Tower of Hanoi Lab

AP Computer Science A

Name: _____

Contents

Background

In 1883, Édouard Lucas invented a mathematical logic puzzle known as the Tower of Hanoi. In this puzzle, the solver is asked to move a series of different sized disks from one of three pegs, or towers, to another of the pegs. An image of a typical version of this puzzle can be seen below, courtesy of the Wikimedia Commons repository.

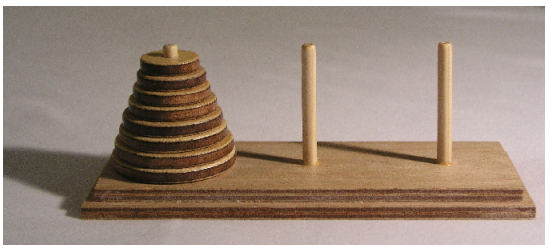

Image by Ævar Arnfjörð Bjarmason, Distributed Under CC Attribution Share-Alike 3.0

Although this might at first seem like a fairly simple task, the following set of rules makes the puzzle far more challenging:

1. Only one disk may be moved at a time.
2. This disk must be the top disk of one of the pegs/towers.
3. A larger disk cannot be placed on top of a smaller disk.

You are encouraged at this time to actually attempt this puzzle. If a physical copy of the Tower of Hanoi puzzle is not available, there have been multiple adaptations of it online and in mobile apps. A quick search for "online Tower of Hanoi" online or simply "Tower of Hanoi" in your app store of choice will turn one up that is free to play. Although the standard puzzle introduced by Lucas had eight disks, you should first try it with fewer disks (I suggest five or fewer). Hopefully, you will not only get a handle on how the rules interact with one another, but also be able to solve the puzzle a few times!

The Minimum Number of Moves

Lucas' invention came with a story of the Tower of Brahma. In this story, there exists a temple with three posts and 64 golden disks. According to the myth, Brahmin priests work day and night to transfer the 64 golden disks from one of the posts to the other using the rules above. Once their work is complete, the world will come to an end.

Although this stylized version of the Tower of Hanoi puzzle is interesting, the real question that comes out of it is: how long will it take the priests to move these 64 golden disks?

It turns out that the answer to the question of the *minimum* number of moves needed to solve the Tower of Hanoi puzzle has been well studied. The minimum number of moves necessary to solve an n-disk Tower of Hanoi puzzle is given by the formula:

$$2^n - 1$$

Note that the derivation of this formula is beyond the scope of this lab; however, it does give us a convenient test to ensure that our algorithms are working efficiently.

Examples

8-Disk Tower of Hanoi

Using the formula above, the minimum number of moves required to solve Lucas' original 8-disk puzzle is:

$$2^8 - 1 = 255$$

64-Disk Tower of Brahma

Use the formula above, the minimum number of moves required to solve the Tower of Brahma puzzle is:

$$2^{64} - 1 \approx 1.8 \times 10^{19}$$

How large is this number? Consider this: if the priests were able to move a single disk every second, twenty-four hours a day, it would take over $584,542,046,090$ *years* for them to complete the puzzle, far exceeding the estimated lifespan of the Universe.

Applications

Question #1: Why do you think the Tower of Hanoi puzzle has intrigued computer scientists for decades?

Question #2: Why is it "natural" to think of the solution to the Tower of Hanoi puzzle recursively?

Question #3: Using the notation: (source, destination), write every step you need to make to move all disks from the first tower to the third in a three-disk Tower of Hanoi puzzle.
Example: (0, 2) would indicate moving the top disk off the first tower and placing it on the third.

Question #4: Using the same notation as above, now write the steps required to solve a four-disk Tower of Hanoi puzzle; however, this time you also have access to the command: `solve3(source, destination)`, which will automatically move the top three disks off of `source` and place them on `destination`.

Activity #1

Introduction

In this activity, you will be creating a method that will help you visualize the moves you are making to solve of a Tower of Hanoi puzzle, as well as a method used to calculate the tower needed as a "buffer" when solving a given puzzle. Although neither of these methods are complex, they will prove to be an immense help in the solving of the actual Tower of Hanoi puzzle in Activity #2.

Exercises

1. Implement the `performMove()` method. This method should take the following parameters:
   - `puzzle` The Tower of Hanoi puzzle to perform the move on.
   - `from` The source tower to move a disk from.
   - `to` The destination tower to move a disk to.
   - `verbose` Whether or not the move should be output to the screen.

   In particular, you will have to manage the output to the screen. If `verbose` is `true`, your method should output: `Move #` followed by the current move number being performed, as well as a full print out of the current state of the given `puzzle`. If the move was unsuccessful, your method should output: `Illegal Move` instead. A `verbose` value of `false` should perform the move silently.
   Note: You should use the `move()` method built as part of the `HanoiPuzzle` class.
2. Implement the `calcOther()` method. This method should take as parameters the numbers of two towers in the Tower of Hanoi puzzle and return the number of the third, ungiven tower.
   Note: Although this can be easily accomplished with a series of `if` statements, I encourage you to attempt to find a mathematical formula that can calculate the number of the ungiven tower without a single conditiona.

Questions

Question #5: Why is it important to be able to "visually inspect" each move we make when solving the Tower of Hanoi puzzle?

Question #6: How did you discover the formula for `calcOther()`? If you did not discover a formula, still explain the steps you took in your attempts to find one.

Activity #2

Introduction

Exercises

1. Implement the `solve3()` method. This method should take the following parameters:

   - `puzzle` A 3-disk Tower of Hanoi puzzle to solve.
   - `from` The source tower all of the disks are starting on.
   - `to` The destination tower to move all the disks to.
   - `verbose` Whether or not to output all moves to the screen.

   It will then "manually" solve a 3-disk Tower of Hanoi puzzle.
   Hint: Use your solution to Question #3 in the Applications section as a guide.

2. Implement the `solveN()` method. This method will take the above parameters; however it will also take a parameter, `n`, representing how many disks should be moved from the destination tower to the source tower.
   Hint: Consider how knowing how to `solve3()` helped solving a four-disk puzzle in Question #4 in the Applications section and implement a recursive solution for `solveN()`.

Questions

Question #7: How did you modify your solution to Question #3 in the Applications section to account for a potentially variable source and destination tower?

Question #8: Did you use `solve3()` in your `solveN()` recursion? It turns out that you can use a 1-disk Tower of Hanoi puzzle as your recursion's base case. Explain how this works.

Final Analysis

Question #9: Since its invention, there have been iterative methods developed for solving the Tower of Hanoi puzzle with n disks. Nevertheless, it remains an important academic exercise in the study of recursive algorithms. Why do you think this is?

Question #10: Run the command: `solveN(puzzle, 1, 3, 20, true)`. Separately, run: `solveN(puzzle, 1, 3, 20, false)`. What do you notice happening? Why do you think this occurs?

Question #11: What challenges did you face in the implementation of `solveN()`? How did you overcome these challanges?

Question #12: What new programming techniques or knowledge did you learn as a result of this lab?

Template Class & Test Cases

```
/**
 * Tower of Hanoi Lab (Template Classes and Test Cases)
 * This is the template class and test cases for the Tower of Hanoi Lab.
 * Written for the Woodstock School in Mussoorie, Uttarakhand, India.
 *
 * @author Jeffrey Santos
 * @version 1.0
 */

import java.util.Stack;

public class TowerOfHanoi {
  public static void main(String[] args) {
    //  Tests of the performMove() method:
    HanoiPuzzle hp1 = new HanoiPuzzle(5);
    //   Output: Move #1:... followed by an appropriate view of the puzzle.
    performMove(hp1, 0, 2, true);
    //   Output: Should show the state of the puzzle after moving a disk from
    //           tower 0 to tower 1.
    performMove(hp1, 0, 1, false);
    System.out.println(hp1);
    //   Output: Illegal Move
    performMove(hp1, 0, 1, true);

    // Tests of the calcOther() method:
    System.out.println(calcOther(1, 3));    // Output: 2
    System.out.println(calcOther(2, 1));    // Output: 3
    System.out.println(calcOther(3, 2));    // Output: 1

    // Tests of the solve3() method:
    HanoiPuzzle hp2 = new HanoiPuzzle(3);
    HanoiPuzzle hp3 = new HanoiPuzzle(3);

    //  Output: Should show all appropriate moves to solve the Tower of Hanoi
    //          puzzle with three disks (in 15 moves)!
    solve3(hp2, 1, 3, true);

    //  Output: Should show only the final, solved position of the Tower of
    //          Hanoi puzzle with three disks.
    solve3(hp3, 1, 3, false);
    System.out.println(hp3);

    // Tests of the solveN() method:
    HanoiPuzzle hp4 = new HanoiPuzzle(5);
    HanoiPuzzle hp5 = new HanoiPuzzle(8);
    HanoiPuzzle hp6 = new HanoiPuzzle(12);

    //   Output: Every move required to solve a 5-disk Tower of Hanoi puzzle,
    //           followed by the correct number of moves (31).
    solveN(hp4, 1, 3, 5, true);
    System.out.println(hp4.getMoves());

    //   Output: A solved 8-disk Tower of Hanoi puzzle, followed by the correct
    //           number of moves (255).
    solveN(hp5, 1, 3, 8, false);
    System.out.println(hp5);
    System.out.println(hp5.getMoves());

    //   Output: A solved 12-disk Tower of Hanoi puzzle, followed by the correct
    //           number of moves (4095).
    solveN(hp6, 1, 3, 12, false);
    System.out.println(hp6);
    System.out.println(hp6.getMoves());
```

```
}

/**
 * Moves a disk in the given puzzle from the given source tower to the given
 * destination tower. When verbose is true, the following is output to screen.
 * If a move is successful:
 * "Move #" followed by the current move number, then the entire current
 * state of the given puzzle.
 * If a move is not successful:
 * "Illegal Move"
 *
 * @param puzzle The puzzle to perform the move on.
 * @param from The source tower to move the disk from.
 *  Precondition: from is a valid tower number (0, 1 or 2).
 * @param to The destination tower to move the disk to.
 *  Precondition: to is a valid tower number (0, 1, or 2).
 * @param verbose Whether or not the given move should be output to the screen.
 *
 * Postcondition: If the move was valid, puzzle should contain towers holding
 *                the appropriate disks.
 */
public static void performMove(HanoiPuzzle puzzle, int from, int to, boolean verbose) {
  //  To be implemented in Activity #1, Exercise 1
}

/**
 * Calculates what the "other" tower is when given a specific source and
 * destination tower.
 *
 * @param from The given source tower.
 *  Precondition: from is a valid tower number (0, 1 or 2).
 * @param to The given destination tower.
 *  Precondition: to is a valid tower number (0, 1, or 2).
 * @return  The tower not given as either the source or destination tower.
 */
public static int calcOther(int from, int to) {
  //  To be implemented in Activity #1, Exercise 2
  return 0;
}

/**
 * Solves a given three-disk tower of hanoi puzzle, moving all disks from the
 * given source tower to the given destination tower. When verbose is true,
 * this method will output all moves performed while solving the puzzle.
 *
 * @param puzzle The puzzle to solve.
 *  Precondition: puzzle is a 3-disk tower of hanoi puzzle with all disks
 *                currently on the "from" tower.
 * @param from The source tower to move the disks from.
 *  Precondition: from contains all disks in the current puzzle
 * @param to The destination tower to move the disks to.
 *  Precondition: to is a valid tower number (0, 1, or 2).
 * @param verbose Whether or not each move should be output to the screen.
 *
 * Postcondition: puzzle should hold a solved Tower of Hanoi puzzle and the
 *                moves counter variable should hold the value 15.
 */
public static void solve3(HanoiPuzzle puzzle, int from, int to, boolean verbose) {
  //  To be implemented in Activity #2, Exercise 1
}

/**
 * Solves a given n-disk tower of hanoi puzzle, moving all disks from the
 * given source tower to the given destination tower. When verbose is true,
 * this method will output all moves performed while solving the puzzle.
 *
```

```
   * @param puzzle The puzzle to solve.
   *  Precondition: puzzle is an n-disk tower of hanoi puzzle with all disks
   *                currently on the "from" tower.
   * @param from The source tower to move the disks from.
   *  Precondition: from contains all disks in the current puzzle
   * @param to The destination tower to move the disks to.
   *  Precondition: to is a valid tower number (0, 1, or 2).
   * @param n The number of disks to move from the destination tower to the
   *          source tower.
   *  Precondition: 0 < n <= puzzle.diskCount
   * @param verbose Whether or not each move should be output to the screen.
   *
   * Postcondition: puzzle should hold a solved Tower of Hanoi puzzle and the
   *                moves counter variable should hold the value (2^n - 1).
   */
  public static void solveN(HanoiPuzzle puzzle, int from, int to, int n, boolean verbose) {
    //  To be implemented in Activity #2, Exercise 2
  }
}

/**
 * The HanoiPuzzle class is the core class designed to represent the Tower
 * of Hanoi problem in Java. It is primarily responsible for enabling and
 * verifying moves between towers, as well as presenting a graphical display
 * of the current state of the towers.
 *
 * The HanoiPuzzle class here presents an opportunity for students to read
 * and understand how it works; however, it should not be modified for
 * the purposes of this lab.
 *
 * Note: This class has been hard-coded to handle only a three tower problem,
 * which is sufficient for all exercises and questions presented in the lab.
 */

class HanoiPuzzle {
  private Tower[] towers;           //  an array to store the towers
  private int diskCount, moves;     //  diskCount represents the number of
                                    //  disks present in the current puzzle,
                                    //  moves represents the current total
                                    //  number of moves taken

  /**
   * HanoiPuzzle constructor to create and store the towers and
   * initialize the instance variables for each object. Additionally,
   * this constructor sets up the first tower to hold all of the disks
   * appropriately.
   *
   * @param diskCount   The number of disks for the current puzzle.
   *  Precondition: diskCount > 0
   */
  public HanoiPuzzle(int diskCount) {
      this.diskCount = diskCount;
      moves = 0;

      towers = new Tower[3];
      towers[0] = new Tower();
      towers[1] = new Tower();
      towers[2] = new Tower();

      // Create and add all of the disks to the first tower. Because
      // the storage of disks is implemented as a stack, they are created
      // and added in reverse order (largest to smallest).
      for (int i = diskCount; i > 0; i--)
        towers[0].place(new Disk(i));
  }
```

```
/**
 * Accessor method for the moves instance variable.
 *
 * @return  The current number of moves taken.
 */
public int getMoves() {
    return moves;
}

/**
 * Attempts to move the top-most disk from the given source towers
 * to the given destination tower. If successful, the move count will
 * be updated as appropriate.
 *
 * @param from  The source tower for the move.
 *  Precondition: from is a valid tower number (0, 1, or 2).
 * @param to    The destination tower for the move.
 *  Precondition: to is a valid tower number (0, 1, or 2).
 * @return  Returns true if the move was successful, false otherwise.
 */
public boolean move(int from, int to) {
    Tower srcTower = towers[from];
    Tower dstTower = towers[to];

    Disk topDisk = srcTower.remove();
    if ((topDisk != null) && (dstTower.place(topDisk))) {
        moves++;
        return true;
    }
    srcTower.place(topDisk);
    return false;
}

/**
 * Override of the toString method to visually display the current state
 * of all three towers in the current tower of hanoi puzzle.
 *
 * @return  The string to be printed to visually display the current state
 *          of the current tower of hanoi puzzle.
 */
public String toString() {
    int towerWidth = 2 * diskCount - 1;

    // Set-up the tower headings using a String formatting method.
    String headerFormat = "%" + (towerWidth) + "c%" + (2 * towerWidth + 1) + "c%" + (2 * towe
    String towerString = String.format(headerFormat, '0', '1', '2');
    towerString += "\n";

    // Loop through each of the tower "rows".
    for (int i = diskCount - 1; i >= 0; i--) {
      towerString += towers[0].getRowString(i, towerWidth);
      towerString += "  ";
      towerString += towers[1].getRowString(i, towerWidth);
      towerString += "  ";
      towerString += towers[2].getRowString(i, towerWidth);
      towerString += "\n";
    }
    return towerString;
}

/**
 * The Tower class designed to store and process disks. It is also
 * responsible for formatting its own "row strings" referenced below.
 *
 * This class has been implemented as a nested class. Nested classes
 * allow for the creation of classes in Java as members of another class.
```

```
 * They are traditionally used to logically group classes together,
 * particularly classes that need to directly access member variables
 * of one another.
 */
class Tower {
  private Stack<Disk> disks;    // A stack of disks implemented using
                                // Java's built-in Stack class.

  /**
   * Tower class constructor to initialize the stack of disks.
   */
  public Tower() {
    disks = new Stack<Disk>();
  }

  /**
   * Method that pushes a disk to the top of the tower's disk stack,
   * provided it meets the requirement of being small than the top
   * disk on the stack.
   *
   * @param d The disk to place.
   *  Precondition: d is a disk in the Tower of Hanoi puzzle to which this
   *                tower belongs.
   * @return  Returns true if the disk move is valid, fale otherwise.
   */
  public boolean place(Disk d) {
    if ((disks.empty()) || (d.compareTo(disks.peek()) < 0)) {
      disks.push(d);
      return true;
    }
    return false;
  }

  /**
   * Removes and returns the disk at the top of the disk stack.
   *
   * @return  The disk at the top of the disk stack or null if no such
   *          disk exists.
   */
  public Disk remove() {
    return (disks.empty()) ? disks.pop() : null;
  }

  /**
   * Composes and returns a string to graphically represent the current state
   * of a given "row" of the tower.
   *
   * @param row Which row to represent as a string.
   *  Precondition: row is a valid row for the current Tower of Hanoi puzzle.
   * @param width How wide the row should be (for string padding reasons).
   *  Precondition: width > 0
   * @return  A string graphically representing the current state of the
   *          given "row" of the tower, padded to the given width.
   */
  public String getRowString(int row, int width) {
    int stringLength = 2 * width - 1;
    String rowString = "";

    // If there is no disk present, represent the blank "peg" as a line.
    String diskString = (disks.size() <= row) ? "|" : disks.get(row).toString();

    // Calculate the required padding on either side of the present disk.
    int paddingLength = (stringLength - diskString.length()) / 2;

    // Add padding to the left of the present disk.
    for (int i = 0; i < paddingLength; i++)
```

```
        rowString += " ";
      rowString += diskString;
      // Add padding to the right of the present disk.
      for (int i = 0; i < paddingLength; i++)
        rowString += " ";

      return rowString;
    }
  }

  /**
   * The Disk class used to represent individual disks within the Tower of
   * Hanoi puzzle. This class shows implementation of the Comparable interface
   * in order to allow for determining whether or not a move is allowed.
   */
  class Disk implements Comparable<Disk> {
    private final int size;   // size of the current disk
                              // disk is labelled final as there should not be
                              // any reason why the size should be able to
                              // change throughout the solving of a puzzle.

    /**
     * Disk class constructor to initialize the disk to the given size.
     *
     * @param size The size of the disk to be created.
     */
    public Disk(int size) {
      this.size = size;
    }

    /**
     * Compares the current disk to the given one in accordance to the
     * compareTo method described in the Comparable interface.
     *
     * @param disk The disk to compare this one to.
     *  Precondition: disk is a valid disk for the current puzzle.
     * @return  -1 if the current disk is smaller than the given disk, +1 if
     *          the current disk is larger than the given disk, 0 otherwise.
     */
    public int compareTo(Disk disk) {
      if (size < disk.size)
        return -1;
      else if (size > disk.size)
        return 1;
      return 0;
    }

    /**
     * Override of the toString() method in order to graphically represent the
     * current disk.
     *
     * @return  A string graphically representing the current disk.
     */
    public String toString() {
      String diskString = "";

      // (2 * size - 1) allows for always having an odd number of symbols
      // representing each disk, which allows for centering of the disk
      // graphically on each tower
      for (int i = 0; i < (2 * size - 1); i++)
        diskString += "=";
      return diskString;
    }
  }
}
```