Unit 13 – Recursions

By Alexander Tilkin

Topics

- The Definition of Recursion
- The Components of the Recursion
- Time Complexity Analysis
- Recursions, Arrays, and Strings
- Drills

The Definition of Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.

Simple Recursion Example

```
public static void main(String[] args) {
      star();
}
```

```
public static void star() {
      System.out.print("*");

      star();
}
```

```
public static void star() {
      System.out.print("*");

      star();
}
```

```
public static void star() {
      System.out.print("*");

      star();
}
```

- This program is infinite, because one instance of the method star invokes a new instance
- The program will print a long line of '*' (more than 9,000, depends on the stack size)
- Eventually it will stop with the exception java.lang.StackOverflowError

Simple Recursion Example - 2

```
public static void main(String[] args) {
      star();
}
```

```
public static void star() {
      star();

      System.out.print("*");
}
```

```
public static void star() {
      star();

      System.out.print("*");
}
```

```
public static void star() {
      star();

      System.out.print("*");
}
```

- In this case there won't be an output. Because the invocation of the method star is placed above the print line
- The result will be many invocations of the method star and then the program will stop with the exception java.lang.StackOverflowError

How Do We Stop a Recursion?

- To stop a recursion we simply need to return from the method
- The return statement is wrapped with condition (depends on the implementation), we'll call it **Stop Condition**

```
public static void main(String[] args) {
        star(3);
}
```

```
public static void star(int starCounter) {
        if(starCounter == 0) {
                return;
        }

        System.out.println("*");

        star(starCounter - 1);
}
```

3

```
public static void star(int starCounter) {
        if(starCounter == 0) {
                return;
        }

        System.out.println("*");

        star(starCounter - 1);
}
```

2

```
public static void star(int starCounter) {
        if(starCounter == 0) {
                return;
        }

        System.out.println("*");

        star(starCounter - 1);
}
```

1

```
public static void star(int starCounter) {
        if(starCounter == 0) {
                return;
        }

        System.out.println("*");

        star(starCounter - 1);
}
```

0

Solving Problems Using Recursion

- Some problems can be solved by solving the same problem but on smaller size (for example smaller input)
- Each problem relies on the solution of the smaller problem
- Eventually we reach the simplest problem which is easy for us to solve
- In most cases all problems can be solved with loops (but not all of them)

Example: Calculating Factorial

- Factorial is the faction n! = 1*2*3*…*n (n>0)
- Another way to see it is n! = (n-1)!*n
- The simplest case in factorial is when n = 1, the solution is 1

```
public static void main(String[] args) {
      int result = factorial(3);
      System.out.println(result);
}

public static int factorial(int number) {
      if(number <= 1) {
            return 1;
      }

      return factorial(number - 1) * number;
}
```

Example: Calculating Factorial (breakdown)

```
public static void main(String[] args) {
    6   int result = factorial(3);
        System.out.println(result);
}
```

```
public static int factorial(int number) {
        if(number <= 1) {
                return 1;          3
        }

        return factorial(number - 1) * number;
}                    2
```

```
public static int factorial(int number) {
        if(number <= 1) {
                return 1;          2
        }

        return factorial(number - 1) * number;
}                    1
```

```
public static int factorial(int number) {
        if(number <= 1) {
                return 1;          1
        }

        return factorial(number - 1) * number;
}
```

The Components of the Recursion

- For writing a recursion three components are required:
  - Stop condition
  - Recursive invocation
  - Link between the problem and it's simpler case

```
public static int factorial(int number) {
     if(number <= 1) {
          return 1;
     }

     return factorial(number - 1) * number;
}
```

Stop Condition

Recursive Invocation

Link between the problems

Recursions – Time Complexity Analysis

- We might want to calculate the time complexity of a recursion
- The time complexity of a recursion is calculated as follows: (the time complexity of the content of the method) * (number of recursive invocations)

Factorial – Time Complexity Analysis

- The method *factorial* has basic functionality which take O(1) in total
- For the value of 3 we'll have 3 recursive invocations, and for n we'll have n recursive invocations
- Hence, the time complexity of the *factorial* is O(1)*O(n) = O(n)

Triangle

This is a tail recursion because the recursive invocation is placed in the top of the method

```
public static void main(String[] args) {
        triangle(3);
}
```

```
private static void triangle(int size) {
        if(size == 0) {
                return;
        }

        triangle(size - 1);

        for (int index = 0; index < size; index++) {
                System.out.print("*");
        }

        System.out.println();
}
```

3

***

```
private static void triangle(int size) {
        if(size == 0) {
                return;
        }

        triangle(size - 1);

        for (int index = 0; index < size; index++) {
                System.out.print("*");
        }

        System.out.println();
}
```

1

*

```
private static void triangle(int size) {
        if(size == 0) {
                return;
        }

        triangle(size - 1);

        for (int index = 0; index < size; index++) {
                System.out.print("*");
        }

        System.out.println();
}
```

2

**

```
private static void triangle(int size) {
        if(size == 0) {
                return;
        }

        triangle(size - 1);

        for (int index = 0; index < size; index++) {
                System.out.print("*");
        }

        System.out.println();
}
```

0

<u>Time Complexity Analysis</u>:

- The complexity of the code in the method is O(n) (loop)
- For the value of 3 we have four invocations, for n we'll have n+1 = O(n)
- O(n)*O(n) = O(n^2)

The result ->

```
*
**
***
```

Triangle – Tail Recursion

```
public static void main(String[] args) {
      triangle(1, 3);
}
```

```
private static void triangle(int currentLength, int size) {
      if(currentLength > size) {
      return;
      }

      for (int index = 0; index < currentLength; index++) {
            System.out.print("*");
      }

      System.out.println();

      triangle(currentLength + 1, size);
}
```

Why do we care?
The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

Calculation of Power

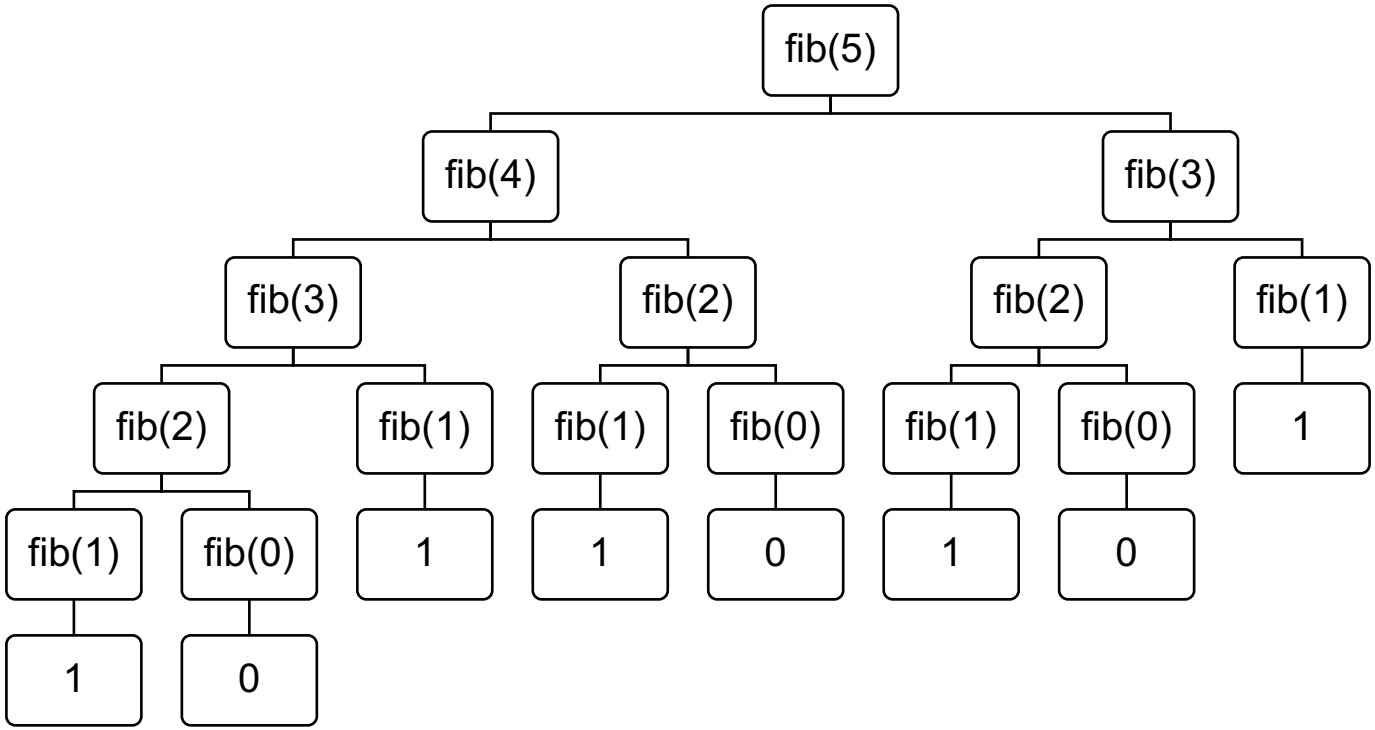
- Power is a function with two parameters: base (n) and power (k): n^k
- Or it can be represented as n^k = n^(k-1)*n
- The simplest case is when k = 0 -> n^0 = 1

```
public static void main(String[] args) {
      int result = power(2, 3);

      System.out.println(result);
}


private static int power(int base, int power) {
      if(power == 0) {
            return 1;
      }

      return power(base, power - 1) * base;
}
```

Time Complexity Analysis:

- The complexity of the code in thee method is O(1)
- The method is executed as the value of power (n) = O(n)
- Total complexity is O(1) * O(n) = O(n)

Fibonacci Sequence - Breakdown



Fibonacci Sequence - Code

- Fibonacci Sequence is represented as followas: f(2) = f(1) + f(0)
- In general f(n) = f(n-1) + (n-2)
- The simplest case is when n <= 1, f = 1

```
public static void main(String[] args) {
      int result = fibonacciSequence(4);

      System.out.println(result);
}


private static int fibonacciSequence(int power) {
      if(power <= 1) {
            return 1;
      }

      return fibonacciSequence(power - 1) + fibonacciSequence(power - 2);
}
```

Fibonacci Sequence – Time Complexity Analysis

First, we'll calculate lower bound.
$F(n) = F(n-1) + F(n-2)$

$T(n) = T(n-1) + T(n-2) + 4$ // 4 is the number of atomic operations we have in the Fibonacci equation
$T(0) = T(1) = 1$

Assumption: $T(n-1) \approx T(n-2)$
$T(n) = 2T(n-2) + c$

$= 2\{2T(n-4) + c\} + c$ // $T(n-2) = 2T(n-4) + c$ because $T(n) = 2T(n-2) + c$

$= 4T(n-4) + 3c$ // $\mathrm{k} = 2$

$= 8T(n-6) + 7c$ // $\mathrm{k} = 3$

$= 16T(n-8) + 15c$ // $\mathrm{k} = 4$
$T(n) = 2^k T(n-2k) + \left(2^k - 1\right)c$

$n - 2k = 0 \Rightarrow k = \frac{n}{2}$ // Because we are looking for the lower bound
$T(n) = 2^{\frac{n}{2}} T(0) + \left(2^{\frac{n}{2}} - 2\right) C = (1+c)2^{\frac{n}{2}} - c \Rightarrow T(n) = 2^{\frac{n}{2}}$

Fibonacci Sequence – Time Complexity Analysis

Now we'll calculate upper bound.

Assumption: $T(n-2) \approx T(n-1)$
$T(n) = 2T(n-1) + c$
$= 2\{2T(n-2) + c\} + c$

$= 4T(n-2) + 3c$ // k = 2

$= 8T(n-3) + 7c$ // k =3
$T(n) = 2^k T(n-k) + \left(2^k - 1\right)c$

$n - k = 0 \Rightarrow k = n$ // Because we are looking for the lower bound
$T(n) = 2^n T(0) + (2^n - 2)C = (1 + c)2^n - c \Rightarrow T(n) = 2^n$

Conclusion, the time complexity of recursive Fibonacci Sequence is $O(2^n)$

Fibonacci Sequence – Iterative Recursive Comparison

```
public int fibonacciIterative(int n) {
      if(n <= 1) {
            return n;
      }

      int fib = 1;
      int prevFib = 1;
      for(int i=2; i<n; i++) {
            int temp = fib;
            fib+= prevFib;
            prevFib = temp;
      }

      return fib;
}
```

Time complexity of the iterative solution is $O(n)$
Time complexity of recursive Fibonacci Sequence is $O(2^n)$

Conclusion, the iterative solution is faster

Recursions and Arrays

- Recursions can help us also with solutions where arrays are involved
- The idea: the stop condition is when the size of the array is 1 or 0 and we need to know what is the solution for this size
- The recursion knows the solution for the simplest case
- All we must do is to link between the simple case and the more complex

Calculating the Sum of Elements

- The sum of all elements in array is $\sum_{i=0}^{n-1} array[i]$
- Another way to see it is $\sum_{i=0}^{n-1} array[i] = \sum_{i=0}^{n-2} array[i] + array[n-1]$
- The simplest case is when n=0 -> 0

```
public static void main(String[] args) {
     int[] array = new int[]{1, 2, 3, 4};
     int result = sumOfElements(array);
     System.out.println(result);
}

private static int sumOfElements(int[] array) {
     if(array.length == 0) {
          return 0;
     }

     int newSize = array.length - 1;
     int[] newArray = Arrays.copyOf(array, newSize);

     return sumOfElements(newArray) + array[newSize];
}
```

Increases space complexity to $O(n!\,m)$

m - # of method invocations

$n!$ – each time we allocate n-1 cells on the heap

Calculating the Sum of Elements - Optimized

```
public static void main(String[] args) {
      int[] array = new int[]{1, 2, 3, 4};

      int result = sumOfElements(array, array.length);

      System.out.println(result);
}

private static int sumOfElements(int[] array, int length) {
      if(array.length == 0) {
            return 0;
      }

      int newLength = array.length - 1;

      return sumOfElements(newArray, newLength) + array[newLength];
}
```

- Now time Complexity is O(n) and Space Complexity is O(n), because we have n invocations of the method sumOfElements
- The time complexity of the iterative solution is O(n) and the Space complexity is O(1) reducing the space of the array
- Conclusion, the iterative solution is more efficient

Finding the Maximum

- max(array, length) = array[length - 1] > max(array, length - 1) ? array[length - 1] : max(array, length - 1)
- The simplest solution is when n=1: array[0]

```
public static void main(String[] args) {
      int[] array = new int[] {4, 7, 2, 8, 4, 7};

      int result = findMaximum(array, array.length);

      System.out.println(result);
}


private static int findMaximum(int[] array, int length) {
      if(length == 0) {
            return array[0];
      }

      int newLength = length - 1;

      int localMaximum = findMaximum(array, newLength);

      return localMaximum > array[newLength] ? localMaximum : array[newLength];
}
```

Is the Array Symmetric?

```
public static void main(String[] args) {
	int[] array = new int[] {1, 2, 3, 4, 3, 2, 1};

	boolean result = isSymmetric(array, 0, array.length - 1);

	System.out.println(result);
}


private static boolean isSymmetric(int[] array, int leftPivot, int rightPivot) {
	if(leftPivot >= rightPivot) {
		return true;
	}

	return array[leftPivot] == array[rightPivot] ? isSymmetric(array, leftPivot + 1, rightPivot -1) : false;
}
```

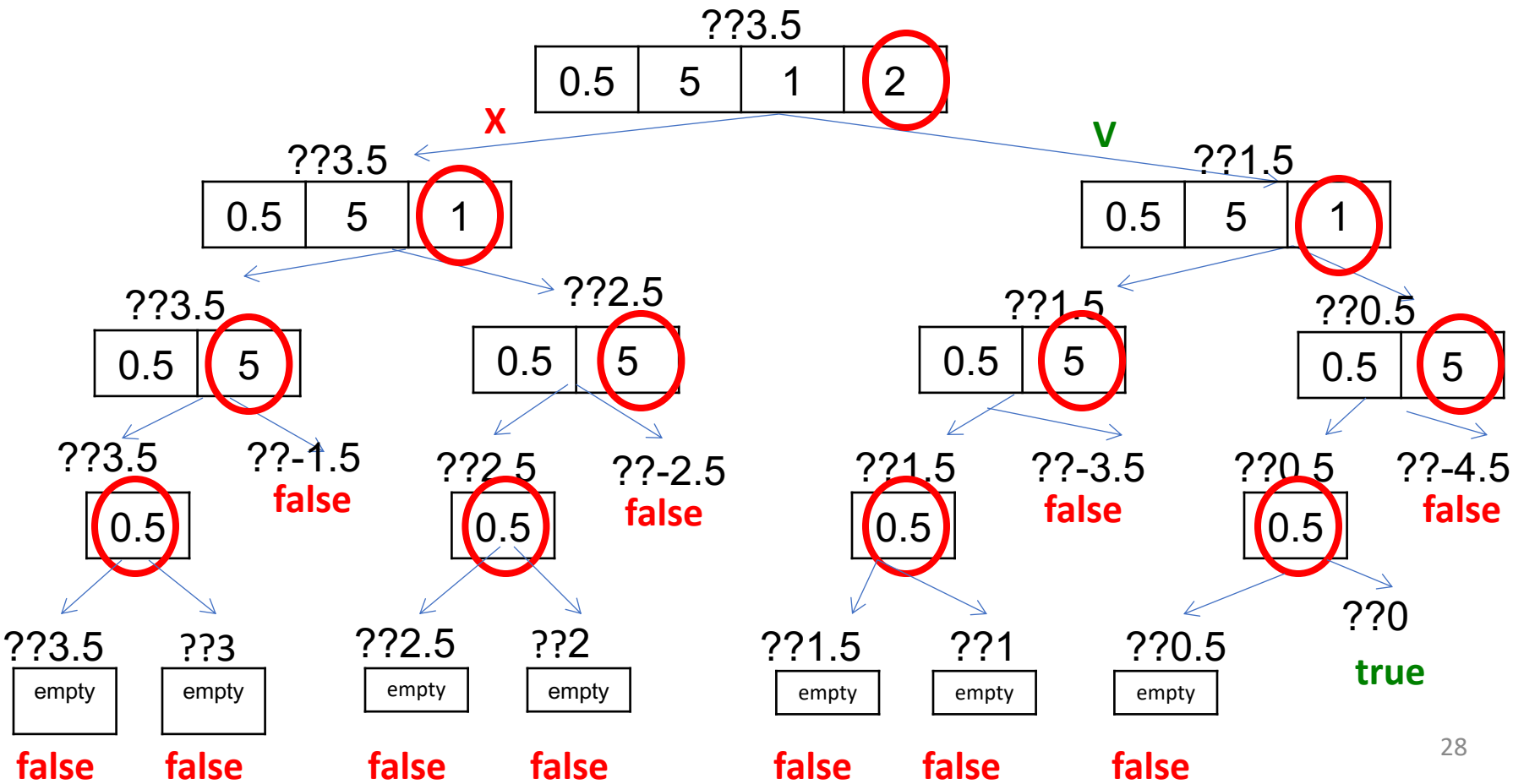
The Subset Problem

- You are given an array, it's length, and a number
- Find a solution that return True in case the array contains subset of elements which its sum is the number, otherwise it returns False
- Examples:
  - {1, 3, 5} and the number is 6 we return True because (1+5=6)
  - {1, 3, 4} and the number 9 we return True because 1+3+5+9
  - {1, 3, 5} and the number 7 we return False

The Subset Problem – The Idea

- Each element is whether part of the sum or not
- The smaller problem is the rest of the array without the removed element:
  - In case the element is part of the sum then we'll look for the rest of the elements that part of the sum, when the last element is subtracted from the sum
  - If the element is not part of the sum, we'll look the rest of the elements and we'll not touch the value of the sum
- The Simplest Problem:
  - If the size of the array is 0 and the value of the number is 0, we return True
  - If the size of the array 0 and the value of the number is != 0, we return False

Subset - Breakdown



Subset Problem - Code

```
public static void main(String[] args) {
      int[] array = new int[] {1, 2, 3, 4, 5};

      boolean result = findSubSet(array, array.length, 6);

      System.out.println(result);
}


private static boolean findSubSet(int[] array, int length, int value) {
      if(length == 0 && value == 0) {
            return true;
      }

      if(length == 0 && value != 0) {
            return false;
      }

      return findSubSet(array, length - 1, value) || findSubSet(array, length - 1, value -
      array[length - 1]);
}
```

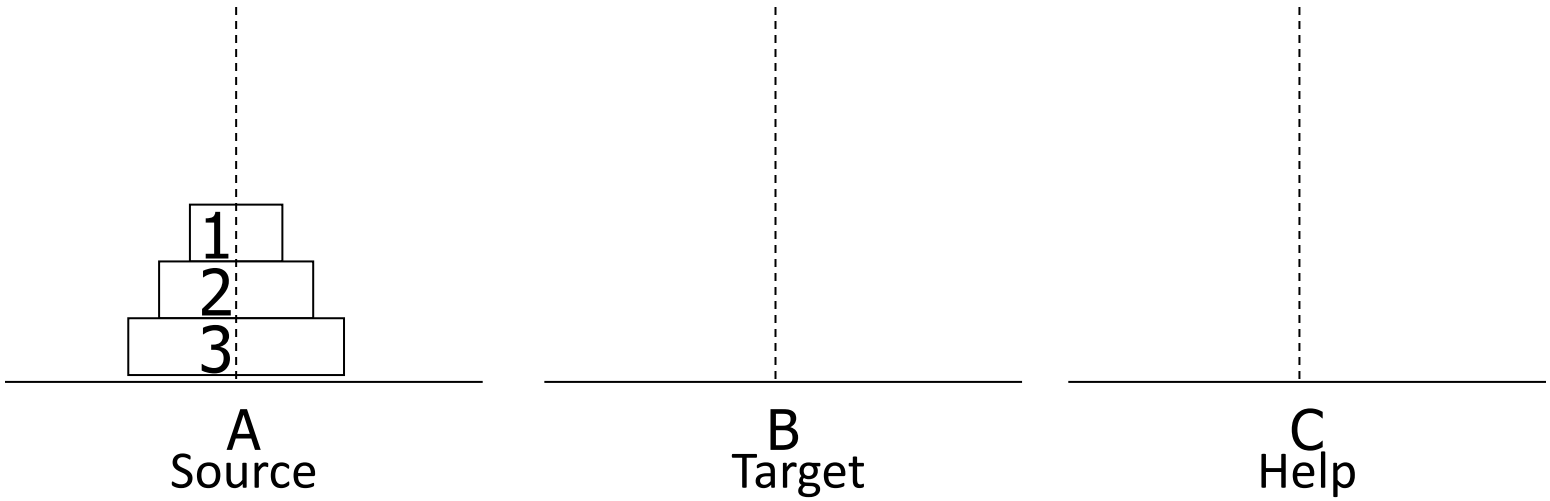
Number of Files in Folder

```
public static void main(String[] args) {
      File file = new File(".");

      int numberOfFiles = countFiles(file);

      System.out.println("This directory has " + numberOfFiles + " files");
}


private static int countFiles(File file) {
      if(file.isFile()) {
            return 1;
      }

      int sum = 0;
      File[] subFiles = file.listFiles();
      for (File subFile : subFiles) {
            sum += countFiles(subFile);
      }

      return sum;
}
```

Recursive Binary Search

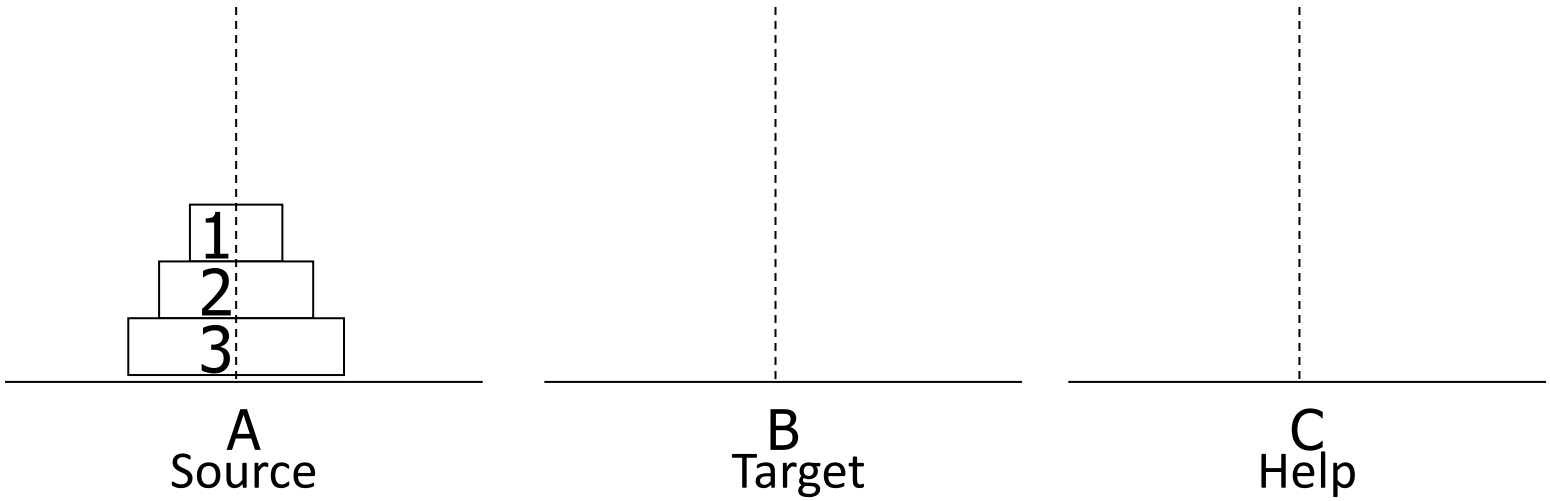
```
public static void main(String[] args) {
      int[] array = new int[] {1, 2, 5, 123, 743, 8324};

      int index = recursiveBinarySearch(array, 0, array.length - 1, 123);

      System.out.println(index);
}


private static int recursiveBinarySearch(int[] array, int lowBound, int highBound, int value) {
      if(lowBound > highBound) {
            return -1;
      }

      int middle = (highBound + lowBound) / 2;
      if(value == array[middle]) {
            return middle;
      }

      if(value < array[middle]){
            return recursiveBinarySearch(array, lowBound, middle - 1, value);
      }else {
            return recursiveBinarySearch(array, middle + 1, highBound, value);
      }
}
```

Hanoi Towers

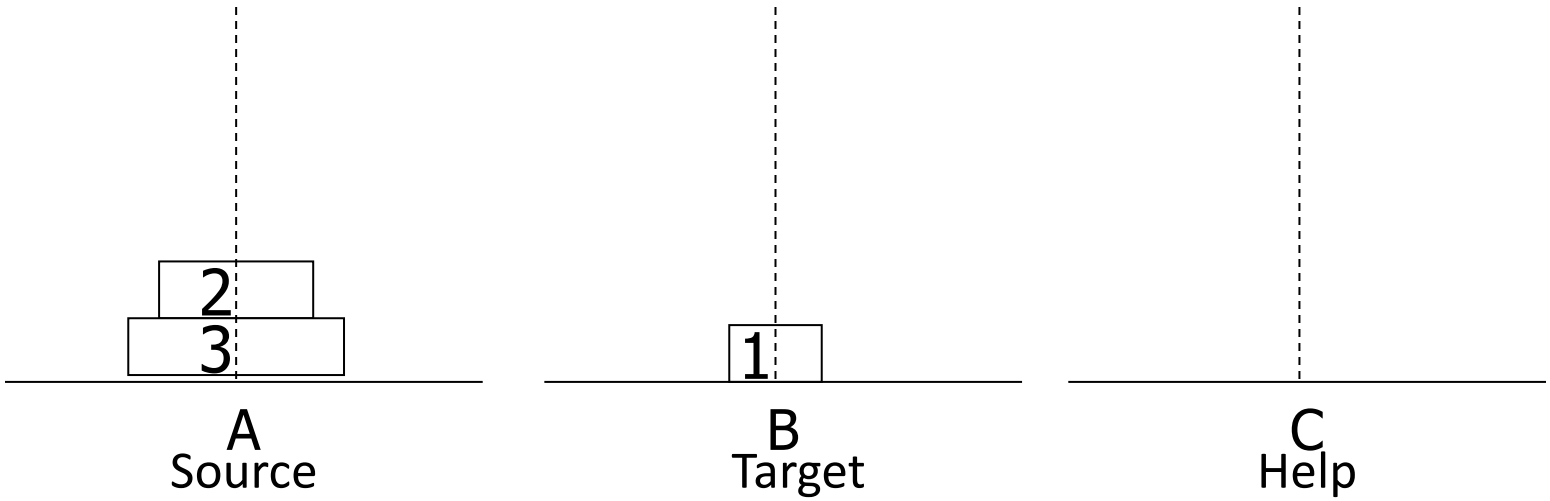


- The target is to transfer all disks from pillar A to pillar B according to the following rules:
  - Each step you can transfer one disk
  - You can’t place a large disk over a small disk
  - The disk must be place on one of the three pillars

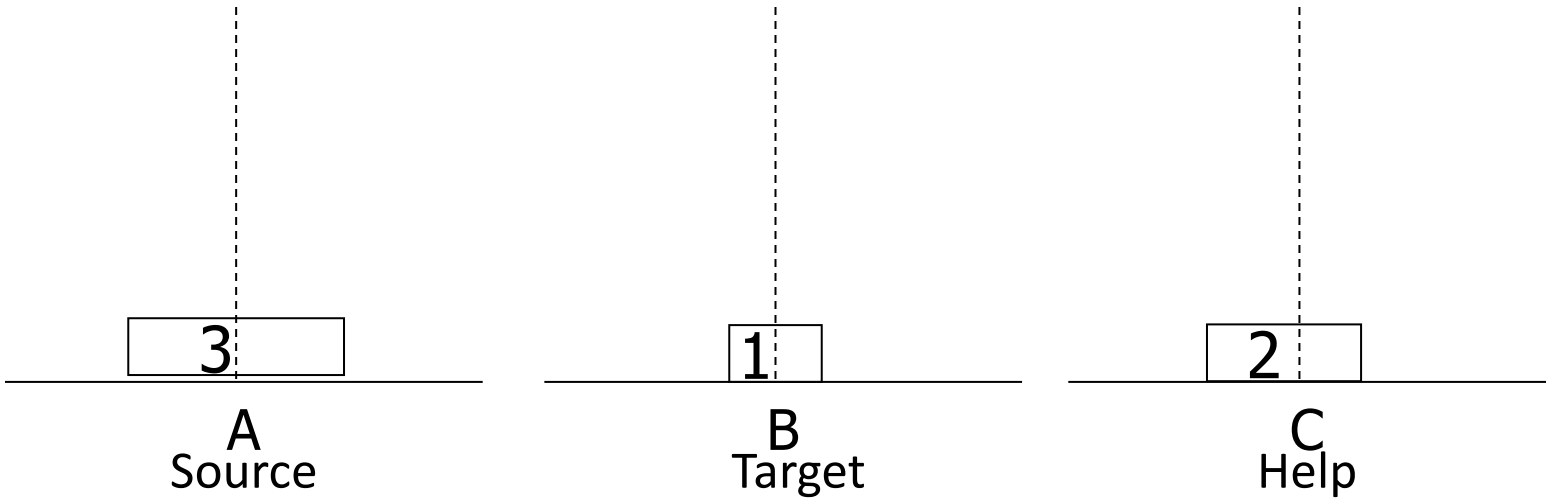
Hanoi Towers - Example



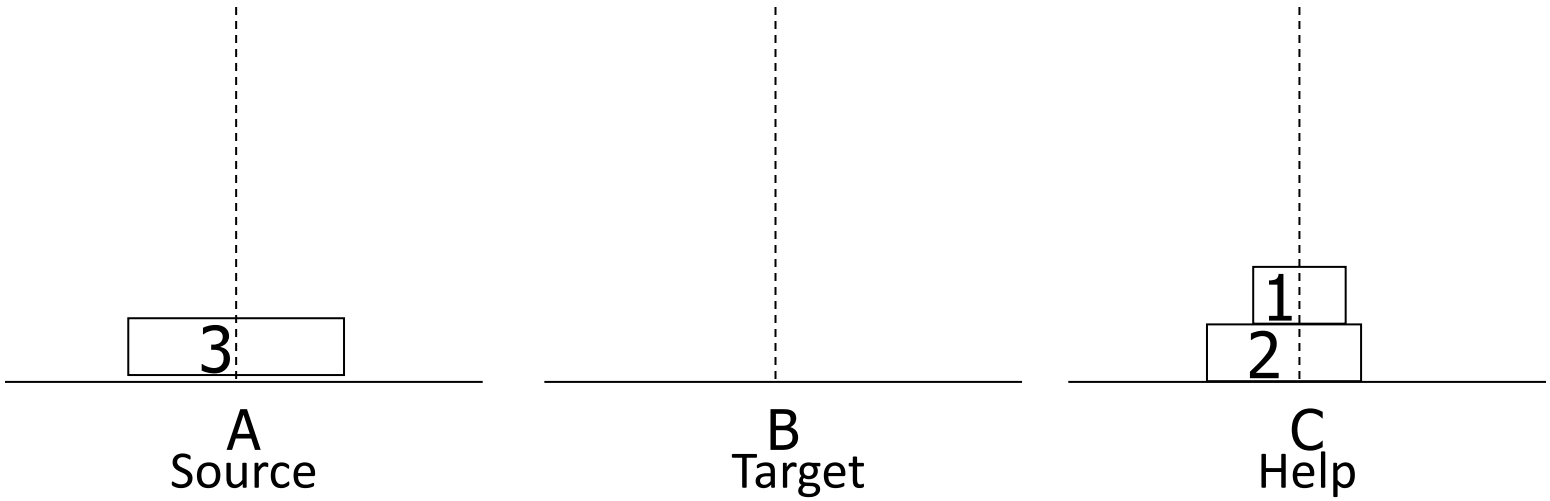
Hanoi Towers - Example



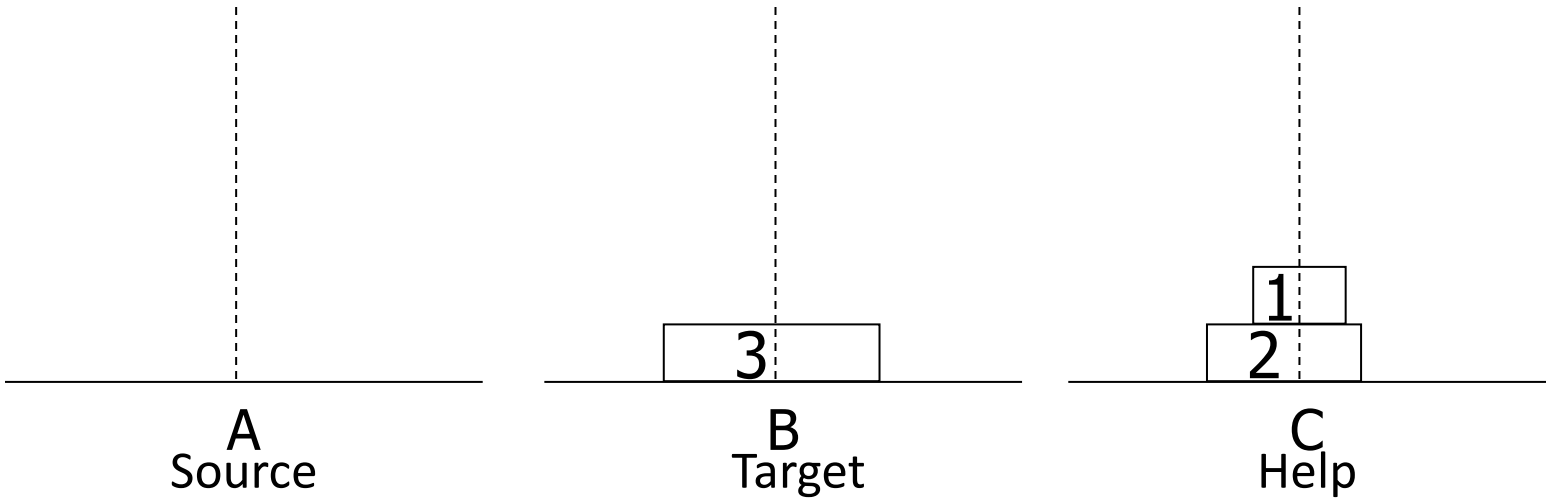
Hanoi Towers - Example



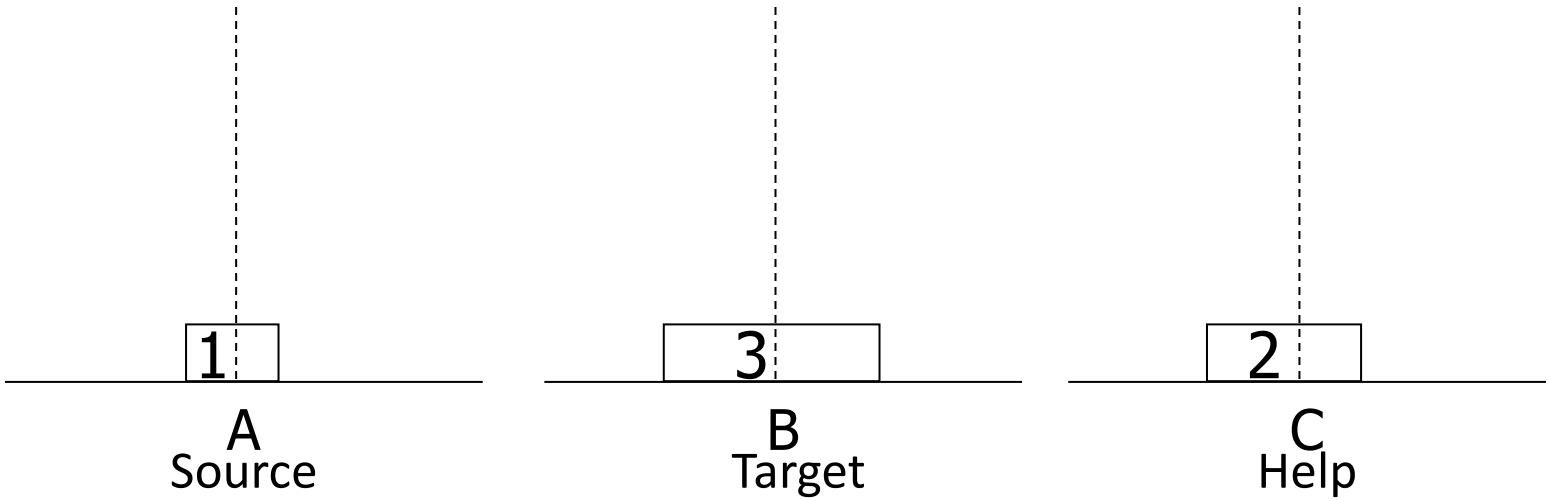
Hanoi Towers - Example



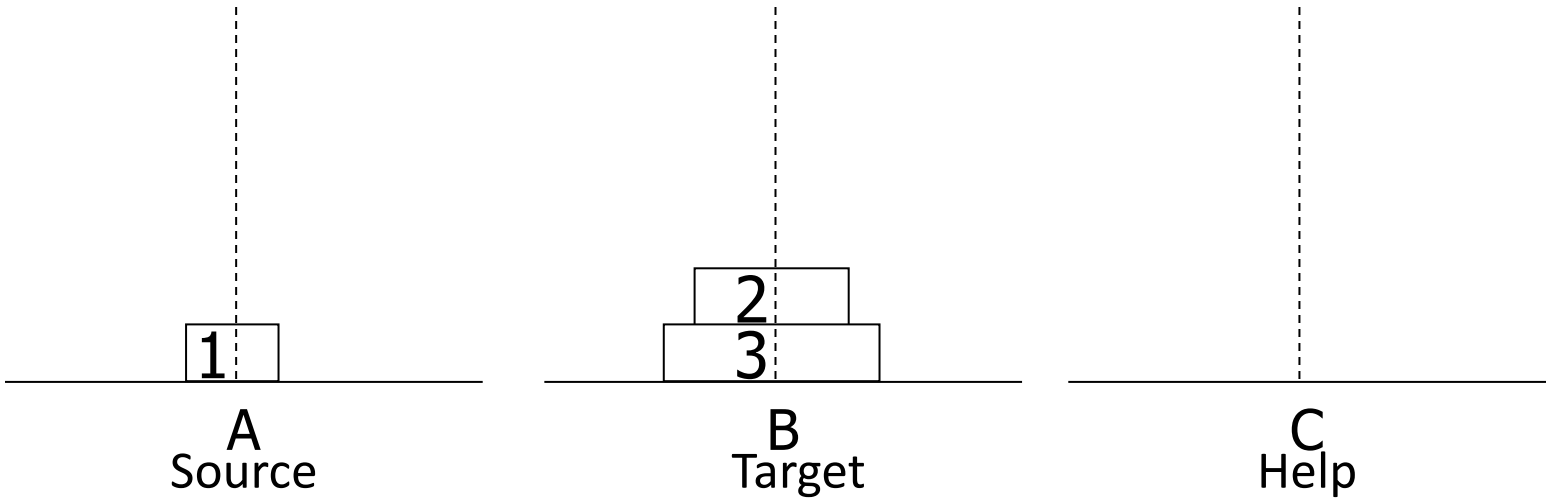
Hanoi Towers - Example



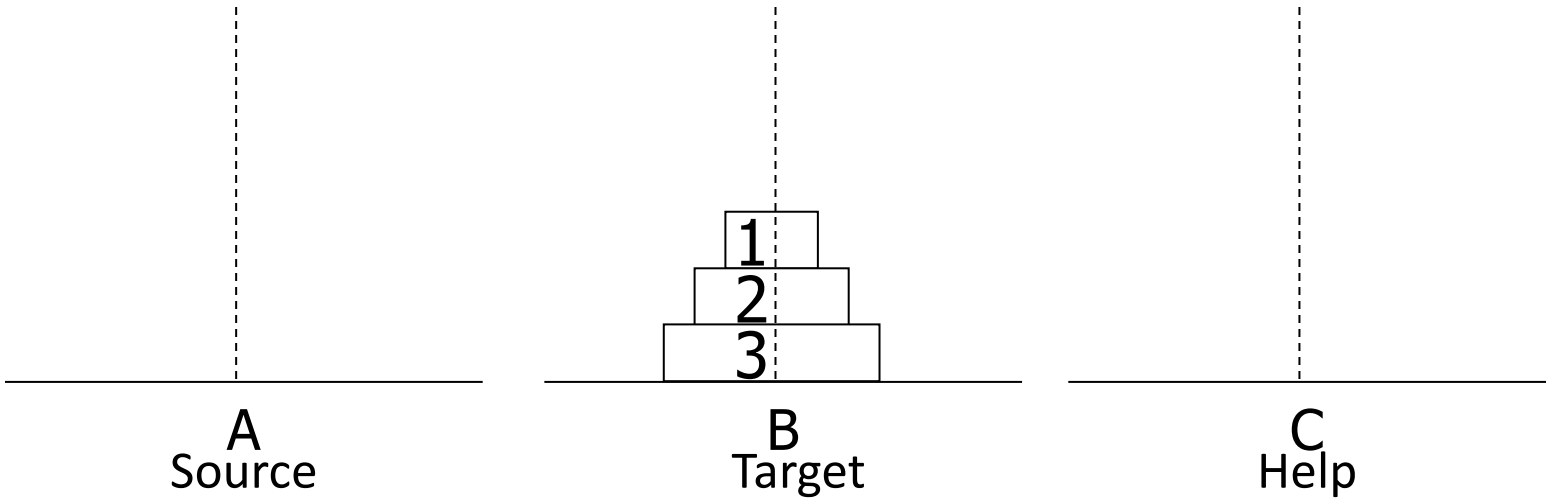
Hanoi Towers - Example



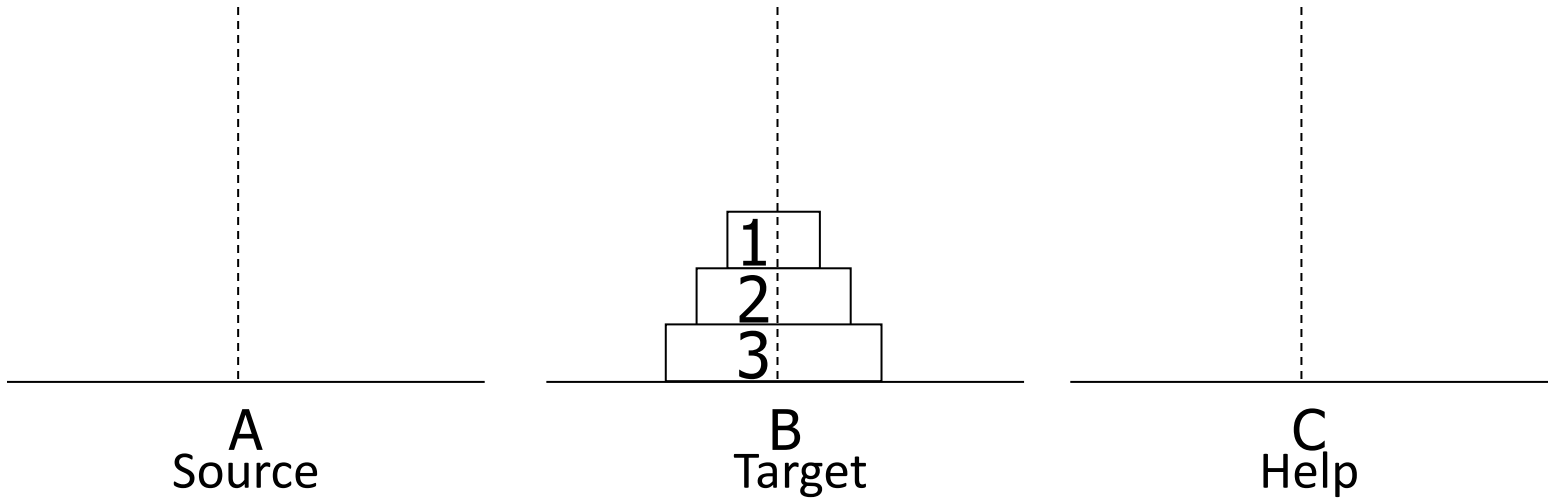
Hanoi Towers - Example



Hanoi Towers - Example



Hanoi Towers - Solution



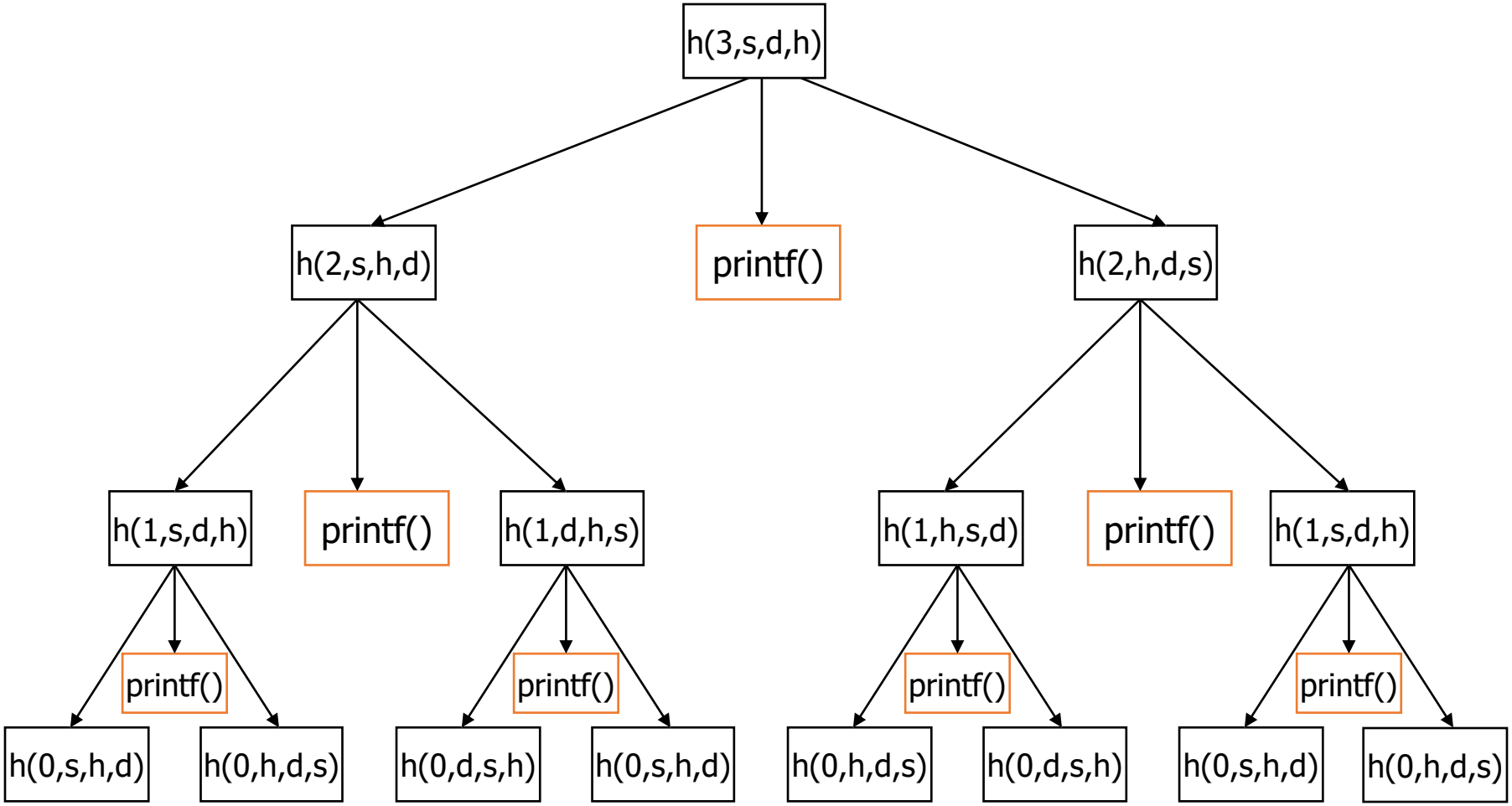
- Transfer disks [1, n-1] from pillar A to pillar C while using pillar B as helper
- Transfer disk n from pillar A to pillar B
- Transfer disks [1, n-1] from pillar C to pillar A while using pillar A

Hanoi Towers - Code

```java
public static void main(String[] args) {
      int numberOfDisks = 3;

      solveHanoiTowers(3, 'A', 'B', 'C');
}


private static void solveHanoiTowers(int numberOfDisks, char source, char destination, char help) {
      if(numberOfDisks == 0) {
            return;
      }

      solveHanoiTowers(numberOfDisks - 1, source, help, destination);

      System.out.println("Move disk " + numberOfDisks + " from " + source + " to " + destination);

      solveHanoiTowers(numberOfDisks - 1, help, destination, source);
}
```

```
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
```

Hanoi Towers - Breakdown



Drills

- The drills and solutions are in the repository
- The file is in drills/unit_13_recursion_drills.pdf
- The solutions are in src/unit_13.drills