Recursion

Goal

The exploration of recursion is continued in this assignment with a focus on double recursion. Two applications will be developed. The first is the Reve's problem, which is similar to the towers of Hanoi puzzle, and the second computes the maximum of an array.

Resources

- Chapter 9: Recursion
- *`Hanoi.jar`*—Sample application: Towers of Hanoi on four poles
- *`Reves.jar`*—Sample application: The Reve's puzzle
- Appendix —An Animation Framework

In `javadoc` directory

- *`Disk.html`*— Interface documentation for *`Disk.java`*
- *`Pole.html`*—Interface documentation for *`Pole.java`*

Java Files

- *`BadArgumentsForMaxException.java`*
- *`RecursiveMaxOfArray.java`*
- *`TestMax.java`*

In *`RevesApplication`* directory

- *`Disk.java`*
- *`Pole.java`*
- *`RevesActionThread.java`*
- *`RevesApplication.java`*

There are other files used to animate the application. For a full description, see Appendix: An Animation Framework.

Introduction

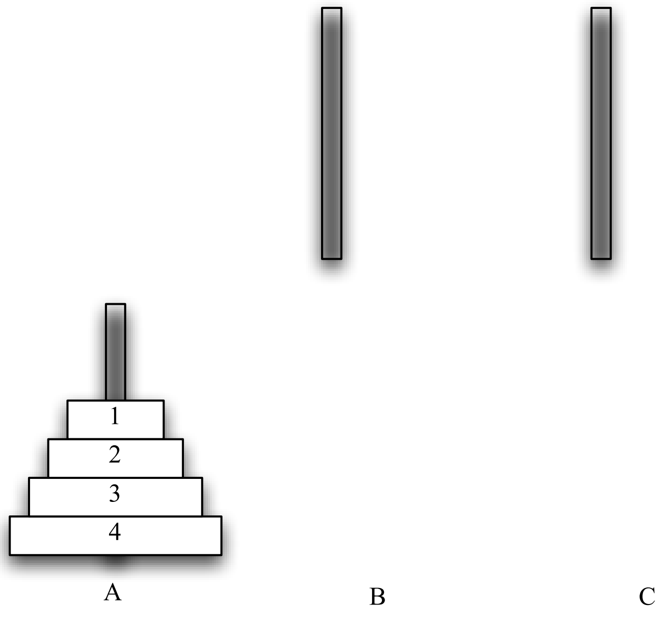
One of the classic examples of a doubly recursive algorithm is the solution to the towers of Hanoi. There are three poles and n disks of different sizes. The disks are stacked on one of the poles and are to be moved another pole. There are two basic rules. First, only one disk can be moved at a time. Second, no disk can be moved on top of a smaller disk.

Identify the problem:

Hanoi(n, from, to, extra)

Identify the smaller problems:

With four disks to be moved from pole A to pole C, the puzzle starts with the following picture.



If the disks from 1 through 3 could be moved off of pole A, the largest disk would be free to move. The stack of three disks must all be moved from A onto pole B, or C will not be available for disk 4. Once we move the largest disk, the remaining disks 1 through 3 must then be moved from pole B onto pole C.

There are two smaller problems to be solved.

```
Hanoi(n–1, from, extra, to) and
    Hanoi(n–1, extra, to, from)
```

Identify how the answers are composed:

```
Hanoi(n, from, to, extra) is
    1. Hanoi(n–1, from, extra, to)
    2. Move disk n off pole from onto pole to.
    3. Hanoi(n–1, extra, to, from)
```

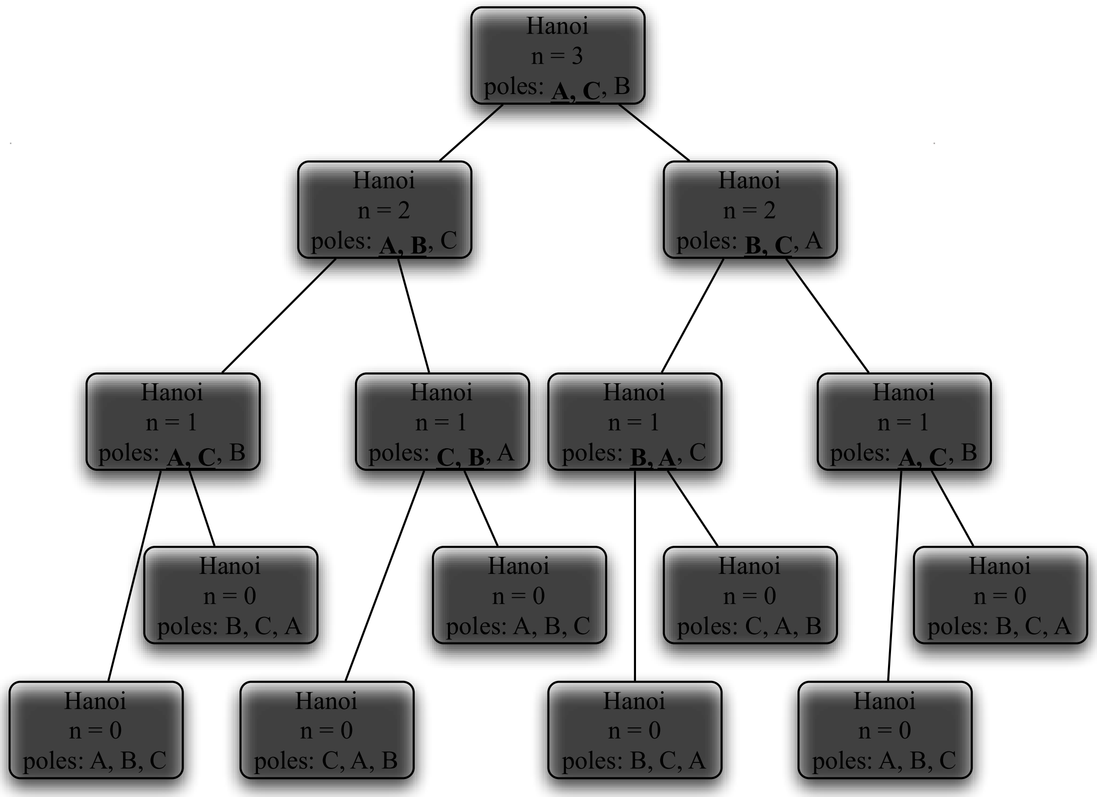
Identify the base cases:
There are two possibilities for the base case. Either n is 0 or n is 1. Certainly, if *n* is zero, nothing needs to be done. If *n* is 1, one disk needs to be moved. Either one will give a satisfactory base case.

```
Hanoi(0, from, to, extra) is
    1. Do nothing.
```

Compose the recursive definition:

```
Hanoi(n, from, to, extra) is
    If n =0
        1. Do nothing.
    If n > 0
        1. Hanoi(n–1, from, extra, to)
        2. Move disk n off pole from onto pole to.
        3. Hanoi(n–1, extra, to, from)
```

The tree for three disks is shown here.



The move made by each recursive call is indicated by underlining the poles involved in the move.

For the first application, you will do the Reve's puzzle. It is a game that is similar to the towers of Hanoi puzzle. The only difference is that there are four poles instead of three. For the towers of Hanoi, it is known that the recursive algorithm will move the disks optimally (in the least number of moves). Of the algorithms that solve the Reve's puzzle, it is not known which, if any, is optimal. There is a conjecture that the FrameStewart algorithm gives the optimum.

Visualization

The Reve's Puzzle

The Frame-Stewart algorithm for the Reve's puzzle will use the solution to the towers of Hanoi.

Suppose you need to move n disks from pole 1 to pole 4, with poles 2 and 3 as extras. If there is a single disk, Frame-Stewart will just move the disk, from pole 1 to pole 4. If there is more than one disk the algorithm has 3 steps. First, recursively move the top n–k disks from pole 1 to pole 2. Second, move the remaining k disks from pole 1 to 4, using the towers of Hanoi algorithm on just poles 1, 3, and 4. Last, recursively move the n–k disks from pole 2 to 4.

This is similar in structure to the towers of Hanoi, except that in the middle step more than one disk is moved.

The value of k is chosen to be the smallest integer value where n does not exceed k(k+1)/2 (the kth triangular number).

Fill in the following table with the values of the triangular numbers.

k	k (k +1)/2
1	1
2	3
3	6
4	10
5	15
6	21
7	28

For each n, find the k value such that n does not exceed the kth triangular number from the preceding table.

n	Smallest k where n where is not greater than k (k +1)/2
1	1
2	2
3	2
4	3
5	3
6	3
7	4
8	4
9	4
10	4
11	5

Give an algorithm for a method `computeK` which will find the value of k given n.

```
int computeK(int n) {
        int k = 1;
        int triNum = k * (k+1) / 2;
        while (n > triNum) {
                k++;
                int triNum = k * (k+1) / 2
        }
        // Once we find the smallest kth triangular number that n does not exceed, return k
        return k;
}
```

Using the method `computeK`, complete the recursive design for the Reve's puzzle.

BASICALLY, from = Pole 1, to = Pole 4, extra1 = Pole 2, extra2 = Pole 3

Identify the problem:

Reves(n, from, to, extra1, extra2)

Identify the smaller problems:

Reves(n-k, from, extra1, extra2, to)
Hanoi(k, from, to, extra2)
Reves(n-k, extra1, to, from, extra2)

Identify how the answers are composed:

Reves(n, from, to, extra1, extra2) is

1. **int k = computeK(n)**
2. **Reves(n-k, from, extra1, extra2, to)**
3. **Hanoi(k, from, to, extra2)**
4. **Reves(n-k, extra1, to, from, extra2)**

Identify the base cases:

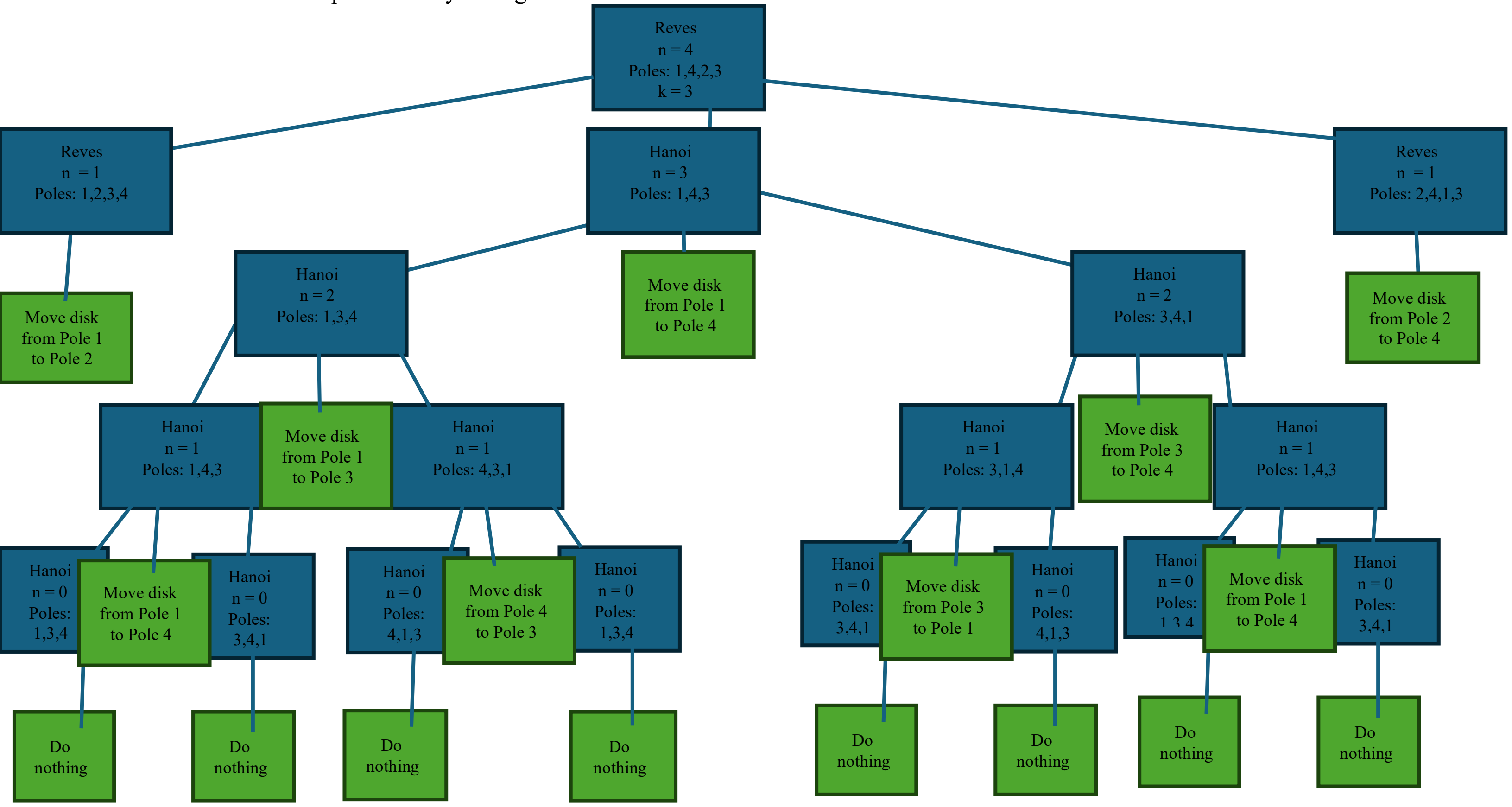
There is 1 base case. If n = 1, you simply move the disk that is on top of the "from" pole of that recursive method to the "to" pole of the recursive method. "from" is the first pole parameter, and "to" is the second.

```
Reves(1, from, to, extra1, extra2) is
            1. Move top disk in from pole to to pole
```

Compose the recursive definition:

```
Reves(n, from, to, extra1, extra2) is
if n == 1
     1.  Move top disk in “from” pole to “to” pole
if n > 1
          1.  int k = computeK(n)
          2.  Reves(n-k, from, extra1, extra2, to)
          3.  Hanoi(k, from, to, extra2)
          4.  Reves(n-k, extra1, to, from, extra2)
```

Trace the operation of your algorithm for n=4.   The solution should have nine moves.



Maximum

The second application will compute the maximum value in an array. It will split the array into halves and thus avoid an exponential performance cost. The pattern of this recursion is similar to what will be seen later for the advanced sorting algorithms.

As with the other recursive algorithms that work on an array, the recursion will be on a range of values that are being examined. This range will be split in half to get the subproblems.

To start consider how the split will be made. Suppose the recursive algorithm is asked to look at the portion of an array that ranges from 3 to 9.

INDEX	...	3	4	5	6	7	8	9	...
VALUE		20	40	10	90	50	70	80	

How many values are there in the range?

7 values

What is the index of the middle value in the range?

6

If the limits of the range are *first* and *last*, give a formula to compute the index of the middle value.

Mid = first + (last – first) / 2

The task now is to split up the array, but where should the middle value go? Potentially, it can go in the first half, the second half, or neither half. To answer this question, consider a portion of the array with just two entries. This is the smallest range that can be split up and is a useful test case. If this case does not work, the recursive algorithm is doomed to failure.

Using the preceding formula, what will be the index of the middle value?

The lower of the two indexes.

In the following arrays, box the left and right halves for the given way of handling the middle value.

Middle is in first half:

INDEX	...	3	4	...
VALUE		20	40	

Middle is in second half

INDEX	...	3	4	...
VALUE		20	40	

Middle is in neither half

INDEX	...	3	4	...
VALUE		20	40	

What is the maximum of an empty range? This is problematic. The maximum is not defined in that situation. Only one of the three cases above will have elements in both of the halves.

If the limits of the range are *first* and *last* and middle is the middle index, what are the ranges for the first and second halves so that both halves are nonempty?

First half range: (**first, middle)**

Second half range: (**middle + 1, last)**

Having thought about how to split the range in half, continue on with the recursive design.

Identify the problem:

Max(arr, first, last)

Identify the smaller problems:

Max(arr, first, mid)
Max(arr, mid+1, last)

Identify how the answers are composed:

```
Max(arr, first, last) is
    1. mid = first + (last – first) / 2 (rounded down)
    2. half1_max = Max(arr, first, mid)
    3. half2_max = Max(arr, mid+1, last)
    4. if half1_max > half2_max:
        1. return half1_max
    5. else (half2_max >= half1_max)
        1. return half2_max
```

Identify the base cases:

If first == last, that means there is just one element, so we return the element that corresponds to the index

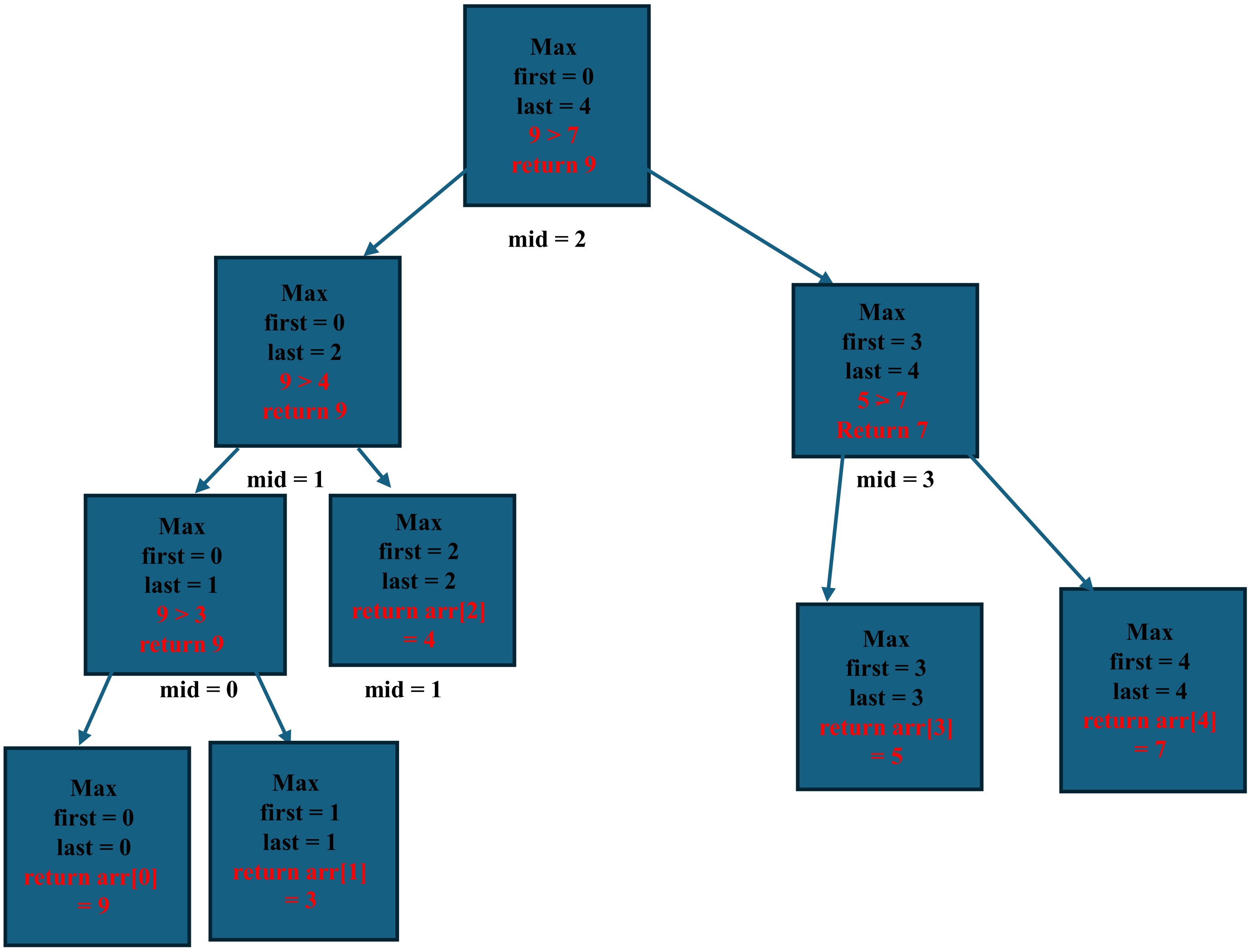
```
Max(arr, index, index) is
    1. return arr[first] (or arr[index])
```

Compose the recursive definition:

```
Max(arr, first, last) is
    1. if (first == last)
            a. return arr[first]
    2. if (first != last)
            a. mid = first + (last – first) / 2 (rounded down)
            b. half1_max = Max(arr, first, mid)
            c. half2_max = Max(arr, mid+1, last)
            d. if half1_max > half2_max:
                a. return half1_max
            e. else (half2_max >= half1_max)
                a. return half2_max
```

Show the operation of your definition on the array [9 3 4 5 7] with the range of 0 to 4 on the following diagram. Inside the boxes, show the values of the arguments passed into the method. On the left-hand side, show the operations done before the recursive call by the method. On the right-hand side, show operations done after the recursive call. There are two more recursive calls that will be made but are not shown in the picture. Their location will depend on how the split was formulated. Add them in.

PRE-CALL OPERATIONS ON BOTTOM
POST-CALL OPERATIONS/RETURN VALUES IN BOX



Directed Lab Work

Reve's Puzzle

In the first part of the assignment you will complete an animated application that solves the Reve's puzzle. Approximately half of the classes in the application implement the framework that animates the application. A description of the classes in the framework is given in Appendix: An Animation Framework. For the most part, these classes can be ignored as the animated application is developed as long as appropriate care is taken. The appendix also gives instructions for using the framework to create new animated applications.

Step 1. Compile all the classes in the folder `RevesApplication`. Run the `main` method in

`RevesApplication.java`.

Checkpoint: The program will run and a graphical user interface will appear. Across the top will be controls for stepping the application. At this point there are only two steps that can be done. Step from the setup phase to the initial state. Then step to the final state. At this point reset must be pressed. Do so. Pressing the go button will automatically step the application to the final state. The speed of the steps is controlled by the delay text field and can be changed at any time. The smaller the value of the delay, the quicker the steps will be. The default delay of 100 results in 1 second between steps. Pause can be pressed at any time and the application can be single stepped again.

In the setup phase, the number of disks can be changed. Enter a value into the text field and press enter. Once step is pressed and the application is in the initial state, the number of disks cannot be changed during the current execution of the application. (You can always press reset to abort the current execution and go back to the setup phase.)

The first goal is to create the disks and poles that the application will use.

Step 2. If you have not already, look at the interface documentation in *`Pole.html`* and *`Disk.html`*.

Step 3. In the `init()` method of `RevesActionThread`, add code to create four poles and assign them to the variables `a`, `b`, `c`, and `d`.

Step 4. Add code that creates the disks (use the variable `disks` as the limit for the loop) and puts them on pole a.

Checkpoint: Compile the classes and run RevesApplication.java. There should be four poles and 10 disks on the first pole. Type 4 in the text field for the number of disks and press enter. The number of disks displayed should change.

The next goal is to complete the code that will move a disk from one pole to another.

Step 5. Examine the method `moveDisk()` in `RevesActionThread`. It already has code that displays the move in the window and waits for a step. Don't change this code.

Step 6. Finish the `moveDisk()` method by adding code that will remove the top disk from one pole and then add it to the other pole.

Step 7. To test this, in the `executeApplication()` method of `RevesActionThread`, add the line of code

```
moveDisk(a,b);
```

Checkpoint: Compile the classes and run RevesApplication.java. Step three times. The top disk on pole a should move to pole b.

The next goal is to create the code that will solve the towers of Hanoi which will be used in the Reve's solution.

Step 8. Refer to the recursive design from the pre-lab exercises and create the method `towersOfHanoi()` in `RevesActionThread`. Make sure you call the `moveDisk()` method that was just completed.

Step 9. Change the single line of code in the `executeApplication()` method of `RevesActionThread` to call `towersOfHanoi()`. The source pole will be a, the destination will be d, and the extra will be b. Use the variable `disks` for the number of disks to move.

Checkpoint: Compile the classes and run RevesApplication.java. The application should move the disks as expected. You may run Hanoi.jar to see what it should look like.

The final goal is to complete the Frame-Stewart algorithm.

Step 10. Refer to the algorithm for finding the value of k and create a method to compute it in `RevesActionThread`.

Step 11. Refer to the recursive design from the pre-lab exercises and create the method `reves()` in `RevesActionThread`. Make sure to call the method that was just completed in the previous step.

Step 12. Change the single line of code in the `executeApplication()` method of `RevesActionThread to call reves()`. The source pole will be a, the destination will be d, and the extras will be b and c. Use the variable `disks` for the number of disks to move.

Final checkpoint: Compile the classes and run RevesApplication.java.

Try the application with three disks. It should use five moves.

Try the application with four disks. It should use nine moves.

The application should move the disks as expected. You may run to Reves.jar to see what it should look like.

Maximum

The second part of the lab is to complete the maximum application.

Step 1. Look at the skeleton in *`RecursiveMaxOfArray.java`*. Compile and run the `main` method in `TestMax`.

Checkpoint: The program should run and fail all tests.

Step 2. Refer to the recursive design from the pre-lab exercises. Complete the recursive method `max()`. Don't forget to throw an exception if there is not at least one value in the range.

Final checkpoint: Run TestMax. All tests should pass.