CMPSCI 187 / Spring 2015
Hanoi

Due on Thursday, March 12, 2015, 8:30 a.m.

Marc Liberatore and John Ridgway

Morrill I N375
Section 01 @ 10:00
Section 02 @ 08:30

Contents

Overview

For this assignment, you will implement a few algorithms recursively. Then, you'll implement a generic collection. Finally, you'll implement both the Tower of Hanoi game and a solution generator for that game.

Learning Goals

- Practice implementing simple recursive methods, including mutually (indirectly) recursive methods.
- Show ability to write a generic class that implements a given interface, fulfilling the contract it specifies (including $O()$ behavior and an `Iterable` implementation).
- Model the Towers of Hanoi game and its solution using appropriate data structures and abstractions (e.g., stacks and lists).
- Write a recursive solver for the Towers of Hanoi game.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then email cs187help@cs.umass.edu immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` directory you export.
- The TAs and instructors are here to help you figure out errors, but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission.

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. But you should be aware that we deliberately don't provide you the full test suite we use when grading.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods taking integers as arguments handle positives, negatives, and zeroes (when those values are valid as input)?
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases, they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Note: It is expected that there will be three errors in the project as distributed, visible in the Problems tab in Eclipse. Each relates to the `ListImplementation` class, and all will be resolved once you've completed Problem 2.

Note: You may not use any of the classes from the Java Platform API that implement the Collection interface (or its sub-interfaces, etc.). The list of these classes is included in the API documentation, available at: http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html. Doing so will be viewed as an attempt to cheat.

Problem 1

Basic Recursion Warmup

Start off by completing the `algorithms.RecursiveMath` class you will find in the `src` directory. You must implement each method in this class using recursion.

Nowhere in this class may you use loops; i.e., you must not use **`for`**, **`while`**, or **`do`** statements. Your code may not cast values to other types – it must operate only on **`int`**s. Nor may you use the `java.util.Math` class. The private tests will be smart enough to detect violations of these requirements, and at best **you will receive a zero for the entire project** if you violate them.

`isEven` and `isOdd` For your implementation of the `isEven` and `isOdd` methods, you must not use the `%` operator. Your code must handle negative values. **Hint**: What does `isEven(0)` return? What does `isOdd(1)` return? What does `isEven(2)` return? You may want to use `isOdd` in your definition of `isEven` and vice versa.

`sumN` For your solution, you must not use the `*`, `/`, `*=`, or `/=` operators. **Hint**: Read about factorial at the end of Section 4.1. The solution to this one is pretty similar!

`factorial` Read about `factorial` at the end of Section 4.1.

`biPower` **Hint**: This one is also very similar to `factorial`!

Problem 2

It may be useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` folder. When we test your assignment, all files not included in the `src` folder will be ignored.

Implement `ListInterface`

Take a mental break from recursion and work on something you should be quite familiar with: lists! You will need to implement the `structures.ListInterface` provided. Make sure to read over the comments carefully; you are being tested on how well you implement the stated contract. You should implement the `ListInterface` in a class `ListImplementation`, and you should locate that class in the `structures` package under `src`.

Problem 2 continued on next page...

You will notice that the `size` and `append` methods must execute in $O(1)$ time. This means you will not pass the tests if you iterate down your links to calculate the size or to append an element. You will have to find another way to do this.

You are now adding a `get` method. The method should take $O(n)$ time where n is the index of the element being returned. One valid approach is to iterate to the specified location on your list and then return the element.

You are also adding an `iterator()` method, which means that you will need to implement a class that implements `Iterator` for your `iterator()` method to return.

Problem 3

It may be useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` folder. When we test your assignment, all files not included in the `src` folder will be ignored.

Code Overview

In the `support` directory you will find provided three classes, `HanoiMove`, `HanoiRing`, and `HanoiSolution`, that must not be modified. In addition there is an exception class `IllegalHanoiMoveException`, and two interfaces, `HanoiBoard`, and `HanoiPeg`.

Corresponding to the interfaces in the `support` directory are skeleton classes in the `src` directory for you to complete. `ArrayBasedHanoiBoard` implements the `HanoiBoard` interface, and `StackBasedHanoiPeg` implements the `HanoiPeg` interface. The `src` directory also contains a skeleton of the `RecursiveHanoiSolver` class. You must correctly complete all three of these classes.

We are also providing you with a `StackInterface` interface and a `LinkedStack` implementation of that interface. You must not change these but you are welcome to use them.

Implement the Tower of Hanoi Framework

In this section, you will be implementing a framework for the creating a Tower of Hanoi puzzle game. The Tower of Hanoi puzzle is presented in Section 4.3 of DJW.

Start by reviewing how the game works and then reading the interfaces and classes provided. Perhaps draw some pictures to show how each of the classes relates to one another before you start. This will help you become familiarized with their functionality.

We suggest you complete the skeleton classes in this order: `StackBasedHanoiPeg`, then `ArrayBasedHanoiBoard`, because it uses `StackBasedHanoiPeg`, and finally `RecursiveHanoiSolver`, which uses the other two. When we say "complete", we mean you should be as certain as possible that each is correct before moving on to the next — minimally, each should be passing all of the supplied tests before you move to the next.

`StackBasedHanoiPeg` Does this class look familiar in some way? Perhaps you can use some code we've provided as part of your implementation.

`ArrayBasedHanoiBoard` This class should be straightforward to finish. Imagine if you build an actual Tower of Hanoi puzzle. How would you interact with it in real life? The name of this class is a hint, but not a requirement, about how you might organize the group of three pegs.

`RecursiveHanoiSolver` In this class, you must implement a recursive algorithm to generate a solution to the Tower of Hanoi puzzle. You must not use loops in your implementation (and trust us, you don't want to try).

You should definitely read (and then re-read) Section 4.3 of the book. Make sure you understand how to solve the puzzle in and out. You will notice that the `solve` method actually returns an instance of `HanoiSolution`. This class is simply a wrapper containing a value representing how one should set up the Tower of Hanoi puzzle and then a list of moves to make to complete the puzzle.

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `hanoi-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`hanoi-student`**. Save the exported file with the `zip` extension (any name is fine). Log into Moodle and submit the exported zip file.