built on 2018/05/14 at 09:29:40 ***due:*** *thu may 17 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to provide you with more experience solving algorithmic problems in Java and writing proofs. This assignment requires a starter package, which can be downloaded from the course website.

READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a3.zip`. You'll upload this to Canvas before the assignment is due.
- For the written part, you *must* typeset your answers and hand it in as a PDF file called `hw3.pdf`, which will go inside your zip file. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted.
- Be sure to **disclose your collaborators in the PDF file.**
- For each task, save your work in a file as described in the task description.
- A script will process and grade your submission before any human being looks at it. *Do not use different function/file names.* The script is not as forgiving as we are.
- Use the Internet to help you learn: It's OK to look up syntax or how a function/class/method is used. It's **NOT** OK to look up how to solve a problem or answers to a problem. The goal here is to learn, not to just hand in an answer. If you wish to do that, just give us a link to the solution!
- You are encouraged to work with other students. However, **you must write up the solutions separately on your own and in your own words**. This also means you must not look at or copy someone else's code.
- Finally, the course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Collaboration Policy: To facilitate cooperative learning, you are permitted to discuss homework questions with other students provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion or what is written on the board. *The fact that you can recreate the solution from memory is taken as proof that you actually understood it, and you may actually be interviewed about your answers.*

Exercise 1: Tail Sum of Squares (2 points)

Consider the following snippet of Java code:

```
int sumHelper(int n , int a) {
    if (n==0) return a;
    else return sumHelper(n-1, a + n*n);
}

int sumSqr(int n) { return sumHelper(n, 0); }
```

Your Task: Prove that for $n \geq 1$, `sumSqr(n)` $\hookrightarrow 1^2 + 2^2 + 3^2 + \cdots + n^2$. To prove this, use induction to show that `sumHelper` computes the "right thing." *(Hint: How did we prove `fact_helper` in class?)*

Exercise 2: Mysterious Function (2 points)

Consider the following Python function `foo`, which takes as input an integer $n \geq 1$ and returns a tuple of length 2 of integers:

```
def foo(n):
    assert n>=1
    if n == 1:
        return (1, 2)
    else:
        p, q = foo(n-1)
        return (q + p*n*(n+1), q*n*(n+1))
```

Prove that for $n \geq 1$, `foo(n)` $\hookrightarrow (p,q)$ such that

$$\frac{p}{q} = 1 - \frac{1}{n+1}.$$

(*Hint:* induction on n.)

Exercise 3: Missing Tile (4 points)

For this task, save your code in `MissingTile.java`

There are *four* ways an L-shaped triomino can be arranged. Labeled by the missing corner, the four arrangements are:

In your Discrete Math class, you proved the following theorem:

> **Theorem:** Any 2^n-by-2^n grid with one painted cell can be tiled using L-shaped triominoes such that the entire grid is covered by triominoes but no triominoes overlap with each other nor the painted cell.

The proof was by induction. There are two tasks in this exercise:

Task I: We know you've seen this proof already. But so that you fully understand how it works, you'll prove this theorem again, in your own words, by induction on n.

Task II: You'll turn this proof into code. Specifically, you'll implement a function

`public static void tileGrid(Grid board)`

that takes as input a `board` instance and tile it using information obtained from the instance. The `board` instance implements the `Grid` interface with the following details:

```
// The top-left corner is coordinate x=0 and y=0. The bottom-right corner is
// coordinate x=size()-1 and y=size()-1.
public interface Grid {

  // return the width/height of the square grid. coordinates in the grid are
  // numbers between 0 and size()-1, inclusive.
  int size();

  // return the x coordinate of the painted cell
```

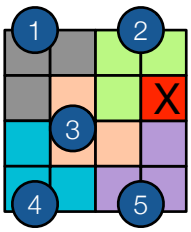
```
  int getPaintedCellX();

  // return the y coordinate of the painted cell
  int getPaintedCellY();

  // install a triomino tile so that the missing corner is at coordinate (x, y)
  // and the orientation of the tile is given by the orientation parameter,
  // encoded as follows: missing NE = 0, missing SE = 1, missing SW = 2, and
  // missing NW = 3. The function returns true if the tiling is possible; it
  // returns false if the tiling failed (e.g., overlap with an existing tile or
  // touches the painted cell.
  boolean setTile(int x, int y, int orientation);


  // return a boolean value indicating whether the grid has been fully tiled
  boolean isFullyTiled();
}
```

As an example, suppose you are given a `board` instance whose size is 4-by-4 and the painted cell is at coordinate $x = 3$ and $y = 1$. This means `board.size()` will return 4, `board.getPaintedCellX()` will return 3, and `board.getPaintedCellY()` will return 1. Moreover, it can be (manually) tiled by calling the `setTile` function as follows:



(1) `board.setTile(1, 1, 1)`
(2) `board.setTile(3, 1, 1)`
(3) `board.setTile(2, 1, 0)`
(4) `board.setTile(1, 2, 0)`
(5) `board.setTile(2, 2, 3)`

Performance Expectations: We'll test your code with grid sizes between 1×1 and 1024×1024, but nothing larger. A call to your function should return within 2 seconds.

Test Driver: To help you get started, we're providing an accompanying Java program in your starter pack that implements the `Grid` interface. You shouldn't need to modify this program. If you modify it, keep in mind that the grader program will use our version of the `Grid` implementation, not yours. The test driver can be found in `TileDriver.java`.

Exercise 4: HackerRank Problems (6 points)

For this task, save your code in `hackerrank.txt`

You'll begin by creating an account on `hackerrank.com` if you don't have one already, so you can solve this set of problems. There are *three* problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website.

Important: You will write down your Hacker ID username in a file called `hackerrank.txt`, which you will submit as part of the assignment. This will be used to match you with your submission on HackerRank.

You can find your problems at

`https://www.hackerrank.com/muic-data-structures-t-317-assignment-3`

Exercise 5: Midway Tower of Hanoi (6 points)

For this task, save your code in `Midway.java`

Monks at a remote monastery are busy solving the Tower of Hanoi problem, a duty passed on for tens of generations. According to an old tale, when this group of monks finishes, P will be shown to equal NP and the world will come to an end.

In Tower of Hanoi, you are given N disks, labeled $0, 1, \ldots, N-1$ by their sizes. In addition, there are 3 pegs: Peg 0, Peg 1, and Peg 2. Initially, all disks are at Peg 0, neatly arranged from small (Disk 0) to large (Disk $N-1$), with the smallest one at the top and the largest one at the bottom. The goal is to transfer all these disks to Peg 1. You can move exactly one disk at a time. However, at all times, a bigger disk cannot be placed on top of a smaller one.

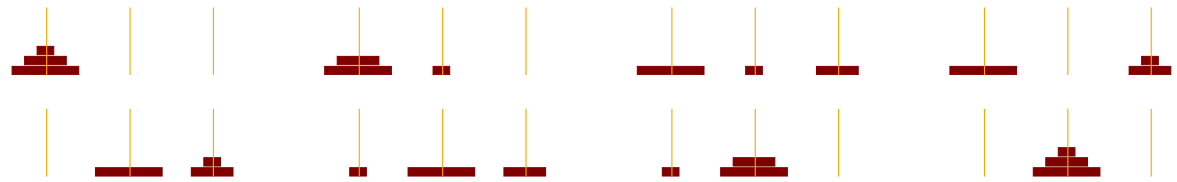
Solving this seemingly complex task turns out to be pretty simple. The following Python code prints out instructions that use the fewest number of moves—in other words, the best possible solution!

```
def solve_hanoi(n, from_peg, to_peg, aux_peg):
    if n>0:
        solve_hanoi(n-1, from_peg, aux_peg, to_peg)
        print("Move disk", n-1, "from Peg", from_peg, "to Peg", to_peg)
        solve_hanoi(n-1, aux_peg, to_peg, from_peg)

solve_hanoi(n, 0, 1, 2)
```

If you follow these steps, you can show that you'll use $2^N - 1$ moves in all. Now this is a large number and carrying out all the steps seems like eternity. So the monks, out of boredom, came up with a puzzle for you: Let's assume that they've strictly followed the instructions the Python program above generated. Given an intermediate configuration, can you figure out how many more steps they are going to need before completing the task?

Here is a visual trace showing all the configurations obtained by following the Python program's instructions:



Task I: You'll begin by showing a useful property. Prove, using mathematical induction, that for any $n \geq 0$, `solve_hanoi(n, ...)` generates exactly $2^n - 1$ lines of instructions.

Task II: To solve the puzzle, you'll implement a function **`public static long`** `stepsRemaining(int[] diskPos)` that takes in the current positions of the disks and returns the number of steps that remain in the computer-generated instructions. The current positions are given as an array of integers: the length of the `diskPos` indicates how many disks there are, and `diskPos[i]` $\in \{0,1,2\}$ indicates the peg at which Disk `i` is. We guarantee that $0 \leq$ `diskPos.length` ≤ 63. As examples (bastardizing Java syntax):

- `stepsRemaining({0})` should return 1.
- `stepsRemaining({2, 2, 1})` should return 3.
- `stepsRemaining({2, 2, 1, 1, 2, 2, 1})` should return 51.

(*Hint:* Solve the following inputs by hand: `{2, 2, 0}` and `{1, 2, 0}`. How many moves do we make before we move the largest disk?)

Performance Expectations: We expect your code to return within 1 second and use only a reasonable amount of memory (e.g., don't explicitly generate the whole instruction sequence).

Task III: The number of lines printed by the Python program above can be expressed as the recurrence

$$f(n) = 2f(n-1) + 1, \text{ with } f(0) = 0.$$

As you showed in a previous task, $f(n)$ has a closed-form of $f(n) = 2^n - 1$. You may use this knowledge in the current task even if you did not solve Task I.

Consider the following program:

```
void printRuler(int n) {
    if (n > 0) {
        printRuler(n-1);
        // print n dashes
        for (int i=0;i<n;i++) System.out.print('-');
        System.out.println();
        // --------------
        printRuler(n-1);
    }
}
```

Let's count the total number of dashes printed. If we are to write a recurrence for that, we will get

$$g(n) = 2g(n-1) + n, \text{ with } g(0) = 0,$$

where the additive n term stems from the fact that we print exactly n dashes in that function call.

The two recurrences are strikingly similar. In this problem, we'll analyze $g(n)$ using our knowledge of $f(n)$. Since $f(n)$ and $g(n)$ have similar recurrences, differing only in an n term, we're going to guess that

$$g(n) = a \cdot f(n) + b \cdot n + c \tag{1}$$

The following steps will guide you through determining the values of a, b, and c—and verifying that our guess indeed works out. In your writeup, **clearly show your work.**

(i) We'll first figure out the value of c. What do you get when plugging in $n = 0$ into equation (1)? It helps to remember that $f(0) = g(0) = 0$. (*Hint: c* should be 0.)

(ii) To figure out the values of a and b, we'll plug in $g(n)$ from equation (1) into the recurrence $g(n) = 2g(n-1) + n$. You should be able write it as

$$\underbrace{\left(\dots\right)}_{=P} n + \underbrace{\left(\dots\right)}_{=Q} = 0$$

and solve for a and b such that $P = 0$ and $Q = 0$.
Keep in mind: Because $f(n) = 2f(n-1) + 1$, we know that $f(n) - 2f(n-1) = 1$.

(iii) Derive a closed form for $g(n)$.

(iv) Use induction to verify that your closed form for $g(n)$ actually works.