CSE 2123
Recursion

Jeremy Morris

Past Few Weeks

- For the past few weeks we have been focusing on *data structures*
  - Classes & Object-oriented programming
  - Collections – Lists, Sets, Maps, etc.
- Now we turn our attention to *algorithms*
  - Algorithm: specific process for solving a problem
  - Specifically *recursive algorithms, search algorithms,* and *sorting algorithms*

Recursion

- Recursive methods
  - Methods that call themselves

- Recall that a method performs a specific task
  - A *recursive method* performs a specific task by calling itself to do it
    - How is that going to work?

Example: reverse a String

- Suppose we want to write a method to reverse a String:

```
/*
 * @param str – the String to be reversed
 * @return the reverse of str
*/
public static String reverse(String str)
```

Example: reverse a String

- Suppose we want to write a method to reverse a String:
  - We could do this with an iterative approach:

```
/*
 * @param str – the String to be reversed
 * @return the reverse of str
*/
public static String reverse(String str) {
     String rev = “”;
     for (int i=0; i<str.length(); i++) {
        rev = str.charAt(i)+rev;
     }
     return rev;
}
```

Example: reverse a String

- But suppose we have a static method that will already do most of the work for us?
  - reverseString will reverse almost any String
  - Except - we have to make our problem smaller first – it won't work on our String
    - Specifically, we can't do this:

```
public static String reverse(String str) {
     return reverseString(str);
}
```

Recursive Thinking – Subproblems

- Recursive problems often have this kind of "subproblem" structure
  - The big problem you're trying to solve is actually a smaller problem that is *exactly the same* as the big problem, combined with a simple additional step
- For the reverse a String problem, we can recognize:

```
reverse(str) ==
     reverse(str.substring(1,str.length()))
                                +str.charAt(0);
```

Recursive Thinking – Subproblems

- For the reverse a String problem, we can recognize:

```
reverse(str) ==
      reverse(str.substring(1,str.length()))
                                + str.charAt(0);
```

- So if we have some way to reverse a String of length 8, we could reverse a String of length 9 by:
  - Removing the first character
  - Reversing the String that's left
  - Appending the first character to the end of our reversed substring

Recursive Thinking – Subproblems

- For the reverse a String problem, we can recognize:

```
reverse(str) =
      reverse(str.substring(1,str.length()))
                                  + str.charAt(0);
```

- So if we have some way to reverse a String of length 8, we could reverse a String of length 9 by:
  - Removing the first character
  - Reversing the String that's left
  - Appending the first character to the end of our reversed substring

This is our *subproblem!*

Example: reverse a String

- So here's how we can use that reverseString method:

```
public static String reverse(String str) {
     String sub = str.substring(1,str.length());
     String rev = reverseString(sub);
     rev = rev + str.charAt(0);
     return rev;
}
```

Example: reverse a String

- So here's how we can use that reverseString method:
  - Does this code work for all input Strings?
    - What test cases will make this code fail?

```
public static String reverse(String str) {
     String sub = str.substring(1,str.length());
     String rev = reverseString(sub);
     rev = rev + str.charAt(0);
     return rev;
}
```

Example: reverse a String

- So here's how we can use that reverseString method:
  - Does this code work for all input Strings?
    - What test cases will make this code fail?

```
public static String reverse(String str) {
     String sub = str.substring(1,str.length());
     String rev = reverseString(sub);
     rev = rev + str.charAt(0);
     return rev;
}
```

What about empty ("") Strings?

Example: reverse a String

- This code takes care of our bad test case:

```
public static String reverse(String str) {
     if (str.length() == 0) {
        return str;
     }
     else {
        String sub = str.substring(1,str.length());
        String rev = reverseString(sub);
        rev = rev + str.charAt(0);
        return rev;
     }
}
```

Example: reverse a String

- Okay, so that works IF we have a method that solves our subproblem for us
  - But we don't
  - …or do we?
- What is a subproblem?
  - "a smaller problem that is *exactly the same* as the larger problem, combined with an extra step"
  - We have a method that solves the bigger problem
    - Let's call it to solve the smaller problem

Example: reverse a String

- This code is *recursive*:

```
public static String reverse(String str) {
     if (str.length() == 0) {
        return str;
     }
     else {
        String sub = str.substring(1,str.length());
        String rev = reverse(sub);
        rev = rev + str.charAt(0);
        return rev;
     }
}
```

Example: reverse a String

- This code is *recursive*:

```
public static String reverse(String str) {
     if (str.length() == 0) {
        return str;
     }
     else {
        String sub = str.substring(1,str.length());
        String rev = reverse(sub);
        rev = rev + str.charAt(0);
        return rev;
     }
}
```

The method calls *itself!*

Recursion

- If your code for a method is correct when it calls a hypothetical helper method that solves a smaller subproblem for it…
  - Then it is *also* correct when you replace that helper method with a call to *the method itself!*
- But this *only* works if you make the problem *smaller*
  - That is crucial for recursive thinking – you have to be working on a *subproblem*
  - Recursive solutions are only guaranteed to work if you make the problem smaller each time!

Another Example: raise to a power

- Suppose we want to write a method to raise an integer to an integer power:

```
/*
 * @param n – the base to be raised
 * @param p – the integer power > 0 to rase n to
 * @return n^p
*/
public static int power(int n, int p)
```

Another example: raise to a power

- What's the hidden subproblem here?
  - Think about this – can we break this up into a smaller problem that is *exactly the same* as our original problem (but smaller) and one extra step?
  - n^p = ?

Another example: raise to a power

- What's the hidden subproblem here?
  - Think about this – can we break this up into a smaller problem that is *exactly the same* as our original problem (but smaller) and one extra step?
  - $n^p = n * n^{p-1}$

Another example: raise to a power

```
/*
 * @param n – the base to be raised
 * @param p – the integer power > 0 to rase n to
 * @return n^p
*/
public static int power(int n, int p)
```

- $n^p = n * n^{p-1}$
- How can we write a recursive power method using this?

Another example: raise to a power

```
/*
 * @param n – the base to be raised
 * @param p – the integer power > 0 to rase n to
 * @return n^p
*/
public static int power(int n, int p)
```

- A different “smaller” problem
- $n^p = (n^{p/2})^2$ (If p>1 and p is even)
- How can we write a *different* recursive power method using this?
  - There’s often more than one way to solve a problem!
  - Which of these implementations is faster?

Properties of recursion

- Recursive methods will have two cases:
  - The *general case*
    - This is the recursive call to itself
    - This is where we make the problem smaller and use our method to solve that smaller problem
  - The *base case*
    - This is the non-recursive case
    - This is where the problem is as small as it is going to get and we need to solve it
    - The “simplest” case.

Example: reverse a String

```
public static String reverse(String str) {
     if (str.length() == 0) {
        return str;
     }
     else {
        String sub = str.substring(1,str.length());
        String rev = reverse(sub);
        rev = rev + str.charAt(0);
        return rev;
     }
}
```

Example: reverse a String

```
public static String reverse(String str) {
    if (str.length() == 0) {
        return str;
    }
    else {
        String sub = str.substring(1,str.length());
        String rev = reverse(sub);
        rev = rev + str.charAt(0);
        return rev;
    }
}
```

Base Case

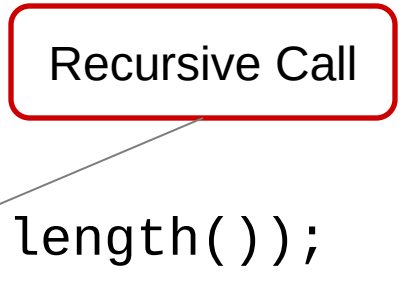
Example: reverse a String

```
public static String reverse(String str) {
     if (str.length() == 0) {
        return str;
     }
     else {
        String sub = str.substring(1,str.length());
        String rev = reverse(sub);
        rev = rev + str.charAt(0);
        return rev;
     }
}
```

General Case

Example: reverse a String

```
public static String reverse(String str) {
     if (str.length() == 0) {
        return str;
     }
     else {
        String sub = str.substring(1,str.length());
        String rev = reverse(sub);
        rev = rev + str.charAt(0);
        return rev;
     }
}
```



Uses of recursion I

- Recursion is often used to solve these "Divide and Conquer" problems
  - Solve a larger problem by:
    - Divide: Split problem into one or more smaller subproblems
    - Conquer: Solve the smaller problems
    - Combine: Merge smaller solutions into larger solution
  - Example: `reverse`
    - Divide: Split into subproblems: reverse of smaller substring
      - Base case: stop when length of String is 0
    - Conquer: Compute reverse of smaller string and "reverse" of first character
    - Combine: append first character to the end of the reversed substring

Class Example - sumArray

- Write a recursive method named `sumArray`

```
public static int sumArray(int[] a, int left, int right)
```

  - Returns the sum of all values between left and right in the array a
  - How do we solve this recursively?
    - What is our subproblem?
    - What is our base case?

Class Example - onlyPositive

- Write a recursive method named `onlyPositive`

```
public static boolean onlyPositive(int[] a, int left, int right)
```

  - Returns true if all values between left and right in the array a are positive
  - Otherwise, returns false
  - How do we solve this recursively?
    - What is our subproblem?
    - What is our base case?

Class Example - countChar

- Write a recursive method named `countChar`

```
public static int countChar(String str, char c)
```

  - Returns a count of number of times character c occurs in String str
    - Hint: Use str.charAt(), str.substring(), and str.length()
  - Divide & Conquer
    - Base Case?
    - General Case?

Class Example - isSorted

- Write a recursive method named `isSorted`

```
public static boolean isSorted(int[] a, int left, int right)
```

  - Returns true if all values between left and right in the array a are in increasing order
  - Otherwise, returns false
  - Divide & Conquer
    - What is our subproblem?
    - What is our Base Case?

Class Example - isPalindrome

- Write a recursive method named `isPalindrome`

```
public static boolean isPalindrome(String str)
```

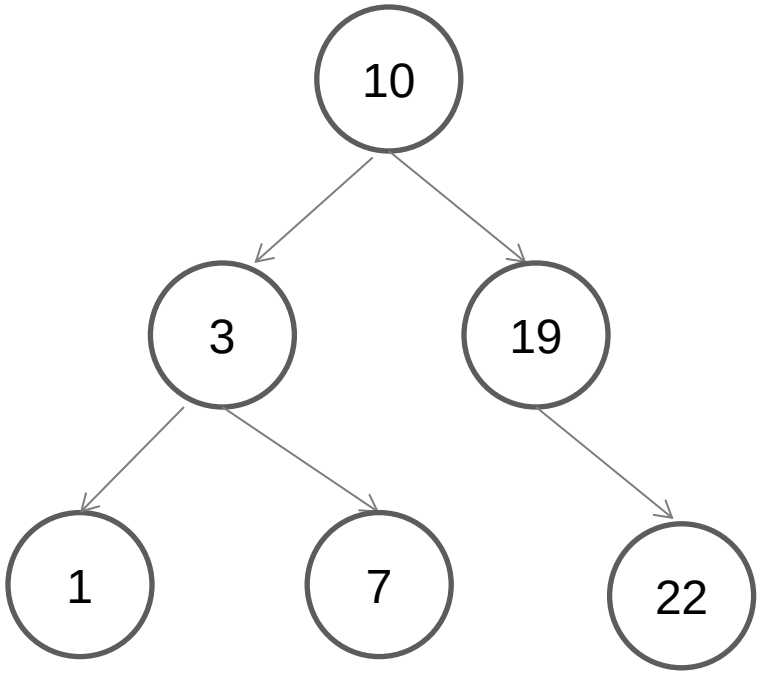
  - Returns true if the String str is a palindrome
  - Otherwise, returns false
  - Divide & Conquer
    - What is our subproblem?
    - What is our base case?

Uses of recursion II

- Recursion is also needed when we deal with *trees*
  - Why might this be the case?
  - Trees are an example of a *recursive* data structure!
- How are trees recursive?
  - Each tree has a root node that has some number of children.
  - Each child node also has some number of children
    - So each child node can be viewed as if it were the root of a new tree
    - A tree is therefore a root node and a collection of subtrees branching off that root node

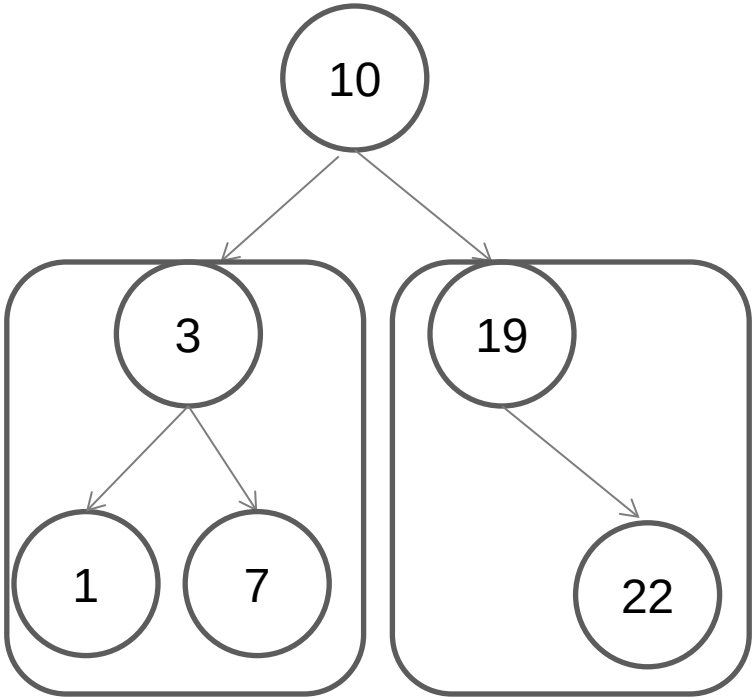
Back to the Binary Tree

- Recall the binary tree structure
  - Each *node* in the tree has up to 2 children (binary)



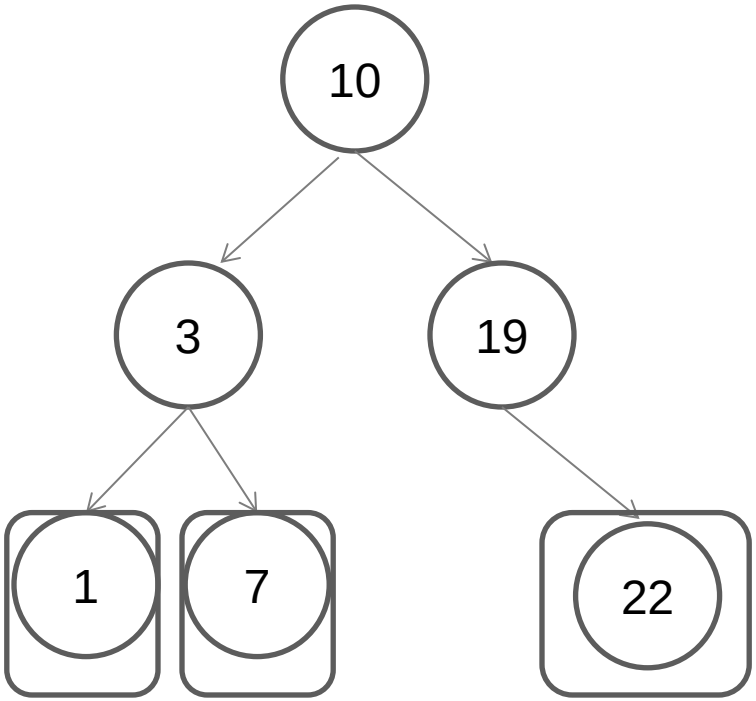
Back to the Binary Tree

- Recall the binary tree structure
  - Each *node* in the tree has up to 2 children (binary)
  - We can view each child node as the root of its own tree



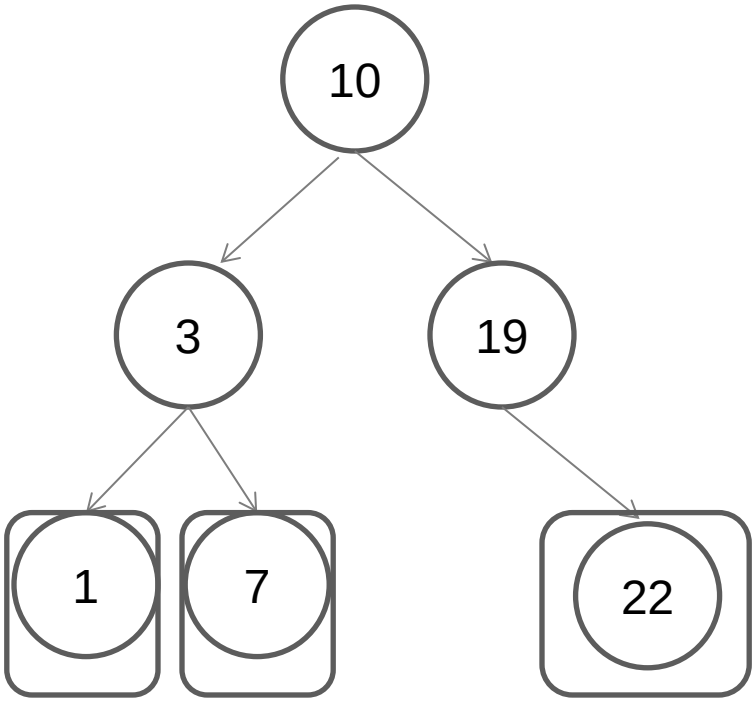
Back to the Binary Tree

- Recall the binary tree structure
  - Each *node* in the tree has up to 2 children (binary)
  - We can view each child node as the root of its own tree
    - And its children as the roots of their own trees



Back to the Binary Tree

- Recall the binary tree structure
  - Each *node* in the tree has up to 2 children (binary)
  - We can view each child node as the root of its own tree
    - And its children as the roots of their own trees
      - And so on



Recursion on Trees

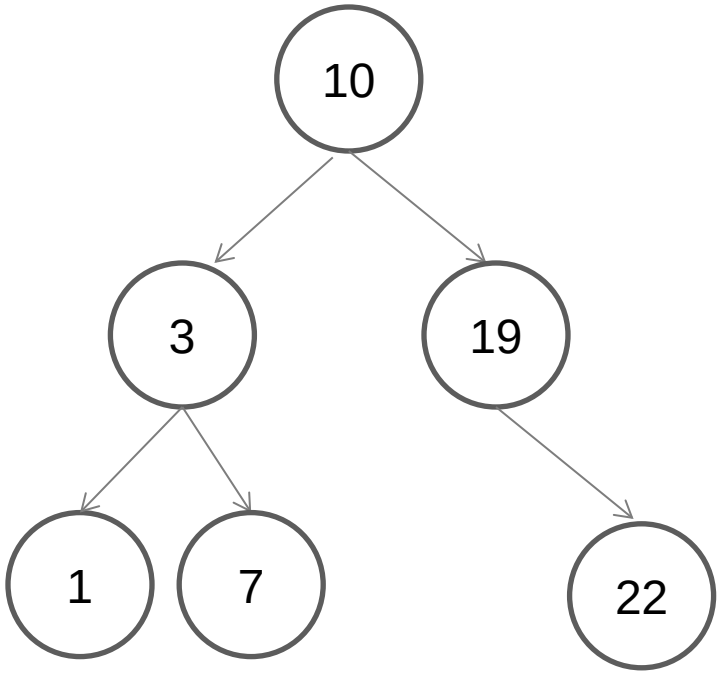
- Recursive algorithms are often easy to implement on trees
  - In fact, sometimes the recursive algorithm is the easiest way to do something with a tree
- And trees show up *a lot* in a computer science context
  - We've already seen binary search trees
    - And how useful they are for Maps, Sets, Priority Queues
  - HTML documents use a tree model
  - Databases use trees to store data

Recursion on Trees

- Let's consider the simple task of just *counting* all of the nodes in binary search tree
  - We might think of this as counting all of the "descendants" of the root node – children, grandchildren, great-grandchildren, etc., plus the root node itself
  - How would we do this?
- Let's consider how we might approach this iteratively – without recursion

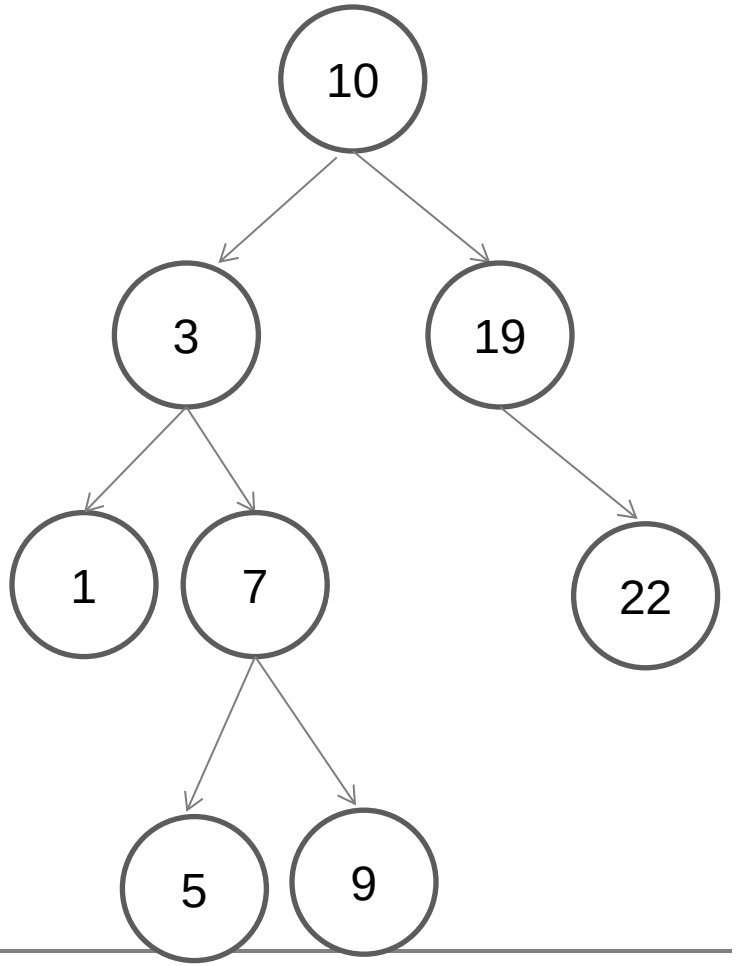
Counting nodes

- How would you write a `while` loop to count the nodes in this tree?



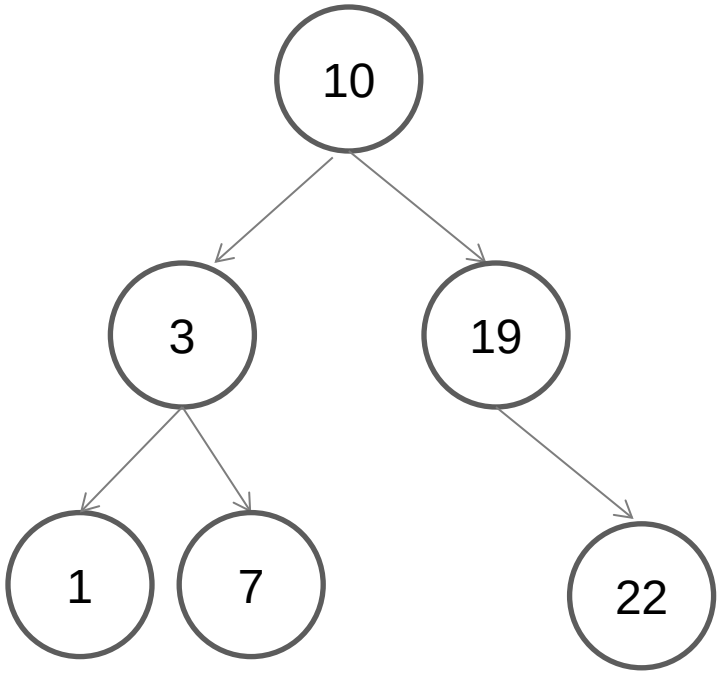
Counting nodes

- How would you write a `while` loop to count the nodes in this tree?
  - Now, will your loop still work if we do this?
  - Or do we need to do something else now?



Counting nodes

- Now let's think of a recursive solution
  - One that exploits the fact that each child node is the root of its own tree
    - What is our subproblem?
    - What is our base case?



Counting Nodes

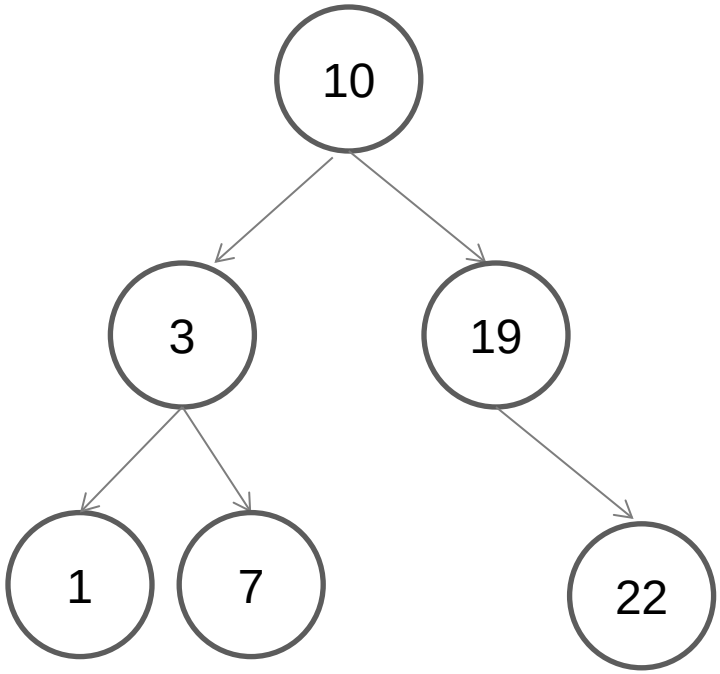
```
public static int countNodes(Node n) {




}
```

Recursion on Trees

- Now consider the task of visiting every node in a binary search tree *in order*
  - That is, turning a search tree into a sorted list by getting the elements in the proper order
  - How would we do this?
- Let's consider how we might approach this iteratively

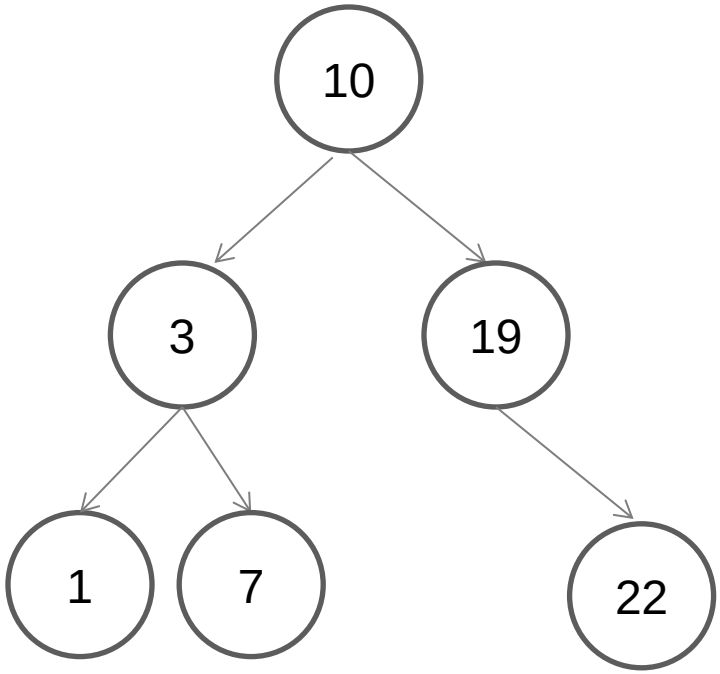
Traversing the Tree

- How would you write a `while` loop to traverse this tree?
  - Will it work if we change the tree to something bigger? Smaller?



Traversing the Tree

- Now let's think of a recursive solution
  - One that exploits the fact that each child node is the root of its own tree
    - What is our subproblem?
    - What is our base case?



Traverse the tree

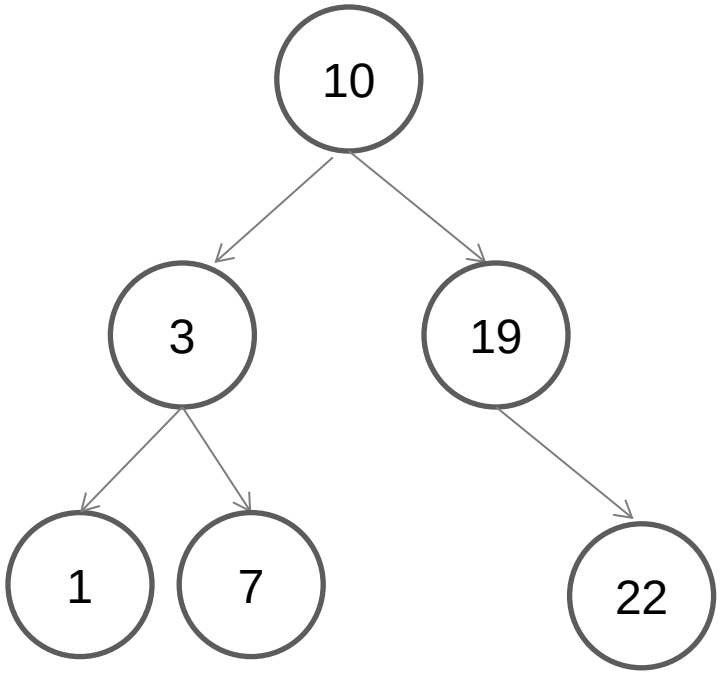
```
public static void traverse(Node n) {




}
```

Recursion on Trees

- Next let's consider the task of inserting a new node into the tree
  - How can we add a node to the proper place in the tree?
- Let's consider how we might approach *this* iteratively

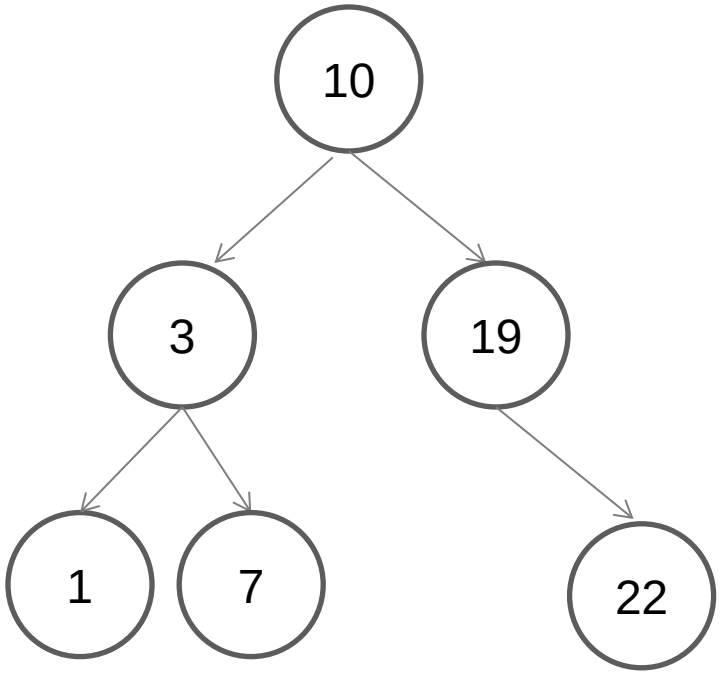
Inserting into the Tree

- Suppose we want to insert a value
  - How would you use a `while` loop to insert into this tree?
  - Will your solution work if we change the tree?



Inserting into the Tree

- Now let's think of a recursive solution
  - One that exploits the fact that each child node is the root of its own tree
    - What is our subproblem?
    - What is our base case?



Insert into the tree

```
public static void insert(Node root, int n) {




}
```

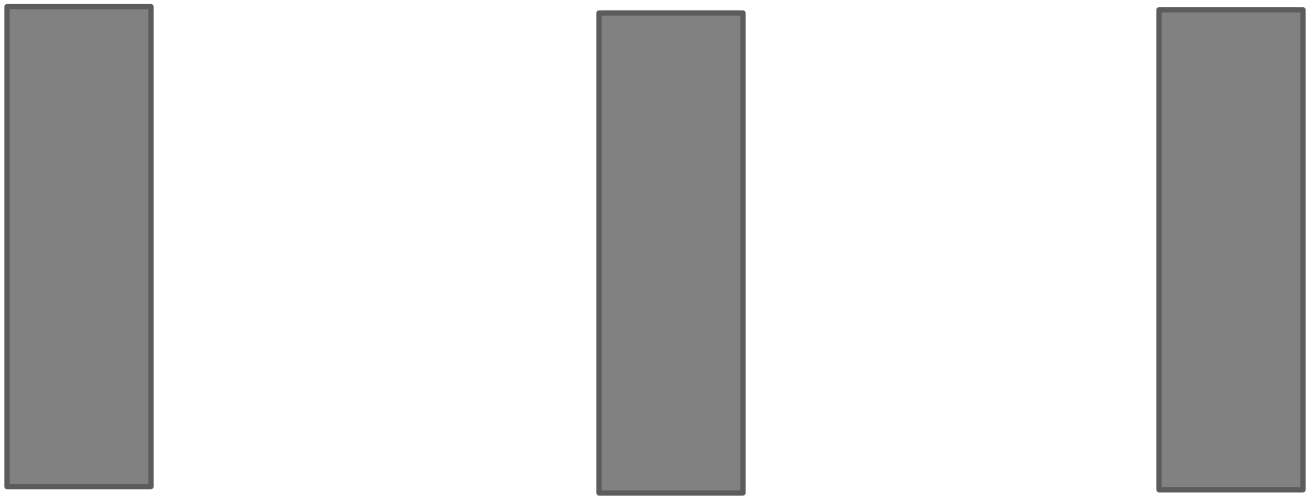
Recursion on Trees

- Trees are an example where recursion is not just useful, it's better than the alternative
  - Sometimes we will have a choice of approaches with an algorithm
    - We could use an iterative approach or a recursive approach
    - The one we use will be the one that most naturally fits the problem
  - But other times we don't have much choice – we must use the recursive approach
    - Trees are an example of this

Understanding Recursion- Toy Problem

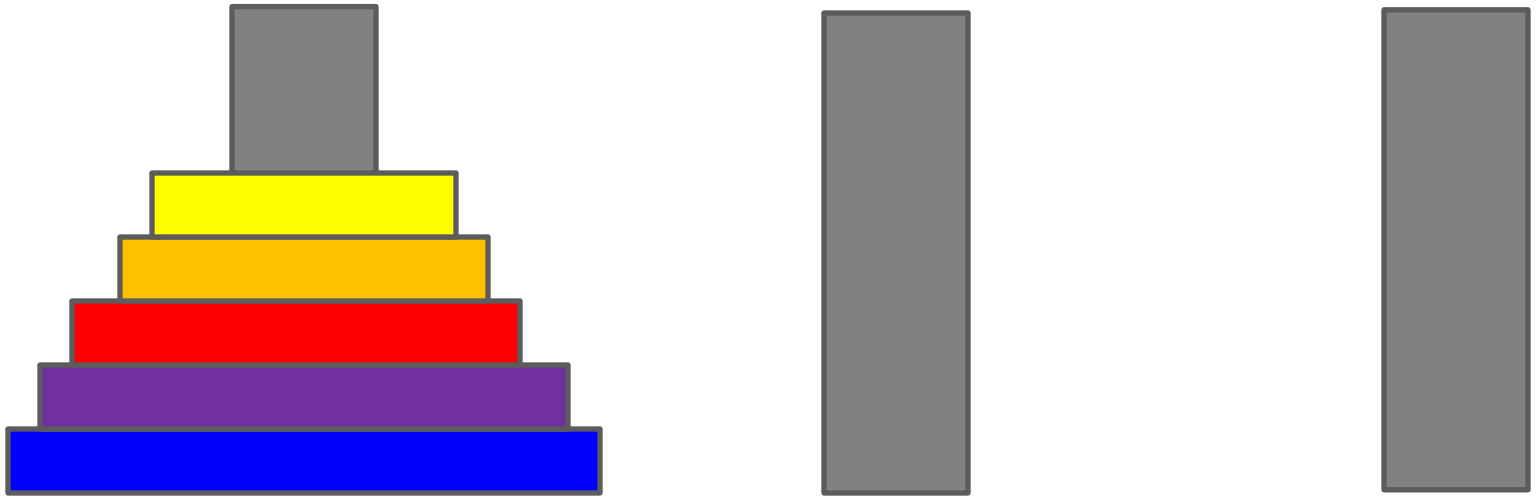
- Sometimes, we like to use “toy problem” to help us understand an idea
  - Simple problems like “reversing a String” are examples of toy problems
    - They’re not terribly interesting
- A famous “toy problem” for recursion is the Towers of Hanoi problem

Towers of Hanoi



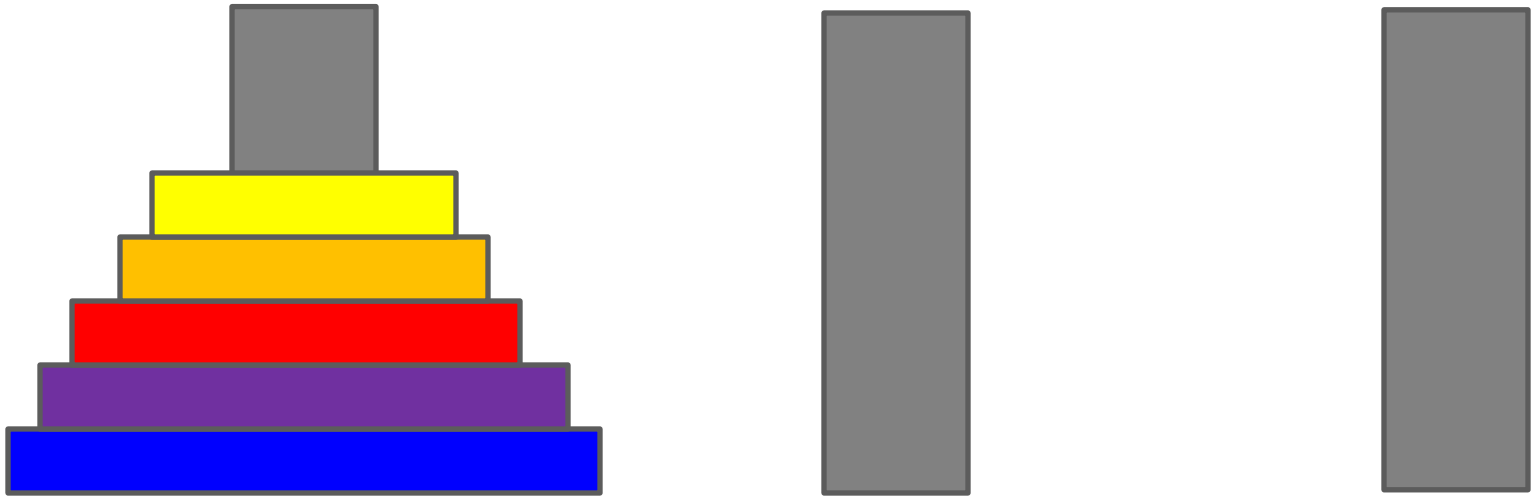
- In the Towers of Hanoi problem there are three posts

Towers of Hanoi



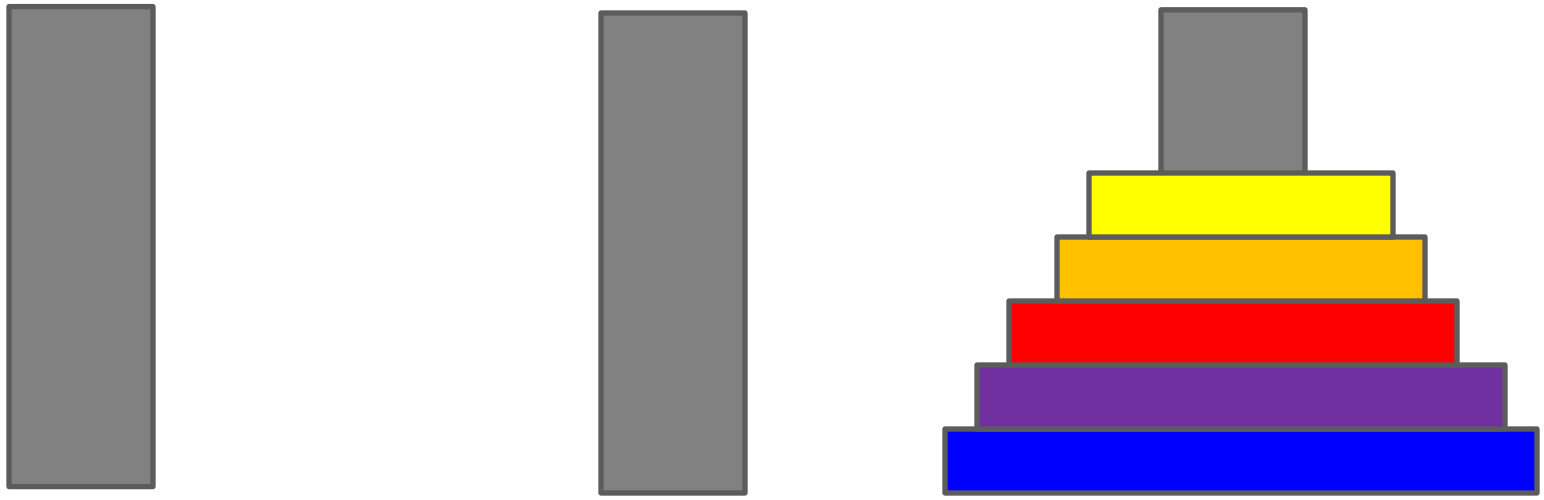
- In the Towers of Hanoi problem there are three posts
  - On one post is a set of discs of different sizes

Towers of Hanoi



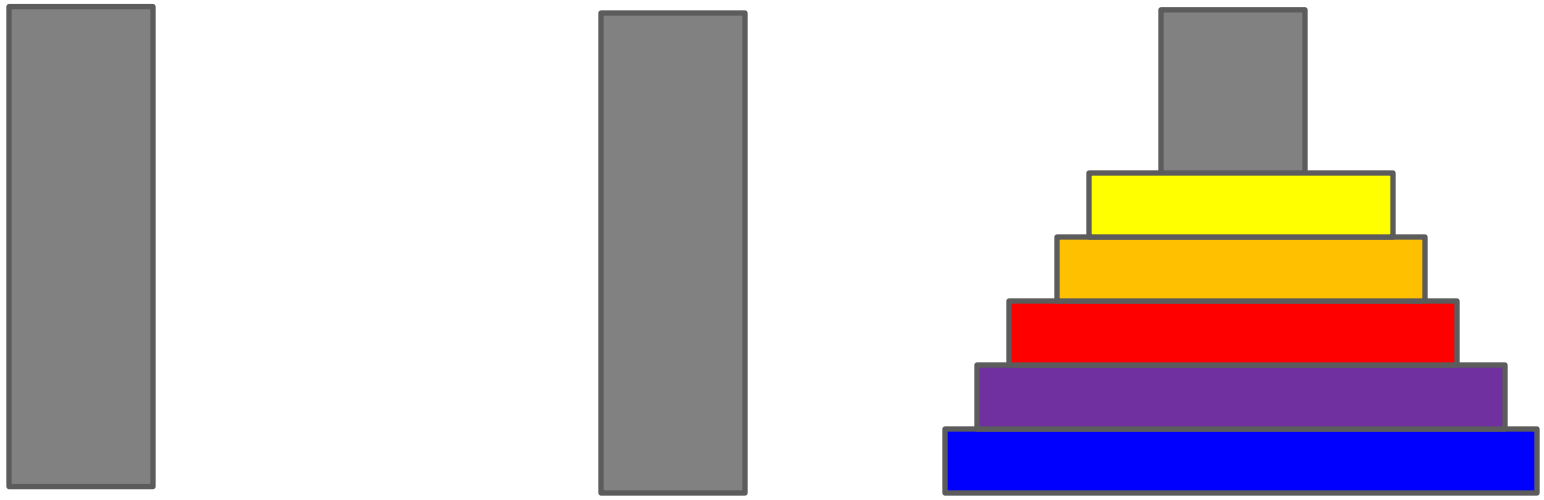
- In the Towers of Hanoi problem there are three posts
  - On one post is a set of discs of different sizes
  - The goal is to move all of the discs from the first post to the last post

Towers of Hanoi



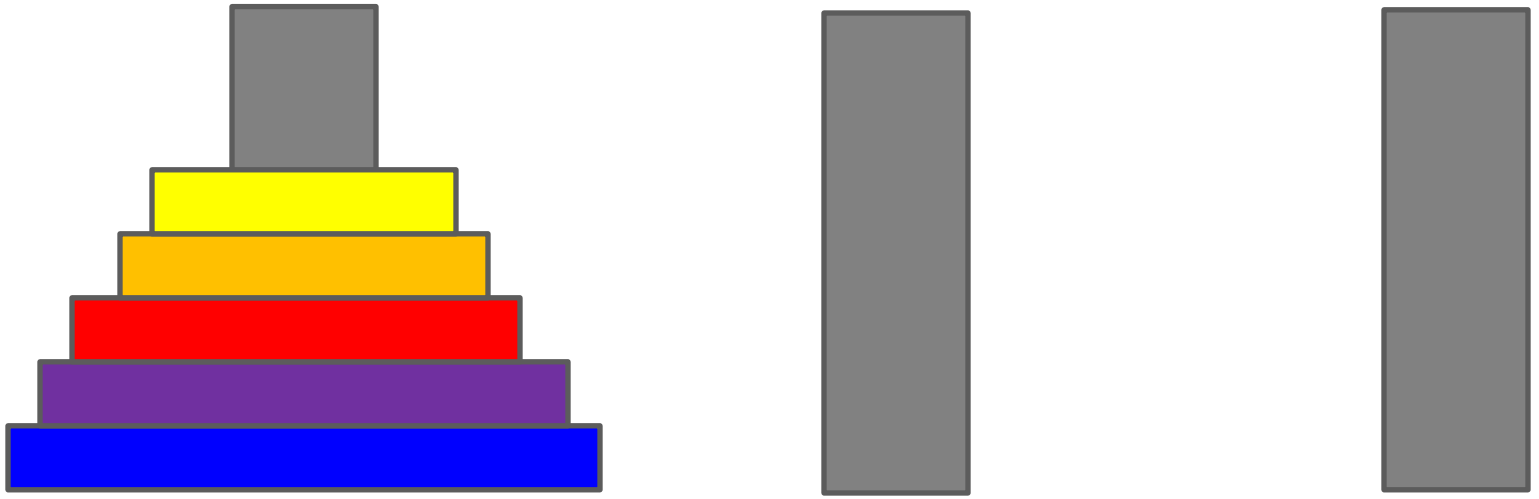
- In the Towers of Hanoi problem there are three posts
  - On one post is a set of discs of different sizes
  - The goal is to move all of the discs from the first post to the last post

Towers of Hanoi



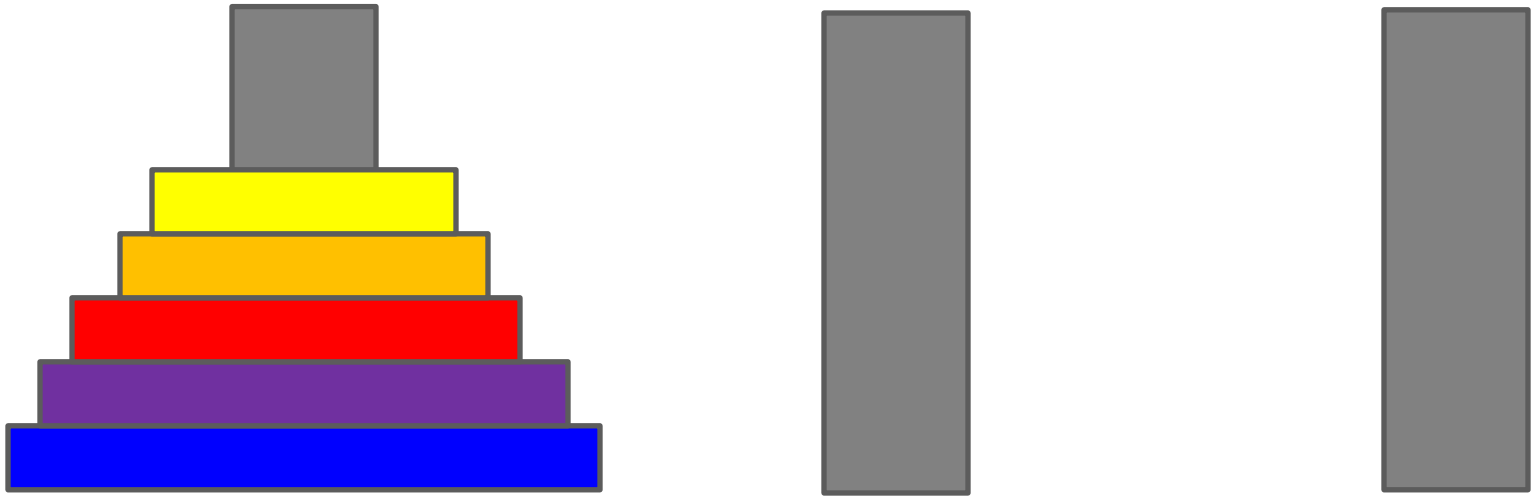
- The goal is to move all of the discs from the first post to the last post
  - BUT – it isn't that easy

Towers of Hanoi



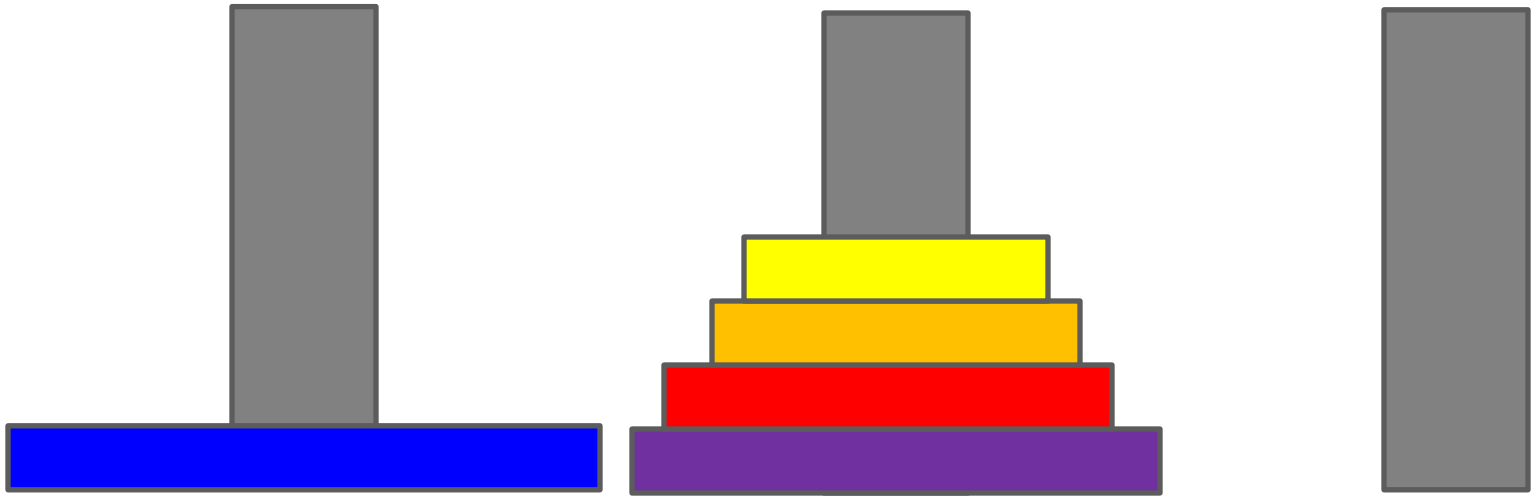
- The goal is to move all of the discs from the first post to the last post
  - BUT – it isn't that easy
  - We can only move the discs one at a time
  - And no disc can be on top of a smaller disc
    - How are we going to do this?

Towers of Hanoi



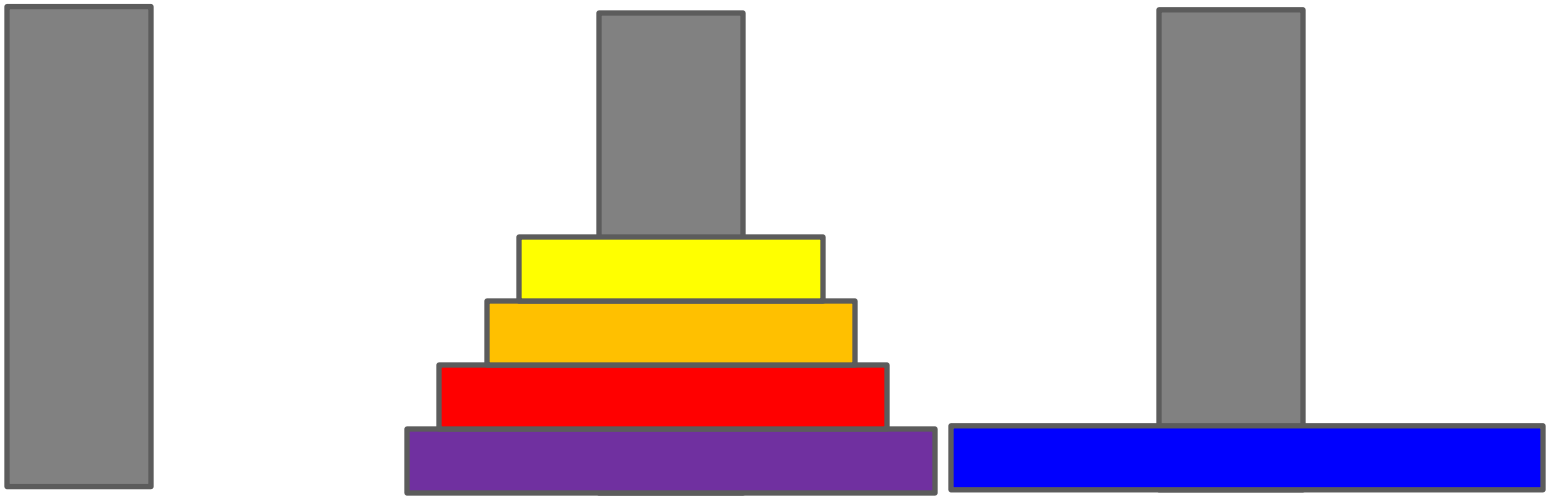
- How are we going to do this?
  - The key is to grasp the *recursive* nature of this problem
  - To move 5 discs to the last peg, we must first move 4 discs to the middle peg

Towers of Hanoi



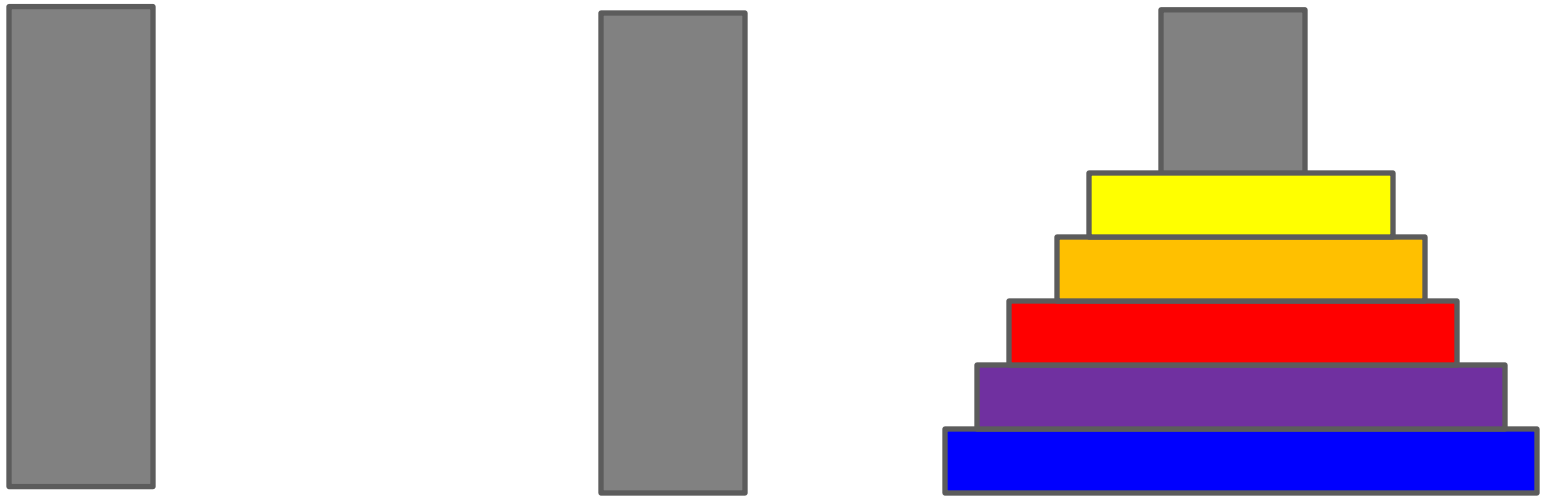
- How are we going to do this?
  - The key is to grasp the *recursive* nature of this problem
  - To move 5 discs to the last peg, we must first move 4 discs to the middle peg

Towers of Hanoi



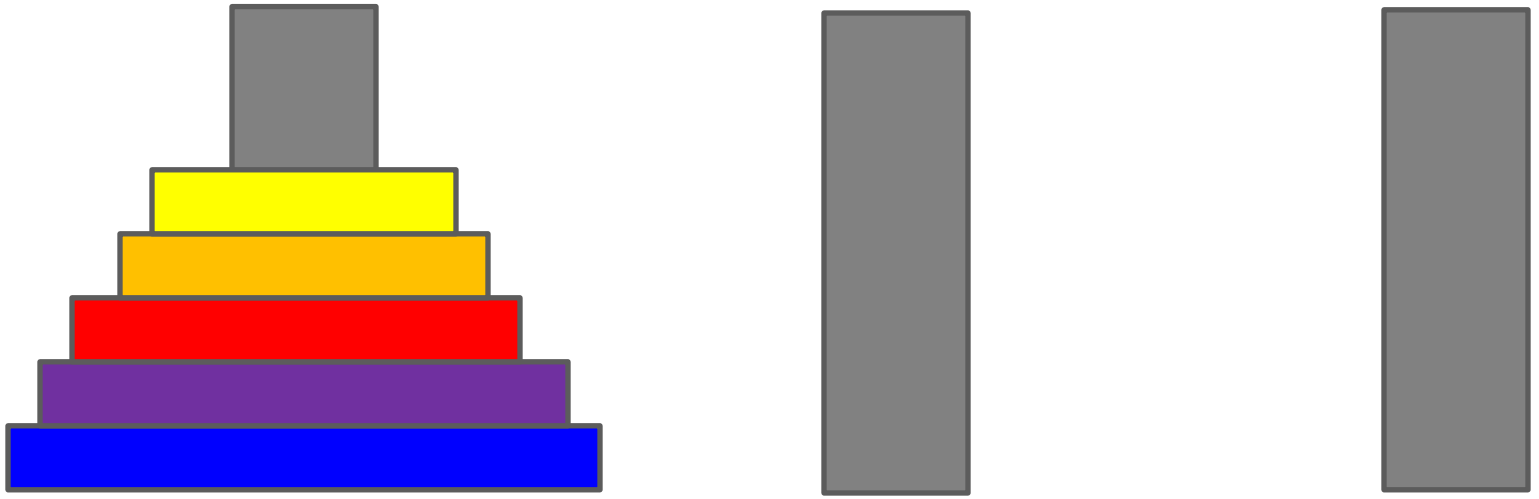
- How are we going to do this?
  - The key is to grasp the *recursive* nature of this problem
  - To move 5 discs to the last peg, we must first move 4 discs to the middle peg
  - Then move the big disc to the last peg

Towers of Hanoi



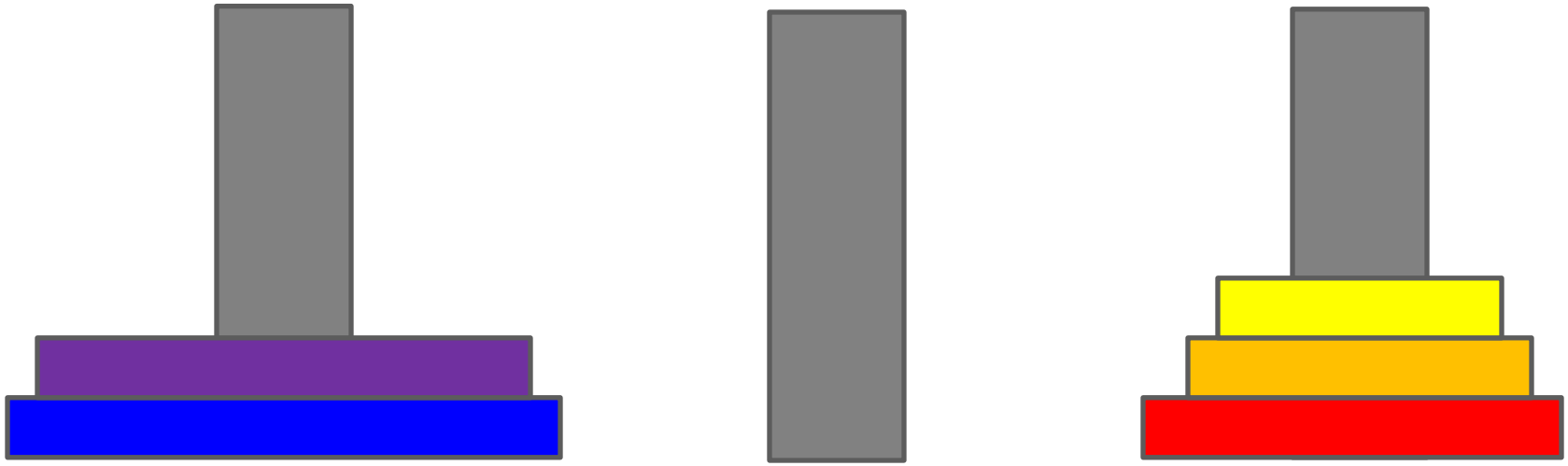
- How are we going to do this?
  - The key is to grasp the *recursive* nature of this problem
  - To move 5 discs to the last peg, we must first move 4 discs to the middle peg
  - Then move the big disc to the last peg
  - Then move 4 discs to the last peg

Towers of Hanoi



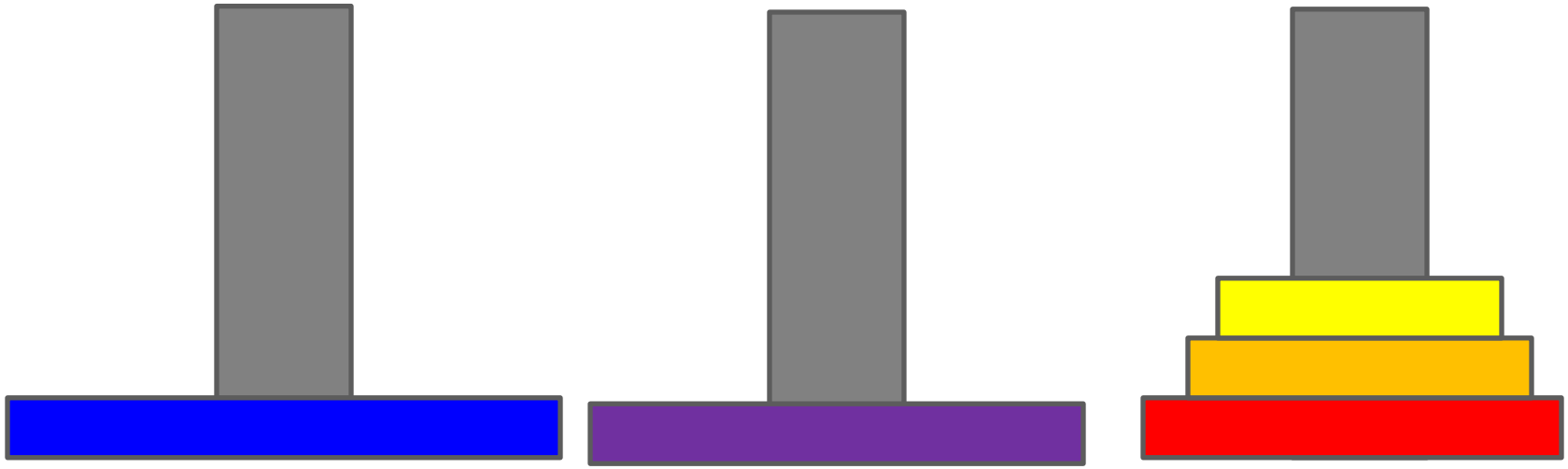
- Okay, how do we move 4 discs to the middle peg?
  - First we move 3 discs to the last peg

Towers of Hanoi



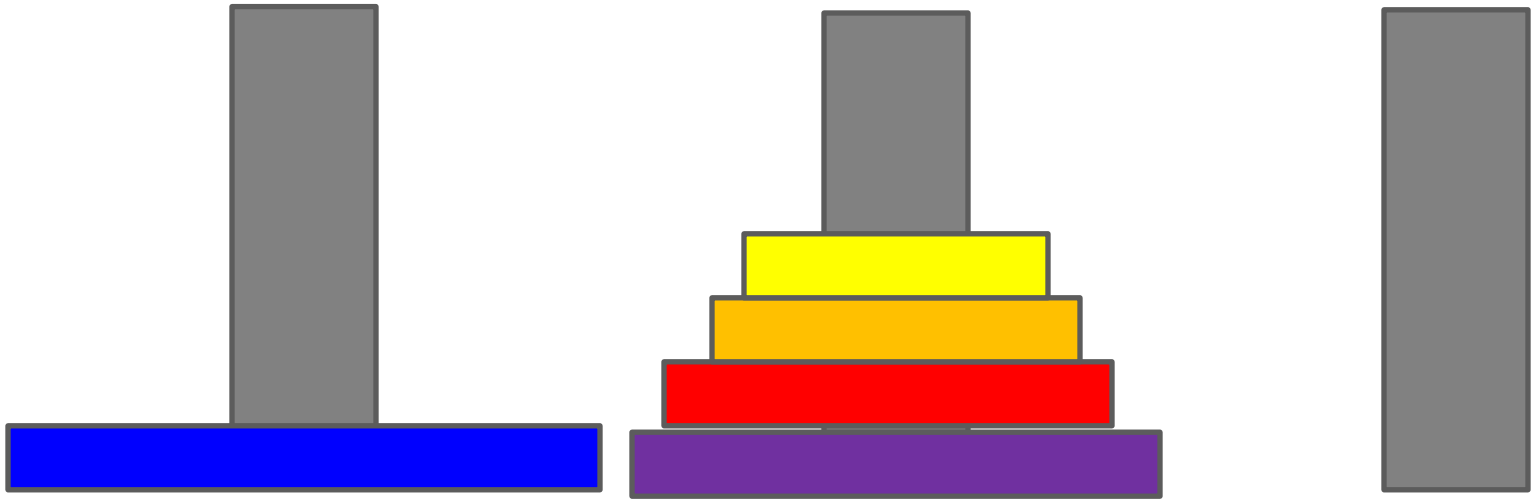
- Okay, how do we move 4 discs to the middle peg?
  - First we move 3 discs to the last peg
  - Then the fourth disc to the middle peg

Towers of Hanoi



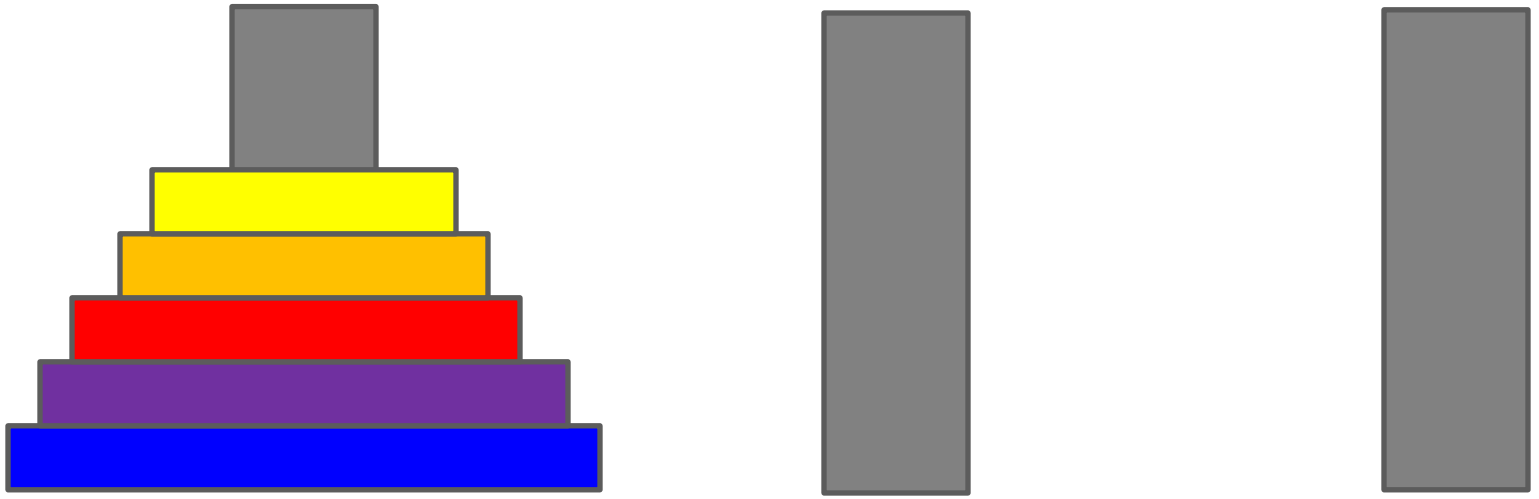
- Okay, how do we move 4 discs to the middle peg?
  - First we move 3 discs to the last peg
  - Then the fourth disc to the middle peg
  - Then the three discs to the middle peg

Towers of Hanoi



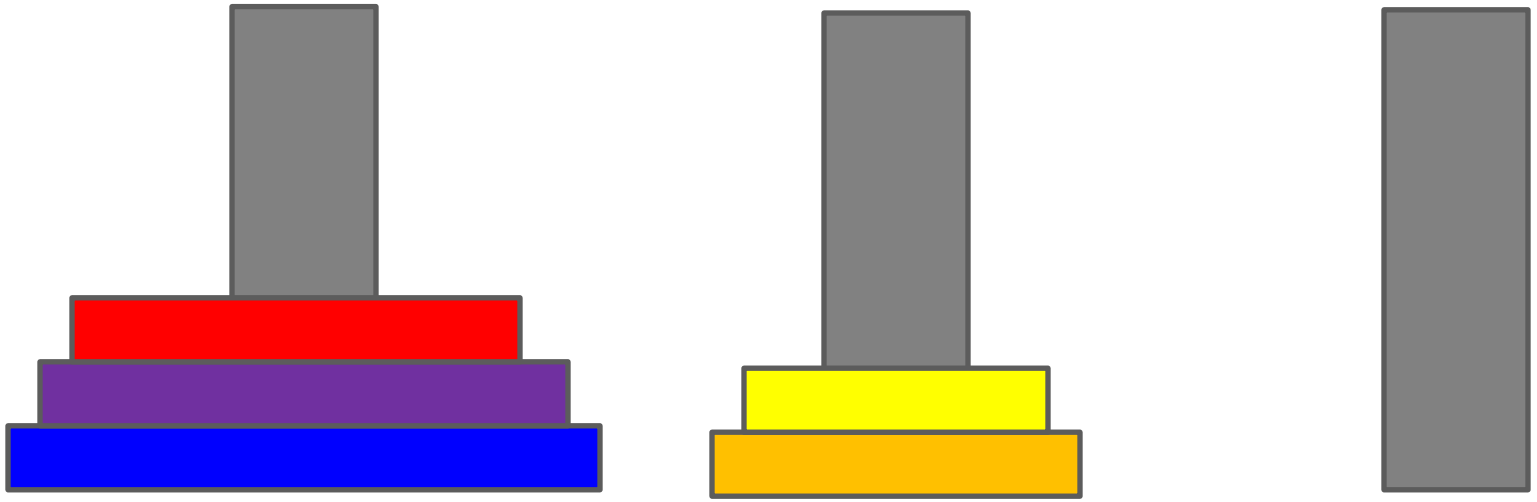
- Okay, how do we move 4 discs to the middle peg?
  - First we move 3 discs to the last peg
  - Then the fourth disc to the middle peg
  - Then the three discs to the middle peg

Towers of Hanoi



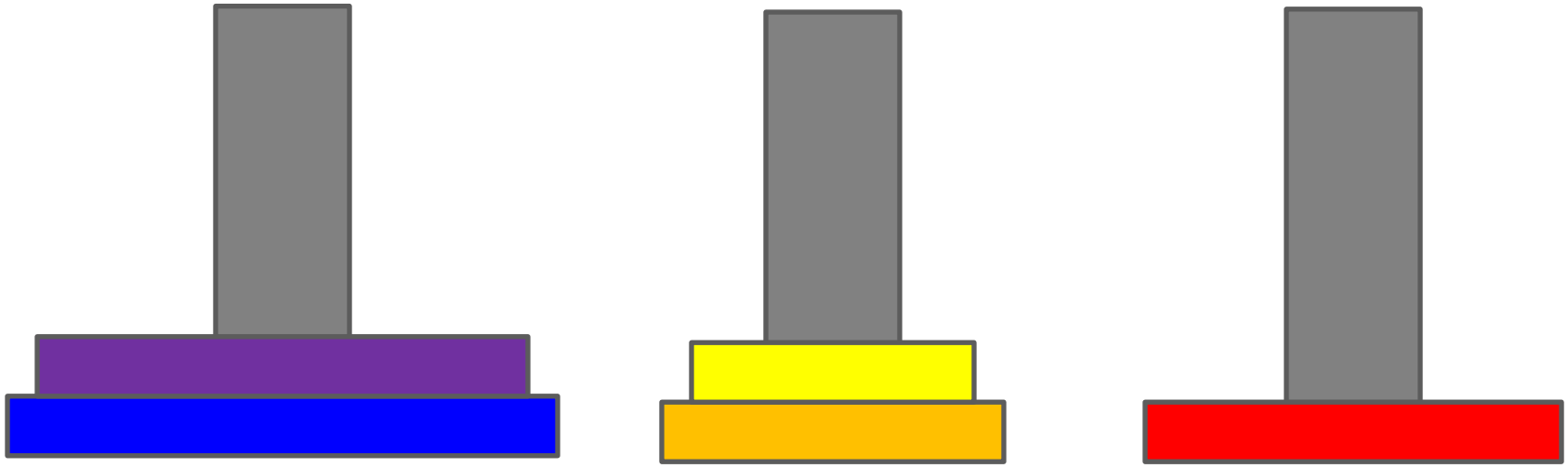
- Okay, how do we move 3 discs to the last peg?
  - First we move 2 discs to the middle peg

Towers of Hanoi



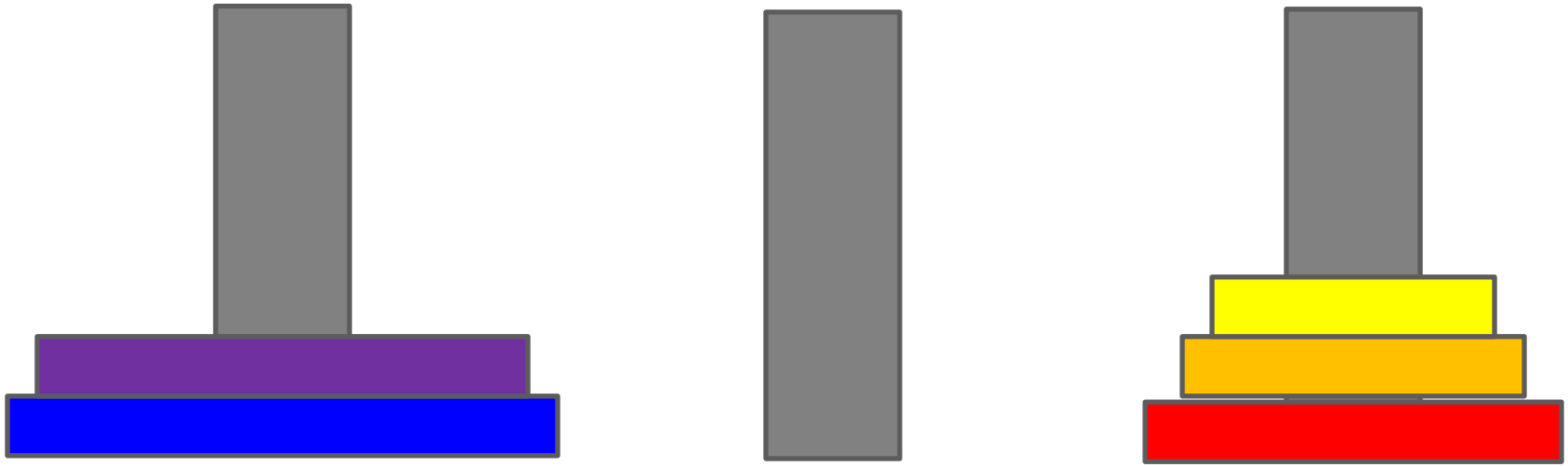
- Okay, how do we move 3 discs to the last peg?
  - First we move 2 discs to the middle peg

Towers of Hanoi



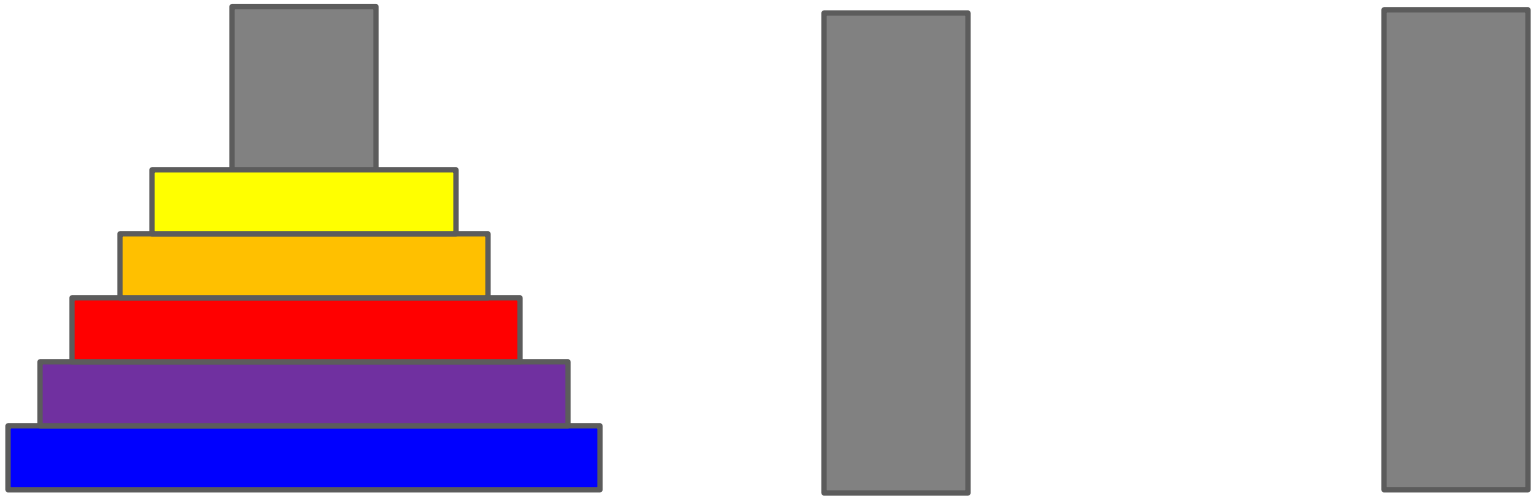
- Okay, how do we move 3 discs to the last peg?
  - First we move 2 discs to the middle peg
  - Then the third disc to the last peg

Towers of Hanoi



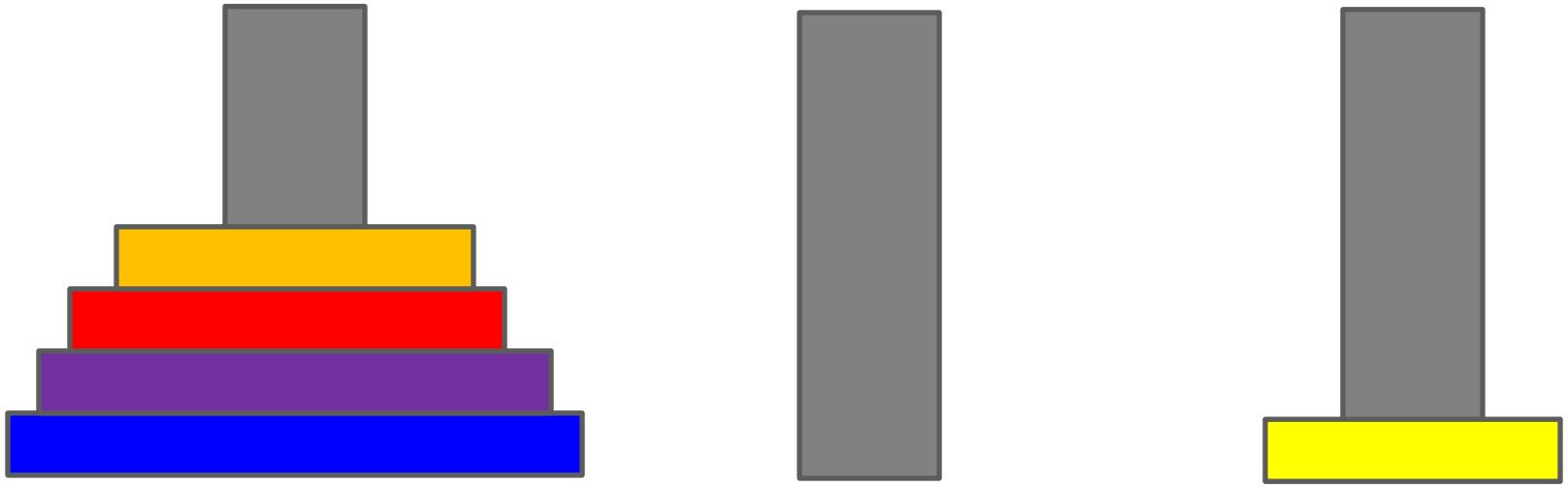
- Okay, how do we move 3 discs to the last peg?
  - First we move 2 discs to the middle peg
  - Then the third disc to the last peg
  - Then the two discs to the last peg

Towers of Hanoi



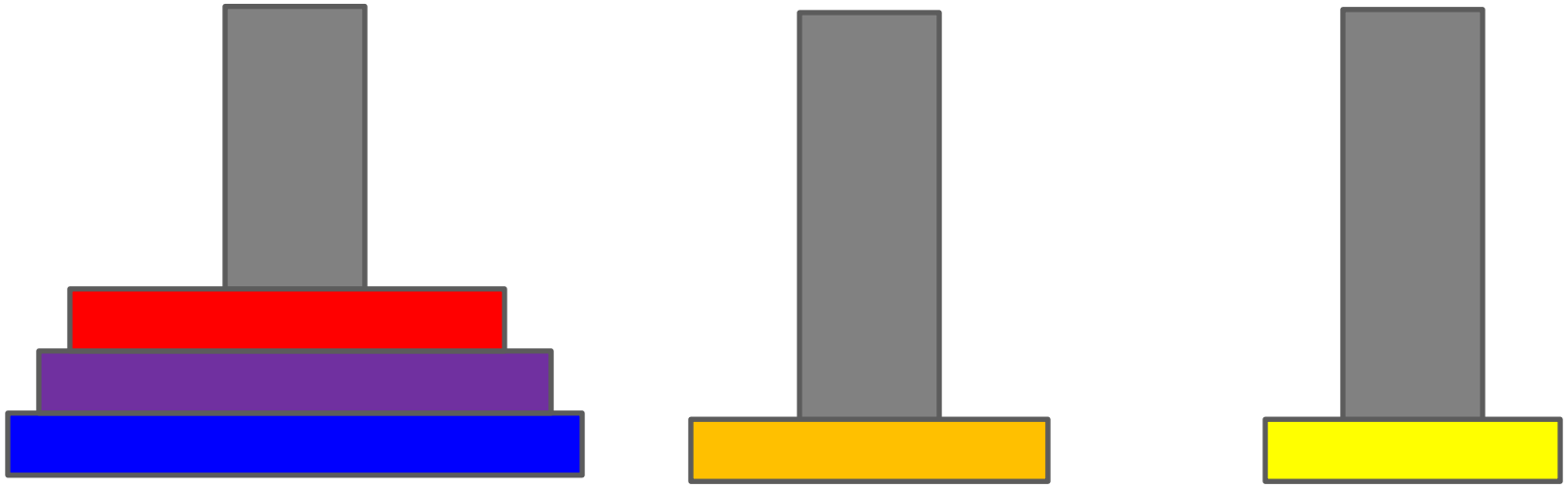
- Okay, how do we move 2 discs to the middle peg?
  - First we move 1 disc to the last peg

Towers of Hanoi



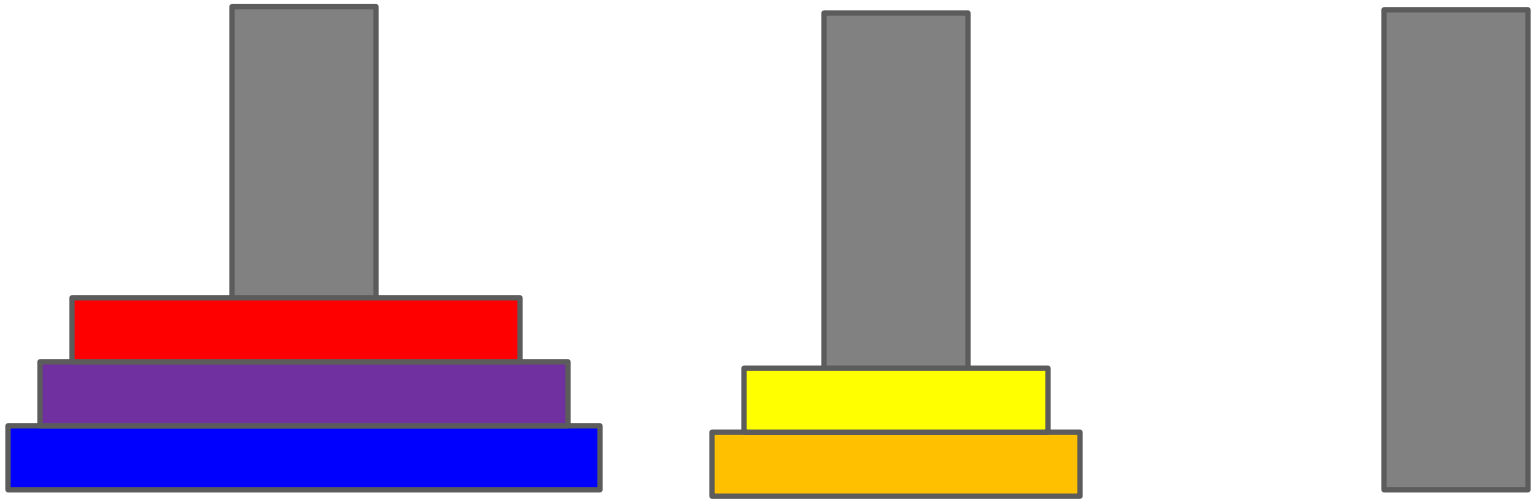
- Okay, how do we move 2 discs to the middle peg?
  - First we move 1 disc to the last peg
  - Then we move the second disc to the middle peg

Towers of Hanoi



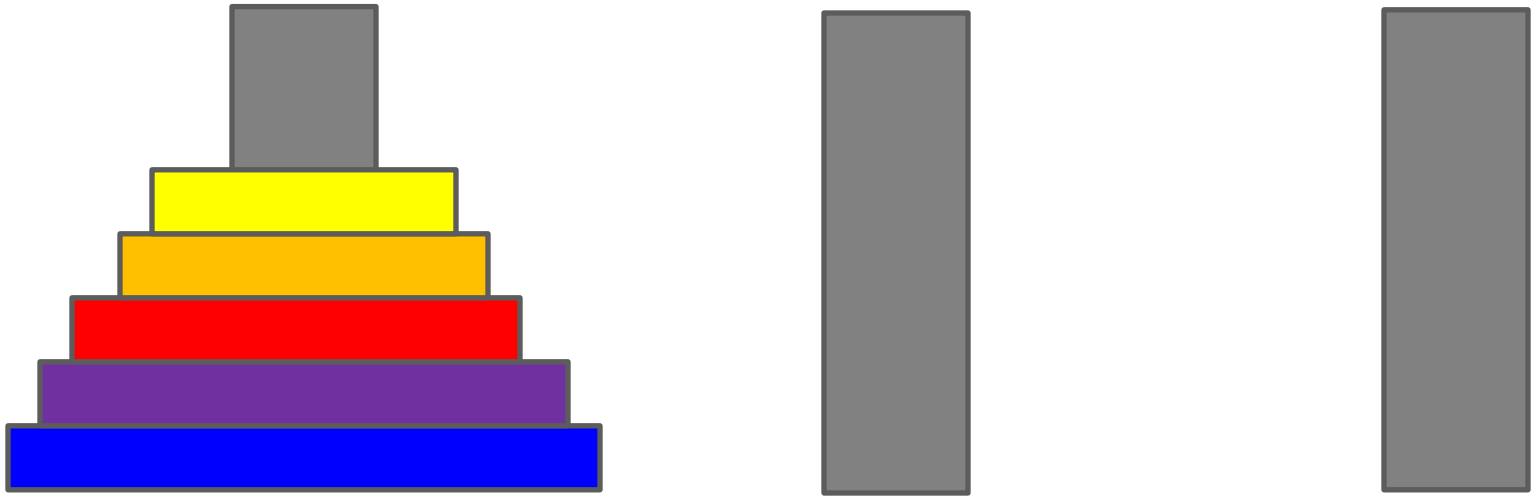
- Okay, how do we move 2 discs to the middle peg?
  - First we move 1 disc to the last peg
  - Then we move the second disc to the middle peg
  - Then the 1 disc to the middle peg

Towers of Hanoi



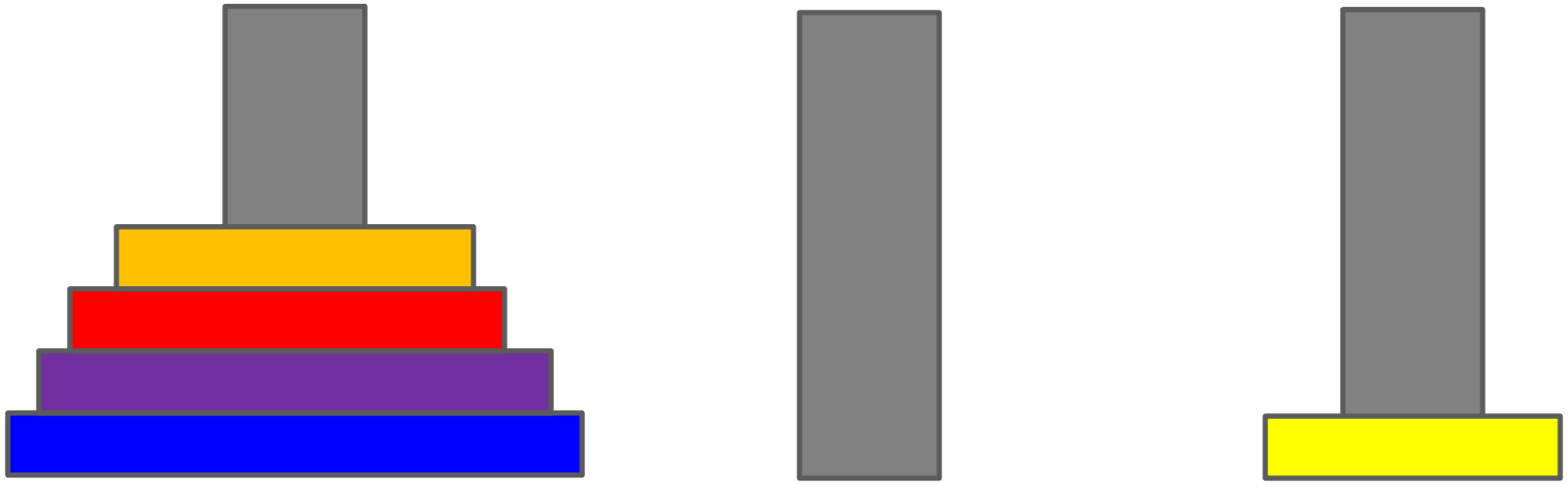
- Okay, how do we move 2 discs to the middle peg?
  - First we move 1 disc to the last peg
  - Then we move the second disc to the middle peg
  - Then the 1 disc to the middle peg

Towers of Hanoi



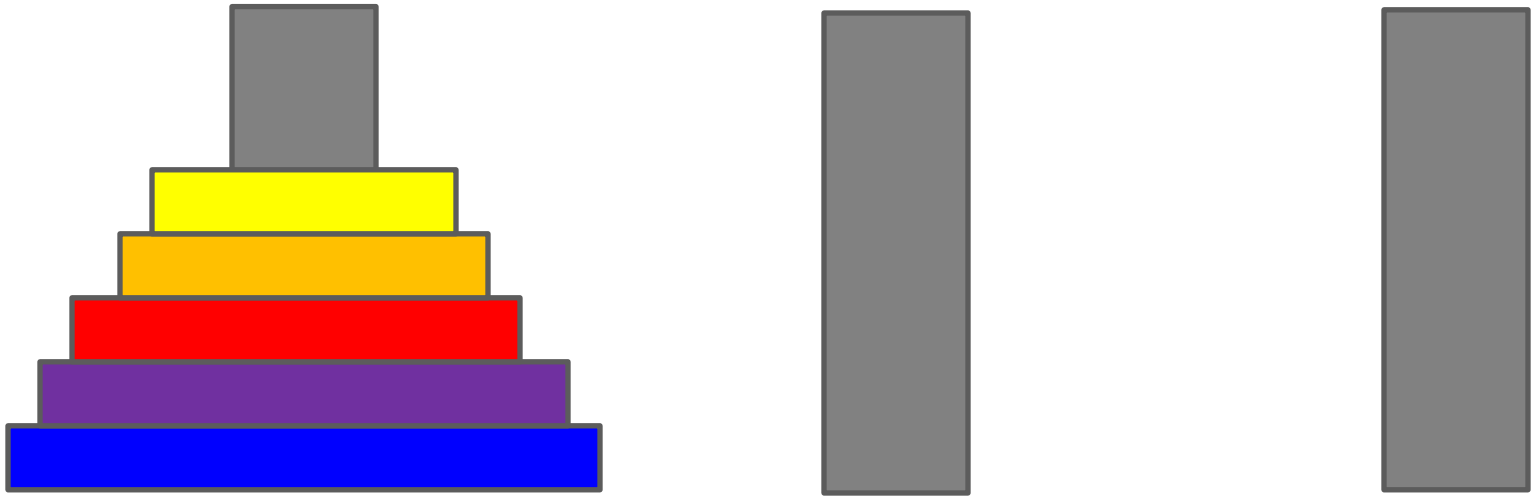
- Okay, how do we move 1 disc to the last peg?
  - Wait – that's easy!
  - We just move the disc!

Towers of Hanoi



- Okay, how do we move 1 disc to the last peg?
  - Wait – that's easy!
  - We just move the disc!

Towers of Hanoi



- So – what is our recursive structure here?
  - What is our base case?
  - What is our general case?

Towers Of Hanoi

- Let's write a recursive method for the Towers of Hanoi
  - The method should print out a series of steps like:

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

(These would be if we had two discs and wanted to move them to the middle peg)

```
public static void move(int disks, int from, int to, int spare)
```