need to get the job done, but there are also **`nextBoolean()`**, **`nextByte()`**, **`nextFloat()`**, **`nextLong()`**, and **`nextShort()`** methods
- Note that all the integer reading methods have a second version that takes a base so that you can read values in bases 2-36

New `Scanner` powers

- When a **`Scanner`** is reading from the keyboard, it has no idea what the user will type next
- However, when a **`Scanner`** is reading from a file, it *can* examine the text it hasn't read yet
- A number of methods are available to tell you if there's some properly formatted data just ahead waiting to be read:
  - **`hasNextInt()`** — There's an **`int`** waiting to be read
  - **`hasNextDouble()`** — There's a **`double`** waiting to be read
  - **`hasNext()`** — There's a **`String`** waiting to be read
  - **`hasNextLine()`** — There's a line waiting to be read
- Such methods are often used in a **`while`** loop to keep reading data until the end of the file is reached

Upcoming

Next time…

- Writing files
- Examples
- Error handling

Reminders

- **Start Project 3**
  - **Form teams immediately!**
- Keep reading Chapter 20