Week 7 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Debugging
- Tower of Hanoi

Questions?

Project 2

Tower of Hanoi

Tower of Hanoi

- The Tower of Hanoi is a mathematical puzzle invented by the mathematician Édouard Lucas in the 19th century
- It is a board with three rods
- On the first rod sits a stack of *n* disks in increasing order of size, with the smallest disk on the top
- The goal is to move all of the disks to the third rod
- There are three rules:
  1. You can only move one disk at once
  2. Each move takes the top disk from one rod and puts it on the top of another (possibly empty) stack on another rod
  3. No larger disk may be placed on top of a smaller disk

Solving Tower of Hanoi

- Professor Stucki has a wooden set you can play with
- It's fun to move the disks around, but how can we come up with an algorithm that solves the problem?
- Recursion!

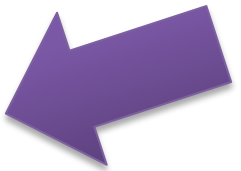
Recursive solution

- Base case (***n*** = 1):
  - If there is only one disk, move it to its destination
- Recursive case (***n*** > 1):
  - First move ***n*** – 1 disks to a temporary pole
  - Then move the n^{th} disk to the destination
  - Then move ***n*** – 1 disks from the temporary pole to the destination

Tower of Hanoi code

```
public static void hanoi(int n, char from, char to,
  char temp) {

  if( n == 1 )
     System.out.println("Move disk from " + from +
          " to " + to);
  else {
     hanoi(n - 1, from, temp, to);
     hanoi(1, from, to, temp);
     hanoi(n - 1, temp, to, from);
  }
}
```



Base Case

Recursive Case

Lessons from Tower of Hanoi

- The recursion is pretty interesting
- You can prove that there's no faster way to do it than the given approach
- But it's very slow!
- 100 disks would take longer than the Universe has been in existence, even on the faster modern computers
- How can we understand how long recursion takes?
- Take COMP 2100 and COMP 4500 to find out!

Merge Sort

Merge Sort algorithm (recursive)

- Beautiful divide and conquer algorithm
- Base case: List has size 1
  - You're done!
- Recursive case: List has size greater than 1
  - Divide your list in half
  - Recursively merge sort each half
  - Merge the two halves back together in sorted order

Merge Sort code

```
public static void mergeSort(int [] array) {


  if(array.length > 1) {
     int[] a = new int[array.length/2];
     int[] b = new int[array.length - a.length];
     for(int i = 0; i < a.length; ++i) //copy first half
          a[i] = array[i];
     for(int i = 0; i < b.length; ++i) //copy second half
          b[i] = array[i + a.length];
     mergeSort(a); //sort first half
     mergeSort(b); //sort second half
     merge(a, b, array);
  }
}
```

(Empty) Base Case

Recursive Case

Merging (the hard part)

- The code to merge two sorted subarrays into a third array trips up a lot of people
- Use three indexes, one for each array
- Always copy the smaller value from the two subarrays
- The tricky part is that you might no longer have anything left to copy from a subarray
- At that point, you must copy from the other subarray
- In other words, always check the validity of an index before using it

Merge code

```
public static void merge(int[] a, int[] b, int[] array) {
	int aIndex = 0;
	int bIndex = 0;
	for(int i = 0; i < array.length; ++i) {
		if(aIndex >= a.length)
			array[i] = b[bIndex++];
		else if(bIndex >= b.length)
			array[i] = a[aIndex++];
		else if(a[aIndex] <= b[bIndex])
			array[i] = a[aIndex++];
		else
			array[i] = b[bIndex++];
	}
}
```

Merging

- I prefer the merging given on the previous slide
- A single **`for`** loop that fills the array makes sense to me
- I'm not a huge fan of using the postincrement operator (**`aIndex++`**), but this is what it's designed for:
  - Getting a value and then incrementing it, all in a single line of code
  - Otherwise, we'd need braces for the cases
- Note that you can combine the four seemingly repetitive cases into three cases (but not two)
- Another way to do the merge is with three **`while`** loops, given on the next slide

Merge code (alternative)

```
public static void merge(int[] a, int[] b, int[] array) {
	int aIndex = 0;
	int bIndex = 0;
	int i = 0;
	while(aIndex < a.length && bIndex < b.length) {
		if(a[aIndex] <= b[bIndex])
			array[i] = a[aIndex++];
		else
			array[i] = b[bIndex++];
		++i;
	}
	while(aIndex < a.length) {
		array[i] = a[aIndex++];
		++i;
	}
	while(bIndex < b.length) {
		array[i] = b[bIndex++];
		++i;
	}
}
```

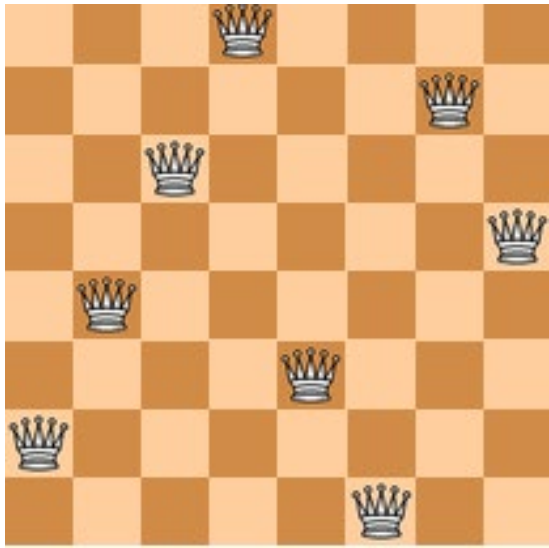
A few things about merge sort

- Merge sort runs in time O(***n*** log ***n***)
- O(***n*** log ***n***) is the fastest **any** comparison-based sort can run (in the worst case)
- The code isn't too hard
- Know how to implement it for job interviews
  - If you write bubble sort in a job interview, you don't deserve the job
- We gave a simple implementation, but a number of clever things can be done to make merge sort go much faster in practice

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



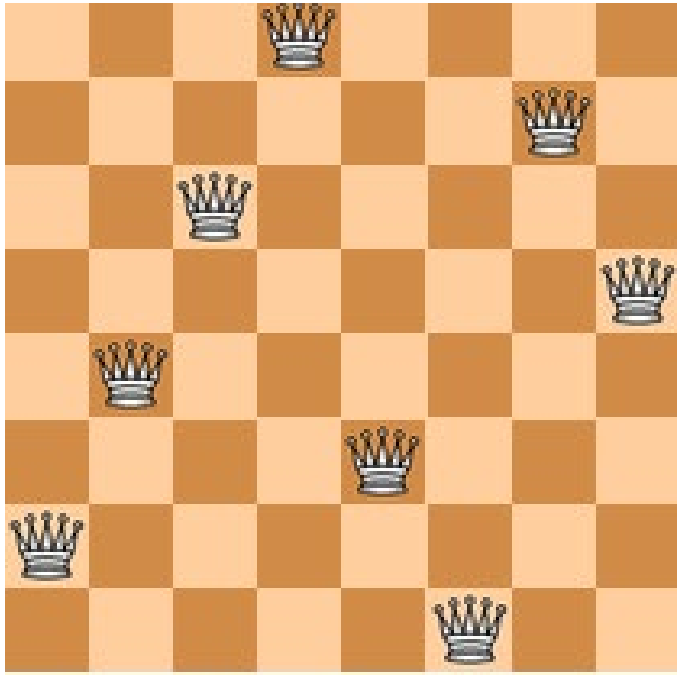
Problem solving approach

- 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

Upcoming

Next time…

- **Spring Break!**
- After Spring Break, finish ***N*-Queens**
- Start reading and writing text files

Reminders

- **Finish Project 2**
  - **Due tonight by midnight!**
- Start reading Chapter 20