WORKED EXAMPLE 13.2 Towers of Hanoi

The “Towers of Hanoi” puzzle has a board with three pegs and a stack of disks of decreasing size, initially on the first peg (see Figure 6).

The goal is to move all disks to the third peg. One disk can be moved at one time, from any peg to any other peg. You can place smaller disks only on top of larger ones, not the other way around.

Legend has it that a temple (presumably in Hanoi) contains such an assembly, with sixty-four golden disks, which the priests move in the prescribed fashion. When they have arranged all disks on the third peg, the world will come to an end.

Let us help out by writing a program that prints instructions for moving the disks.

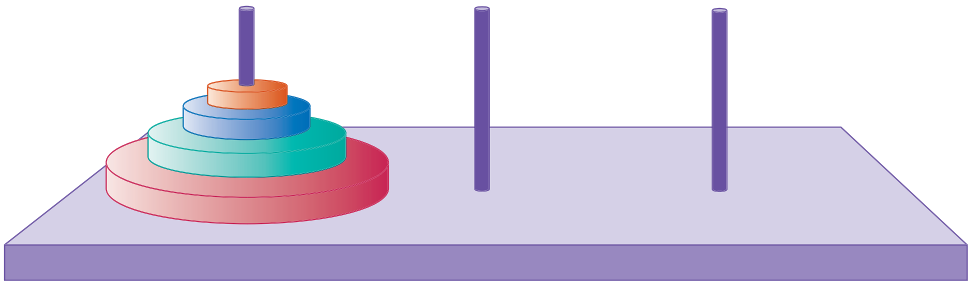

Figure 6 Towers of Hanoi

Consider the problem of moving d disks from peg p_1 to peg p_2, where p_1 and p_2 are 1, 2, or 3, and $p_1 \neq p_2$. Since $1 + 2 + 3 = 6$, we can get the index of the remaining peg as $p_3 = 6 - p_1 - p_2$.

Now we can move the disks as follows:

- Move the top $d - 1$ disks from p_1 to p_3
- Move one disk (the one on the bottom of the pile of d disks) from p_1 to p_2
- Move the $d - 1$ disks that were parked on p_3 to p_2

The first and third step need to be handled recursively, but because we move one fewer disk, the recursion will eventually terminate.

It is very straightforward to translate the algorithm into Java. For the second step, we simply print out the instruction for the priest, something like

```
Move disk from peg 1 to 3
```

worked_example_2/TowersOfHanoiInstructions.java

```
/**
   This program prints instructions for solving a Towers of Hanoi puzzle.
*/
public class TowersOfHanoiInstructions
{
   public static void main(String[] args)
   {
      move(5, 1, 3);
   }

   /**
      Print instructions for moving a pile of disks from one peg to another.
      @param disks the number of disks to move
      @param from the peg from which to move the disks
      @param to the peg to which to move the disks
   */
   public static void move(int disks, int from, int to)
   {
      if (disks > 0)
      {
         int other = 6 - from - to;
         move(disks - 1, from, other);
         System.out.println("Move disk from peg " + from + " to " + to);
         move(disks - 1, other, to);
      }
   }
}
```

Program Run

```
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
```

These instructions may suffice for the priests, but unfortunately it is not easy for us to see what is going on. Let's improve the program so that it actually carries out the instructions and shows the contents of the towers after each move.

We use a class `Tower` that manages the disks in one tower. Each disk is represented as an integer indicating its size from 1 to *n*, the number of disks in the puzzle.

We provide methods to remove the top disk, to add a disk to the top, and to show the contents of the tower as a list of disk sizes, for example, `[5, 4, 1]`.

```
public class Tower
{
   private ArrayList<Integer> disks;
   public Tower(int ndisks)
   {
      disks = new ArrayList<Integer>();
      for (int d = ndisks; d >= 1; d--) { disks.add(d); }
   }

   public int remove()
   {
      return disks.remove(disks.size() - 1);
   }

   public void add(int size)
   {
      if (disks.size() > 0 && disks.get(disks.size() - 1) < size)
      {
         throw new IllegalStateException("Disk is too large");
      }
      disks.add(size);
   }

   public String toString() { return disks.toString(); }
}
```

A TowerOfHanoi puzzle has three towers:

```
public class TowersOfHanoi
{
   private Tower[] towers;

   public TowersOfHanoi(int ndisks)
   {
      towers = new Tower[3];
      towers[0] = new Tower(ndisks);
      towers[1] = new Tower(0);
      towers[2] = new Tower(0);
   }
   . . .
}
```

Its move method first carries out the move, then prints the contents of the towers:

```
public void move(int disks, int from, int to)
{
   if (disks > 0)
   {
      int other = 3 - from - to;
      move(disks - 1, from, other);
      towers[to].add(towers[from].remove());
      System.out.println(Arrays.toString(towers));
      move(disks - 1, other, to);
   }
}
```

Here, we changed the index values to 0, 1, 2. Therefore, the index of the other peg is 3 - from - to.

Here is the main method:

```
public static void main(String[] args)
{
   final int NTOWERS = 5;
   TowersOfHanoi towers = new TowersOfHanoi(NTOWERS);
   towers.move(NTOWERS, 0, 2);
}
```

The program output is

```
[[5, 4, 3, 2], [], [1]]
[[5, 4, 3], [2], [1]]
[[5, 4, 3], [2, 1], []]
[[5, 4], [2, 1], [3]]
[[5, 4, 1], [2], [3]]
[[5, 4, 1], [], [3, 2]]
[[5, 4], [], [3, 2, 1]]
[[5], [4], [3, 2, 1]]
[[5], [4, 1], [3, 2]]
[[5, 2], [4, 1], [3]]
[[5, 2, 1], [4], [3]]
[[5, 2, 1], [4, 3], []]
[[5, 2], [4, 3], [1]]
[[5], [4, 3, 2], [1]]
[[5], [4, 3, 2, 1], []]
[[], [4, 3, 2, 1], [5]]
[[1], [4, 3, 2], [5]]
[[1], [4, 3], [5, 2]]
[[], [4, 3], [5, 2, 1]]
[[3], [4], [5, 2, 1]]
[[3], [4, 1], [5, 2]]
[[3, 2], [4, 1], [5]]
[[3, 2, 1], [4], [5]]
[[3, 2, 1], [], [5, 4]]
[[3, 2], [], [5, 4, 1]]
[[3], [2], [5, 4, 1]]
[[3], [2, 1], [5, 4]]
[[], [2, 1], [5, 4, 3]]
[[1], [2], [5, 4, 3]]
[[1], [], [5, 4, 3, 2]]
[[], [], [5, 4, 3, 2, 1]]
```

That's better. Now you can see how the disks move. You can check that all moves are legal—the disk size always decreases.

You can see that it takes $31 = 2^5 - 1$ moves to solve the puzzle for 5 disks. With 64 disks, it takes $2^{64} - 1 = 18446744073709551615$ moves. If the priests can move one disk per second, it takes about 585 billion years to finish the job. Because the earth is about 4.5 billion years old at the time this book is written, we don't have to worry too much whether the world will really come to an end when they are done.

worked_example_2/TowersOfHanoiDemo.java

```java
import java.util.ArrayList;
import java.util.Arrays;

/**
   This program shows a solution for a Towers of Hanoi puzzle.
*/
public class TowersOfHanoiDemo
{
   public static void main(String[] args)
   {
      final int NTOWERS = 5;
      TowersOfHanoi towers = new TowersOfHanoi(NTOWERS);
      towers.move(NTOWERS, 0, 2);
   }
}
```

worked_example_2/TowersOfHanoi.java

```java
import java.util.Arrays;

/**
   A Towers of Hanoi puzzle with three towers.
*/
public class TowersOfHanoi
{
   private Tower[] towers;

   /**
      Constructs a puzzle in which the first tower has a given number of disks.
      @param ndisks the number of disks
   */
   public TowersOfHanoi(int ndisks)
   {
      towers = new Tower[3];
      towers[0] = new Tower(ndisks);
      towers[1] = new Tower(0);
      towers[2] = new Tower(0);
   }

   /**
      Moves a pile of disks from one peg to another and displays the movement.
      @param disks the number of disks to move
      @param from the peg from which to move the disks
      @param to the peg to which to move the disks
   */
   public void move(int disks, int from, int to)
   {
      if (disks > 0)
      {
         int other = 3 - from - to;
         move(disks - 1, from, other);
         towers[to].add(towers[from].remove());
         System.out.println(Arrays.toString(towers));
         move(disks - 1, other, to);
      }
   }
}
```

worked_example_2/Tower.java

```java
import java.util.ArrayList;

/**
   A tower containing disks in the Towers of Hanoi puzzle.
*/
public class Tower
{
   private ArrayList<Integer> disks;

   /**
      Constructs a tower holding a given number of disks of decreasing size.
      @param ndisks the number of disks
   */
   public Tower(int ndisks)
   {
      disks = new ArrayList<Integer>();
      for (int d = ndisks; d >= 1; d--) { disks.add(d); }
   }

   /**
      Removes the top disk from this tower.
      @return the size of the removed disk
   */
   public int remove()
   {
      return disks.remove(disks.size() - 1);
   }

   /**
      Adds a disk to this tower.
      @param size the size of the disk to add
   */
   public void add(int size)
   {
      if (disks.size() > 0 && disks.get(disks.size() - 1) < size)
      {
         throw new IllegalStateException("Disk is too large");
      }
      disks.add(size);
   }

   public String toString() { return disks.toString(); }
}
```