RECURSION

Chapter 5

Recursive Thinking

Section 5.1

Recursive Thinking

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means
- Recursion reduces a problem into one or more simpler versions of itself

Recursive Thinking (cont.)

Recursive Algorithm to Process Nested Figures

`if` there is one figure

    do whatever is required to the figure

`else`

    do whatever is required to the outer figure

    process the figures nested inside the outer figure in the same way

Recursive Thinking (cont.)

- Consider searching for a target value in an array

Recursive Thinking (cont.)

- Consider searching for a target value in an array

  Sound familiar??

  How did we do this?

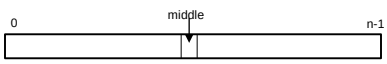
Recursive Thinking (cont.)

- Consider searching for a target value in an array
  - Assume the array elements are sorted in increasing order

  Does that change anything about how we search?

Recursive Thinking (cont.)

- Consider searching for a target value in an array
  - Assume the array elements are sorted in increasing order
  - We compare the target to the middle element and, if the middle element does not match the target, search either the elements before the middle element or the elements after the middle element
  - Instead of searching *n* elements, we search *n*/2 elements



Recursive Thinking (cont.)

Recursive Algorithm to Search an Array

`if` the array is empty

return -1 as the search result

`else if` the middle element matches the target

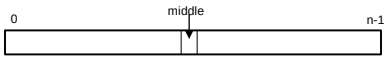
return the subscript of the middle element as the result

`else if` the target is less than the middle element

recursively search the array elements preceding the middle element and return the result

`else`

recursively search the array elements following the middle element and return the result



Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it/them directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Recursive Definitions of Mathematical Formulas

Section 5.2

Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms
- Examples include:
  - factorials
  - powers
  - greatest common divisors (gcd)

Factorial of *n*: *n*!

- The factorial of *n*, or *n*! is defined as follows:

  0! = 1

  n! = *n* x (*n* -1)! (n > 0)

- The base case: *n* is equal to 0
- The second formula is a recursive definition

Factorial of *n*: *n*! (cont.)

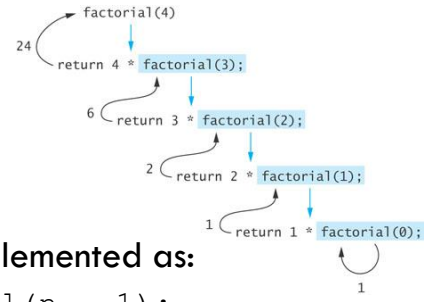
- The recursive definition can be expressed by the following algorithm:

  if *n* equals 0

  n! is 1

  else

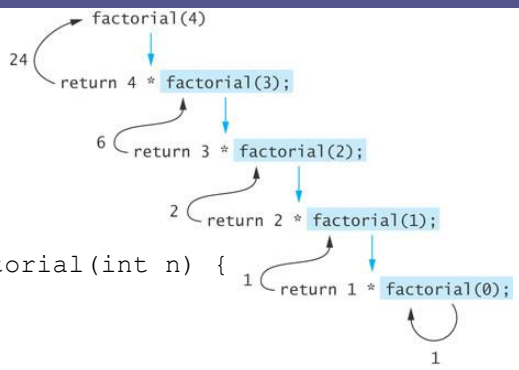
  n! = n x (n – 1)!



- The last step can be implemented as:

```
return n * factorial(n – 1);
```

Factorial of *n*: *n*! (cont.)

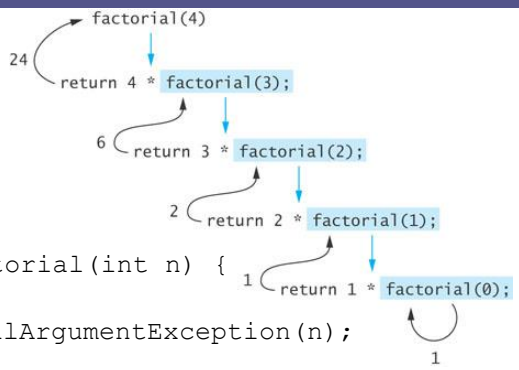



```
public static int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
} // factorial()
```

Infinite Recursion and Stack Overflow

- If you call method `factorial` with a negative argument, the recursion will not terminate because `n` will never equal `0`
- If a program does not terminate, it will eventually throw the `StackOverflowError` exception
- Make sure your recursive methods are constructed so that a stopping case is always reached
- In the `factorial` method, you could throw an `IllegalArgumentException` if `n` is negative

Factorial of *n*: *n*! (cont.)




```
public static int factorial(int n) {
   if (n < 0)
      throw new IllegalArgumentException(n);
   if (n == 0)
      return 1;
   else
      return n * factorial(n – 1);
} // factorial()
```

Recursive Algorithm for Calculating x^n

Recursive Algorithm for Calculating x^n ($n \geq 0$)

if *n is 0*

The result is 1

else

The result is $x \times x^{n-1}$

Recursive Algorithm for Calculating x^n

Recursive Algorithm for Calculating x^n ($n \geq 0$)

if *n is 0*

The result is 1

else

The result is $x \times x^{n-1}$

```
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n - 1);
} // power()
```

Recursive Algorithm for Calculating gcd

- The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers
- The gcd of 20 and 15 is 5
- The gcd of 36 and 24 is 12
- The gcd of 38 and 18 is 2
- The gcd of 17 and 97 is 1

Recursive Algorithm for Calculating gcd (cont.)

- Given 2 positive integers m and n (m > n)

  if n is a divisor of m

  gcd(m, n) = n

  else

  gcd (m, n) = gcd (n, m % n)

Recursive Algorithm for Calculating gcd (cont.)

```
public static double gcd(int m, int n) {
    if (m % n == 0)
        return n;
    else if (m < n)
        return gcd(n, m); // Transpose arguments.
    else
        return gcd(n, m % n);
} // gcd()
```

Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Iterative `factorial` Method

```
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
        result = result * k;
    return result;
} // factoriialIter()
```

Efficiency of Recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency usually does not outweigh the advantage of readable code that is easy to debug

Fibonacci Numbers

- Fibonacci numbers were used to model the growth of a rabbit colony

  $fib_1 = 1$

  $fib_2 = 1$

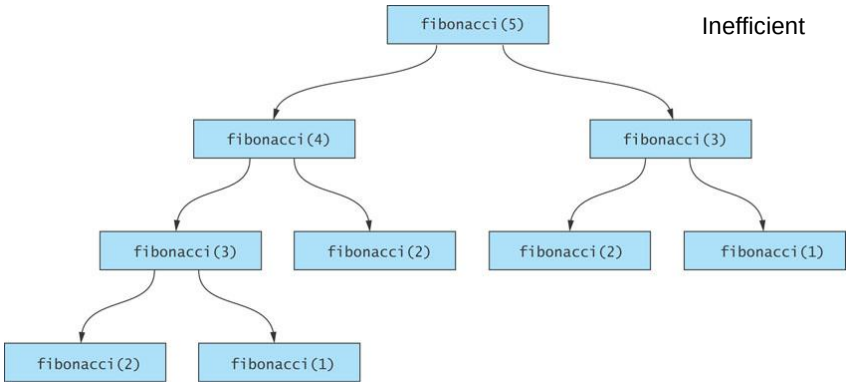
  $fib_n = fib_{n-1} + fib_{n-2}$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

An Exponential Recursive `fibonacci` Method

```
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Efficiency of Recursion: Exponential `fibonacci`



Inefficient

An O(n) Recursive `fibonacci` Method

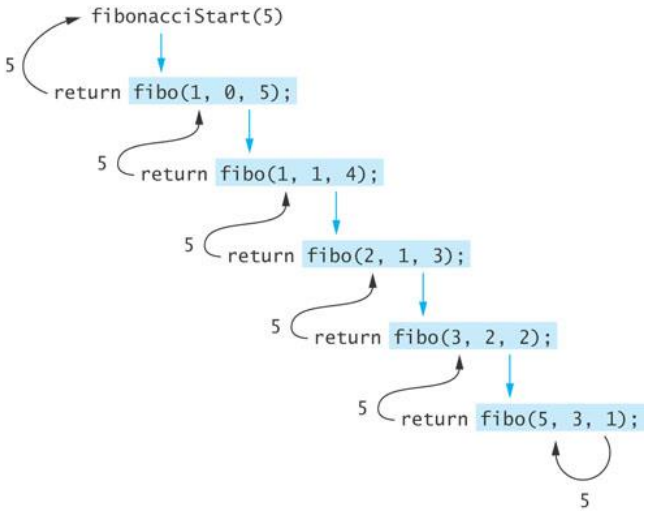
```
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
*/
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

An O(n) Recursive `fibonacci` Method (cont.)

- In order to start the method executing, we provide a non-recursive wrapper method:

```
/** Wrapper method for calculating Fibonacci numbers
   (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the desired Fibonacci
       number
    @return  The value of the nth Fibonacci number
*/
public static int fibonacciStart(int n) {
     return fibo(1, 0, n);
}
```

Efficiency of Recursion: O(n)
fibonacci



Efficient

Efficiency of Recursion: O(n)
fibonacci

- Method `fibo` is an example of *tail recursion* or *last-line recursion*
- When recursive call is the last line of the method, arguments and local variable do not need to be saved in the activation frame

Recursive Array Search

Section 5.3

Recursive Array Search

- Searching an array can be accomplished using recursion
- Simplest way to search is a linear search
  - Examine one element at a time starting with the first element and ending with the last
  - On average, $(n + 1)/2$ elements are examined to find the target in a linear search
  - If the target is not in the list, n elements are examined
- A linear search is $O(n)$

Recursive Array Search (cont.)

- Base cases for recursive search:
  - Empty array, target can not be found; result is -1
  - First element of the array being searched = target; result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

Algorithm for Recursive Linear Array Search

Algorithm for Recursive Linear Array Search

```
if the array is empty
    the result is –1
else if the first element matches the target
    the result is the subscript of the first element
else
    search the array excluding the first element and return the result
```

Implementation of Recursive Linear Search

```
/** Recursive linear search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The item being searched for
    @param posFirst The position of the current first element
    @return The subscript of target if found; otherwise -1
*/
private static int linearSearch(Object[] items,
                                Object target, int posFirst) {
    if (posFirst == items.length)
        return -1;
    else if (target.equals(items[posFirst]))
        return posFirst;
    else
        return linearSearch(items, target, posFirst + 1);
}
```

Implementation of Recursive Linear Search (cont.)

- A non-recursive wrapper method:

```
/** Wrapper for recursive linear search method
    @param items The array being searched
    @param target The object being searched for
    @return  The subscript of target if found;
             otherwise -1
*/
public static int linearSearch(Object[] items, Object target)
  {
    return linearSearch(items, target, 0);
}
```

Implementation of Recursive Linear Search (cont.)

Design of a Binary Search Algorithm

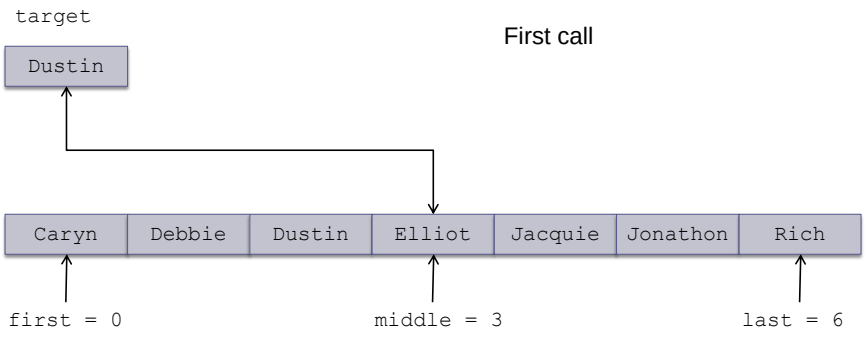
- A binary search can be performed only on an array that has been sorted
- Base cases
  - The array is empty
  - Element being examined matches the target
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- If the middle element does not match the target, a binary search excludes the half of the array within which the target cannot lie

Design of a Binary Search Algorithm (cont.)

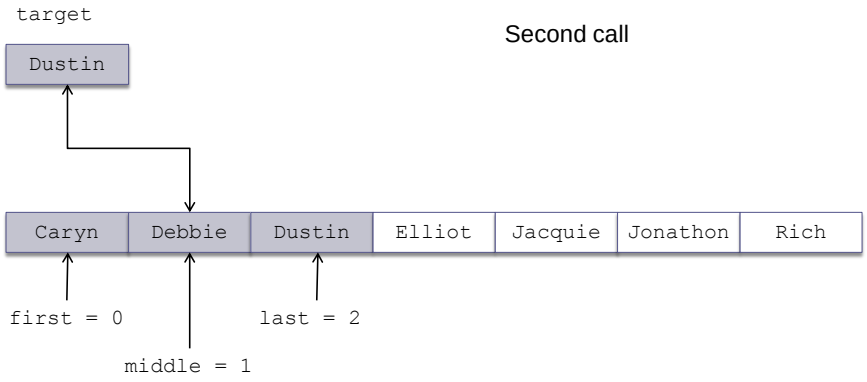
Binary Search Algorithm

`if` the array is empty
    return –1 as the search result
`else if` the middle element matches the target
    return the subscript of the middle element as the result
`else if` the target is less than the middle element
    recursively search the array elements before the middle element and return the result
`else`
    recursively search the array elements after the middle element and return the result

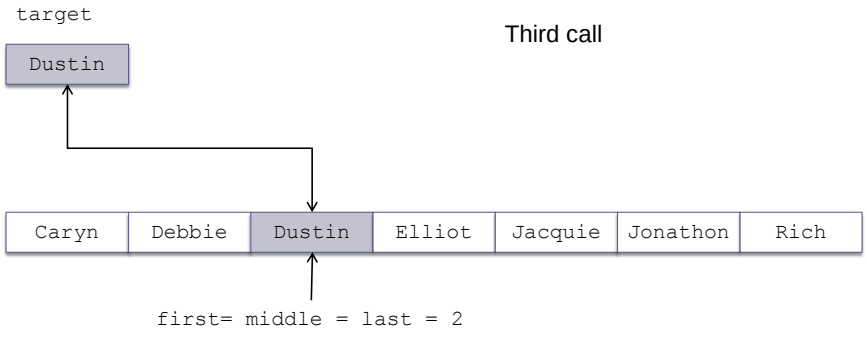
Binary Search Algorithm



Binary Search Algorithm (cont.)



Binary Search Algorithm (cont.)



Efficiency of Binary Search

- At each recursive call we eliminate half the array elements from consideration, making a binary search O(log *n*)
- An array of 16 would search arrays of length 16, 8, 4, 2, and 1: 5 probes in the worst case
  - $16 = 2^4$
  - $5 = \log_2 16 + 1$
- A doubled array size would require only 6 probes in the worst case
  - $32 = 2^5$
  - $6 = \log_2 32 + 1$
- An array with 32,768 elements requires only 16 probes! ($\log_2 32768 = 15$)

`Comparable` Interface

- Classes that implement the `Comparable` interface must define a `compareTo` method
- Method call `obj1.compareTo(obj2)` returns an integer with the following values
  - negative if `obj1 < obj2`
  - zero if `obj1 == obj2`
  - positive if `obj1 > obj2`
- Implementing the `Comparable` interface is an efficient way to compare objects during a search

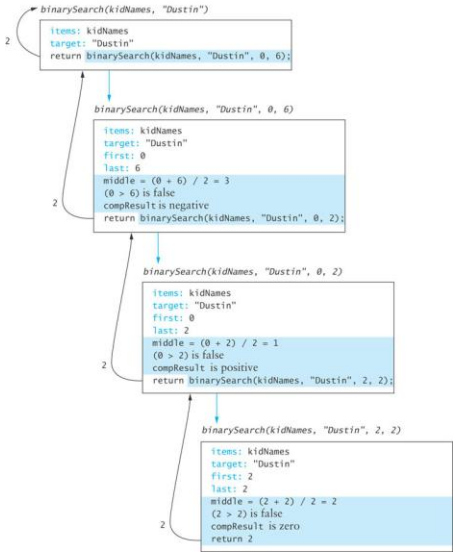
Implementation of a Binary Search Algorithm

```
/** Recursive binary search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @param first The subscript of the first element
    @param last The subscript of the last element
    @return The subscript of target if found; otherwise -1.
*/
private static int binarySearch(Object[] items, Comparable target,
                                int first, int last) {
    if (first > last)
        return -1;     // Base case for unsuccessful search.
    else {
        int middle = (first + last) / 2;  // Next probe index.
        int compResult = target.compareTo(items[middle]);
        if (compResult == 0)
            return middle;   // Base case for successful search.
        else if (compResult < 0)
            return binarySearch(items, target, first, middle - 1);
        else
            return binarySearch(items, target, middle + 1, last);
    }
}
```

Implementation of a Binary Search Algorithm (cont.)

```
/** Wrapper for recursive binary search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @return The subscript of target if found; otherwise -1.
*/
public static int binarySearch(Object[] items, Comparable target) {
    return binarySearch(items, target, 0, items.length - 1);
}
```

Trace of Binary Search



Method `Arrays.binarySearch`

- Java API class `Arrays` contains a `binarySearch` method
  - Called with sorted arrays of primitive types or with sorted arrays of objects
  - If the objects in the array are not mutually comparable or if the array is not sorted, the results are undefined
  - If there are multiple copies of the target value in the array, there is no guarantee which one will be found
  - Throws `ClassCastException` if the target is not comparable to the array elements

Problem Solving with Recursion

Section 5.5

Simplified Towers of Hanoi

- Move the three disks to a different peg, maintaining their order (largest disk on bottom, smallest on top, etc.)
  - Only the top disk on a peg can be moved to another peg
  - A larger disk cannot be placed on top of a smaller disk

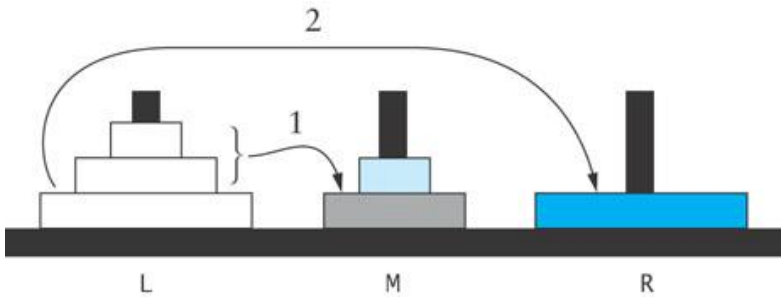
Towers of Hanoi

Problem Inputs
Number of disks (an integer)
Letter of starting peg: L (left), M (middle), or R (right)
Letter of destination peg: (L, M, or R), but different from starting peg
Letter of temporary peg: (L, M, or R), but different from starting peg and destination peg
Problem Outputs
A list of moves

Algorithm for Towers of Hanoi

Solution to Three-Disk Problem: Move Three Disks from Peg L to Peg R

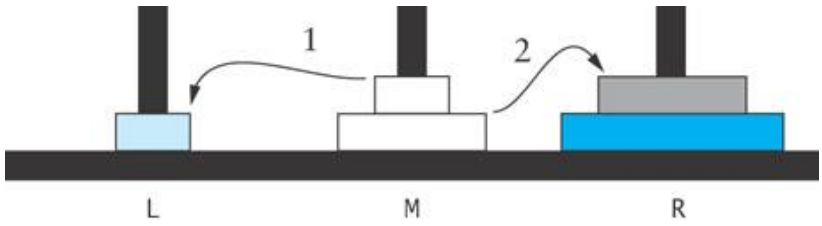
1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.



Algorithm for Towers of Hanoi (cont.)

Solution to Three-Disk Problem: Move Top Two Disks from Peg M to Peg R

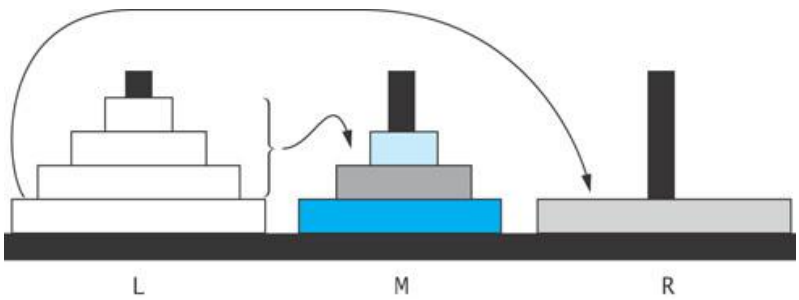
1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.



Algorithm for Towers of Hanoi (cont.)

Solution to Four-Disk Problem: Move Four Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.



Recursive Algorithm for Towers of Hanoi

Recursive Algorithm for n -Disk Problem: Move *n* Disks from the Starting Peg to the Destination Peg

`if` *n* is 1
    move disk 1 (the smallest disk) from the starting peg to the destination peg
`else`
    move the top *n* – 1 disks from the starting peg to the temporary peg (neither starting nor destination peg)
    move disk *n* (the disk at the bottom) from the starting peg to the destination peg
    move the top *n* – 1 disks from the temporary peg to the destination peg

Recursive Algorithm for Towers of Hanoi (cont.)

Method	Behavior
`public String showMoves(int n, char startPeg, char destPeg, char tempPeg)`	Builds a string containing all moves for a game with `n` disks on `startPeg` that will be moved to `destPeg` using `tempPeg` for temporary storage of disks being moved.

Implementation of Recursive Towers of Hanoi

```
/** Class that solves Towers of Hanoi problem. */
public class TowersOfHanoi {
    /** Recursive method for "moving" disks.
        pre: startPeg, destPeg, tempPeg are different.
        @param n is the number of disks
        @param startPeg is the starting peg
        @param destPeg is the destination peg
        @param tempPeg is the temporary peg
        @return A string with all the required disk moves
    */
    public static String showMoves(int n, char startPeg,
                                   char destPeg, char tempPeg) {
        if (n == 1) {
            return "Move disk 1 from peg " + startPeg +
                   " to peg " + destPeg + "\n";
        } else {  // Recursive step
            return showMoves(n - 1, startPeg, tempPeg, destPeg)
                   + "Move disk " + n + " from peg " + startPeg
                   + " to peg " + destPeg + "\n"
                   + showMoves(n - 1, tempPeg, destPeg, startPeg);
        }
    }
}
```