built on 2024/10/18 at 17:17:59 ***due:*** *fri nov 01 @ 11:59pm*

This assignment contains both programming and written portions. The programming assignment will give you more practice understanding the connection between induction and recursion, as well as becoming well-versed with implementing data collection classes. You will implement a few Java programs, test them thoroughly, and hand them in. There is a starter package, which you must download to begin working on this assignment. Note that for the written portion, **you must typeset your answer**.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Logistics: The Starter Package and Handing In

We've created a starter zip file for you to get started with this assignment. It is available for download from the course website. When unzipped, the starter file at the top level contains *two* folders. We've set up Gradle for you, so each of the folders has a `build.gradle`.

Additionally, there is a `game.jar` at the top level of the starter folder. This is the original game jar mentioned in the TextTwist task.

```
Top Level
├── game.jar -> TextTwist sample game
├── histogame
│   └── ⟨other files for histogram and
│       TextTwist⟩
└── smallman
    ├── build.gradle
    └── ⟨other files for other tasks⟩
```

What to Hand In?

- For the written portion, you will typeset your answer and produce a PDF file called `hw4.pdf`. No other format will be accepted.
- For the programming portion, only hand in the relevant source files—do *not* hand in binaries (like `.class` or the given `.jar` file).
- To avoid huge submissions full of junk, you will run the `clean` task in Gradle for each project. This can be done either using the Gradle panel in IntelliJ or on the command line via "gradlew clean".
- When you're ready to submit, you'll zip all your files, including `hw4.pdf`, as `hw4.zip` and upload it Canvas.

Task 1: Missing Tile (6 points)

For this task, save your work in `smallman; 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:

Subtask 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.

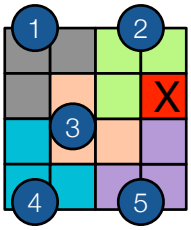
Subtask II: You'll turn this proof into code. Inside `MissingTile.java`, you'll implement a function

```
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)`

In the above example, the line `board.setTile(1, 1, 1)` places a triomino that is missing the SE corner (because its 3rd argument is 1) and the location of the missing corner is (1,1) in the grid. See the description of `setTile` above further explanation.

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`.

Getting Started Guide

Getting started with this task can appear duanting. Here are some tips. If we examine the inductive proof closely, it tells us that to tile a 2^k-by-2^k grid, we will recursively tile *four* 2^{k-1}-by-2^{k-1} subgrids of the big grid, *each with a different painted cell.*

The key question in implementing this logic is, how do we account for each subgrid with a painted cell that potentially differs from ours? Let's take a look at 2 ideas (pick only one and implement it):

- **Idea #1:** Write a helper method with at the least the following parameters:

```
static void tilingHelper(int size, int topX, int topY,
                         int paintedX, int paintedY, Grid bd)
```

- **Idea #2:** If the number of parameters in the helper method above makes you dizzy, you may find the following design more appealing. The idea is known more generally as creating and acting on a *view* of the real object. That is, every time we want to subdivide the grid into a smaller one and specify where the painted cell is, we will return an instance that provides the same interface as `Grid`, except that

  `size`, `getPaintedCellX`, `getPaintedCellY`

  will reflect the information of the subgrid—and when

  `setTile`

  is called, it will "forward" the command to the real grid, compensating for the position accordingly.

  For this, the easiest way is to implement an inner class of `MissingTile`. Let's call this `GridView`, which will implement the `Grid` interface. This means it will support all the methods of the interface (You don't really need to make `isFullyTiled` work). Additionally, it should at least have the following constructors/methods:

```
public GridView(Grid parentGrid);
public GridView subgrid(int topX, int topY, int size,
                        int virtualPaintedX, int virtualPaintedY);
```

You may find it convenient to add even more methods.

Task 2: Histogram (4 points)

For this task, save your work in `histogame/`, package: histogram

Ground Rules. The only files you can modify are

`SimpleHistogram.java` and `SimpleHistogramTest.java`.

You are to implement a histogram class to keep track of data frequencies. The set of possible data values is called the *domain*. Each domain value has a corresponding numerical frequency count. For example, "ant" has appeared 5 times. We have provided an generic interface called `Histogram<DT>`, where `DT` is a type parameter for the histogram's domain type. It defines the following methods:

```
// Returns the total frequency count of all items in the domain combined.
public int getTotalCount()

// Returns the frequency count of a given domain item. If invalid domain
// item is given, return 0.
public int getCount(DT item)

// Sets the frequecy count of a given domain item. If the  domain item
// doesn't yet exist in the domain, this will also add it to the domain.
public void setCount(DT item, int count)
```

Your task is to implement a `SimpleHistogram<DT>` class with the following declaration:

```
public class SimpleHistogram<DT> implements Histogram<DT>, Iterable<DT>
```

This means the class additionally contains a method `iterator()`, which returns an iterator that goes over the domain (in any order). Moreover, it is expected that the class has the following constructors:

```
// constructs an empty histogram (with no information)
public SimpleHistogram()

// constructs a histogram from a list of items given by the parameter items
public SimpleHistogram(DT items[])

// constructs a (new) histogram from an existing histogram, sharing nothing
// internally
public SimpleHistogram(Histogram<DT> hist)
```

As a good citizen of the Java land, your `SimpleHistogram` class must also implement a useable `toString` method, as well as a proper `equals`. In this regard, two histograms `a` and `b` are equal if they have the exact same domain values and the frequency of each item is identical in both histograms.

Testing. To get you started, we have provided some tests in `HistogramTest`. Make sure you check your implementation against them before submission. We encourage you to add in more tests.

Task 3: TextTwist Game (6 points)

For this task, save your work in `histogame/`, package: game

In this task, we will implement a game called *TextTwist*. Here's the source of inspiration in case you haven't heard of it: https://www.wordplays.com/wordgames/text-twist. In short, the game starts by displaying 6 random letters and ask the player to form words using those letters. The game ends when all words that can be formed have been discovered or the time is up.

This task requires a working histogram implementation from a previous task. Prior to attempting this task, you should make sure that your histogram implementation work as expected. There are two parts to this task.

Part I: Word and WordDatabase

For this part, the following two files are the only files that you can modify:

`Word.java` and `WordDatabase.java`.

The other files must remain untouched.

Word

You are to implement classes to represent words and a collection of words. An object of the `Word` class will be used to represent a word. The class implements two interfaces: `Formable` (provided) and `Comparable` (from `java.lang`). That is to say, it has the following declaration:

```
public class Word implements Formable<Word>, Comparable<Word>
```

Each `Word` stores two pieces of information: a `String` representation of the word and a `Histogram` that describes the frequency of each character in the word. Your `Word` must implement the following constructors/methods:

```
// The only constructor of the class, which takes a String representation
// of the word. The histogram is generated here from the given word.
public Word(String word)

// Returns the String representation of the word.
public String getWord()

// Returns a Histogram describing the character distribution of the word.
public Histogram<Character> getHistogram()

// Returns true if the Word represented by other can be formed using some
// or all of the characters of this word.
public boolean canForm(Word other)

// Return -1 if this word is shorter than the word represented by o OR
// when this word and the word represented by o have the same length but
// this word comes before the word represented by o alphabetically.
// Return zero if this word and o word are identical.
// Return +1 otherwise.
public int compareTo(Word o)
```

You might already sense that the method `canForm` will be key in implementing this game; methods from `Histogram` should come in handy here.

Word Database

Next, you will implement a `WordDatabase` that implements the interface `IDatabase`. Specifically, you need to implement the following constructors/methods:

```
// Load all the words from a file given by filename.
public WordDatabase(String filename) throws FileNotFoundException

// Adds a word to the database.
public void add(Word w)

// Removes w from the database and has no effects if w is not present
public void remove(Word w)

// Returns a List of Words in the database whose length is exactly l.
public List<Word> getWordWithLength(int l)

// Returns a List of Words in the database whose length is at least minLen
// and which can be formed from all or some of the letters of the word w
public List<Word> getAllSubWords(Word w, int minLen)

// Returns true if the word o is in the database and false otherwise
public boolean contains(Word o)
```

NOTES: We haven't discussed how to work with files in this class at all. But everything is pretty much the same as back in Python. You'll find the following tutorial useful: https://www.baeldung.com/reading-file-in-java

(We are reading a file containing lines of `Strings`. You should be happy with `FileReader` and perhaps `BufferedReader` if you want a bit better performance).

Finally, we have provided a dictionary file at `src/main/resources/linuxwords.txt`. To read this, you might find the following (magic) lines useful:

```
// Suppose we're using it from a static method inside a class "ClassName"
//   To read from src/main/resources/linuxwords.txt, begin by creating
//   an input stream, like so:
InputStream is = ClassName.class.getClassLoader()
                    .getResourceAsStream("linuxwords.txt");

// If we're using it from an ordinary method:
//   To read from src/main/resources/linuxwords.txt, begin by creating
//   an input stream, like so:
InputStream is = getClass().getClassLoader().getResourceAsStream("linuxwords.txt");
```

Change the hardcoded filename string as appropriate.

Part II: Implementing the Game

...and now let the wild rumpus start! We will put everything together to make our own version of TextTwist. For this part, you will work on the `TextTwist` class. You're welcome to adapt the skeleton from Zuul you have from the previous assignment. Your implementation should be as close as our sample game as possible. You can play our sample game by running the following on the command line:

```
java -cp game.jar TextTwist
```

Your code should support *at least* the following 3 commands:

- `q` – Quits the program
- `?` – Prints out the list of words that can be formed. Reveal the correctly guessed words and hide the rest. This list should be sorted by length.
- `!` – Reveals the answer and starts a new game (i.e., give up and reset!).

HINTS & TIPS: The class that supports randomization is called `Random`. You already know how to measure elapsed time—`System.currentTimeMillis()` or `System.nanoTime()`. This game is case-insensitive: upper and lower cases are the same.

Here's how the program will be graded: 90% for functionality.

- Randomization of the letters. What you randomize must contain at least ONE 6-letter word.
- The elapsed time is properly displayed.
- The score is properly recorded and displayed.
- The command `?` works as expected.
- The command `!` works as expected.
- The command `q` works as expected.

The remaining 10% is reserved for creativity. Feel free to add any feature that you desire. You must indicate as comment in your code the feature that you have added.

Task 4: 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?)*

Task 5: 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.)

Task 6: Simple Priority Queue (6 points)

For this task, save your work in `smallman; MyPriorityQueue.java`

Like a queue, a *priority queue* is a container that allows users to add and remove objects. The add operation lets the users freely add objects at any time. However, unlike a queue, the remove operation on the priority queue always removes the smallest object in the container.

In this problem, you are to implement a simple priority queue, declared as

```
public class MyPriorityQueue<T> implements IPriorityQueue<T>
```

The generic type parameter `T` specifies the object type that can be added to the container. The class has only one constructor. Because we'll need a way to compare elements of type `T`, the constructor takes a `CompareWith<T>`, like so:

```
public MyPriorityQueue(CompareWith<T> cc)
```

The `CompareWith<T>` interface provides a method `boolean lessThan(T a, T b)`, which returns `true` if and only if `a` is less than `b`. This is how we will compare elements in the priority queue's container.

The interface `IPriorityQueue<T>`, which our class implements, expects the following methods:

```
// adds an item
public void add(T item)

// adds a list of items
public void addAll(List<T> items)

// returns the smallest item currently in the container (if there are multiple
// such items, return any one of them)
public T getMinimum()

// removes the smallest item in the container (if there are multiple such
// items, remove the one that getMinimum would return.)
public void removeMinimum()

// returns how many items the container has
public int size()

// returns an iterator that will list all the items in the container from
// small to large
public Iterator<T> iterator()

// returns an iterator that will list all the items in the container from
// large to small. more specifically, it should list it in the reversed order of
// iterator().
public Iterator<T> revIterator()
```

While Java has a built-in priority queue, you are not allowed to use it. This is for good reasons: the built-in priority queue doesn't have all the functionality that we require and extending from it is more difficult than creating a new implementation from scratch.

We recommend that you keep the items of your container in an `ArrayList<T>`, for example, by declaring an instance variable `private List<T> queueItems;`

More specifically, if `queueItems` is kept sorted, both iterators should be straightforward to support.

Ground Rules: The only file you can modify here (aside from creating/writing tests) is `MyPriorityQueue.java`.

Task 7: Midway Tower of Hanoi (4 points)

For this task, save your work in `smallman; 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.

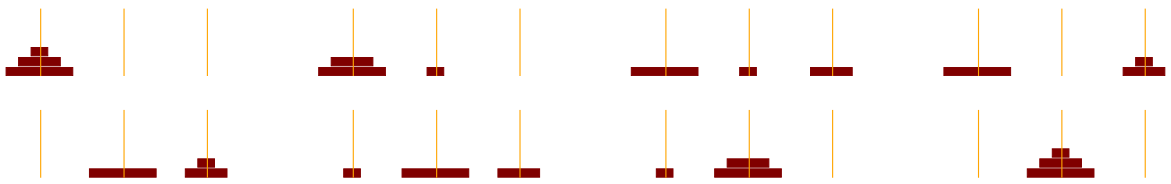
In Tower of Hanoi, you are given N disks, labeled $0, 1, \ldots, N-1$ by their sizes. In addition, there are 3 pegs: Peg 0, Peg 1, and Peg 2. Initially, all disks are at Peg 0, neatly arranged from small (Disk 0) to large (Disk $N-1$), with the smallest one at the top and the largest one at the bottom. 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:



Subtask 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.

Subtask 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).