Implementing Recursion

Based on slides by Prof. Burton Ma

Printing n of Something

- Suppose you want to implement a method that prints out *n* copies of a string

```
public static void printIt(String s, int n)
{
  for(int i = 0; i < n; i++)
  {
    System.out.print(s);
  }
}
```

A Different Solution

- Alternatively we can use the following algorithm:
  1. if n == 0 done, otherwise
     I. print the string once
     II. print the string (n – 1) more times

```
public static void printItToo(String s, int n)
{
  if (n == 0)
  {
    return;
  }
  else
  {
    System.out.print(s);
    printItToo(s, n - 1);    // method invokes itself
  }
}
```

Recursion

- A method that calls itself is called a *recursive* method
- A recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printIt("*", 5)
*printIt("*", 4)
**printIt("*", 3)
***printIt("*", 2)
****printIt("*", 1)
*****printIt("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Infinite Recursion

- If the base case(s) is missing, or never reached, a recursive method will run forever (or until the computer runs out of resources)

```
public static void printItForever(String s, int n)
{
  // missing base case; infinite recursion
  System.out.print(s);
  printItForever(s, n - 1);
}
```

```
printIt("*", 1)
* printIt("*", 0)
** printIt("*", -1)
*** printIt("*", -2) ...........
```

Fibonacci Numbers

- The sequence of additional pairs
  - **`0, 1, 1, 2, 3, 5, 8, 13, ...`**

  are called Fibonacci numbers

- Base cases
  - **`F(0) = 0`**
  - **`F(1) = 1`**
- Recursive definition
  - **`F(n) = F(n – 1) +  F(n – 2)`**

Recursive Methods & Return Values

- A recursive method can return a value
- Example: compute the nth Fibonacci number

```
public static int fibonacci(int n)
{
  if (n == 0)
  {
    return 0;
  }
  else if (n == 1)
  {
    return 1;
  }
  else
  {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

Recursive Methods & Return Values

- Example: write a recursive method **`countZeros`** that counts the number of zeros in an integer number **n**
  - **`10305060700002L`** has 8 zeros

- Trick: examine the following sequence of numbers

1. **`10305060700002`**
2. **`1030506070000`**
3. **`103050607000`**
4. **`10305060700`**
5. **`103050607`**
6. **`1030506 ...`**

Recursive Methods & Return Values

- Not Java:

```
countZeros(n) :
if the last digit in n is a zero
  return 1 + countZeros(n / 10)
else
  return countZeros(n / 10)
```

- Don't forget to establish the base case(s)
  - When should the recursion stop? when you reach a single digit (not zero digits; you never reach zero digits!)
    - Base case #1 : **`n == 0`**
      - **`return 1`**
    - Base case #2 : **`n != 0 && n < 10`**
      - **`return 0`**

```
public static int countZeros(long n)
{
  if(n == 0L)
  {  // base case 1
    return 1;
  }
  else if(n < 10L)
  {  // base case 2
    return 0;
  }
  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero)
  {
    return 1 + countZeros(m);
  }
  else
  {
    return countZeros(m);
  }
}
```

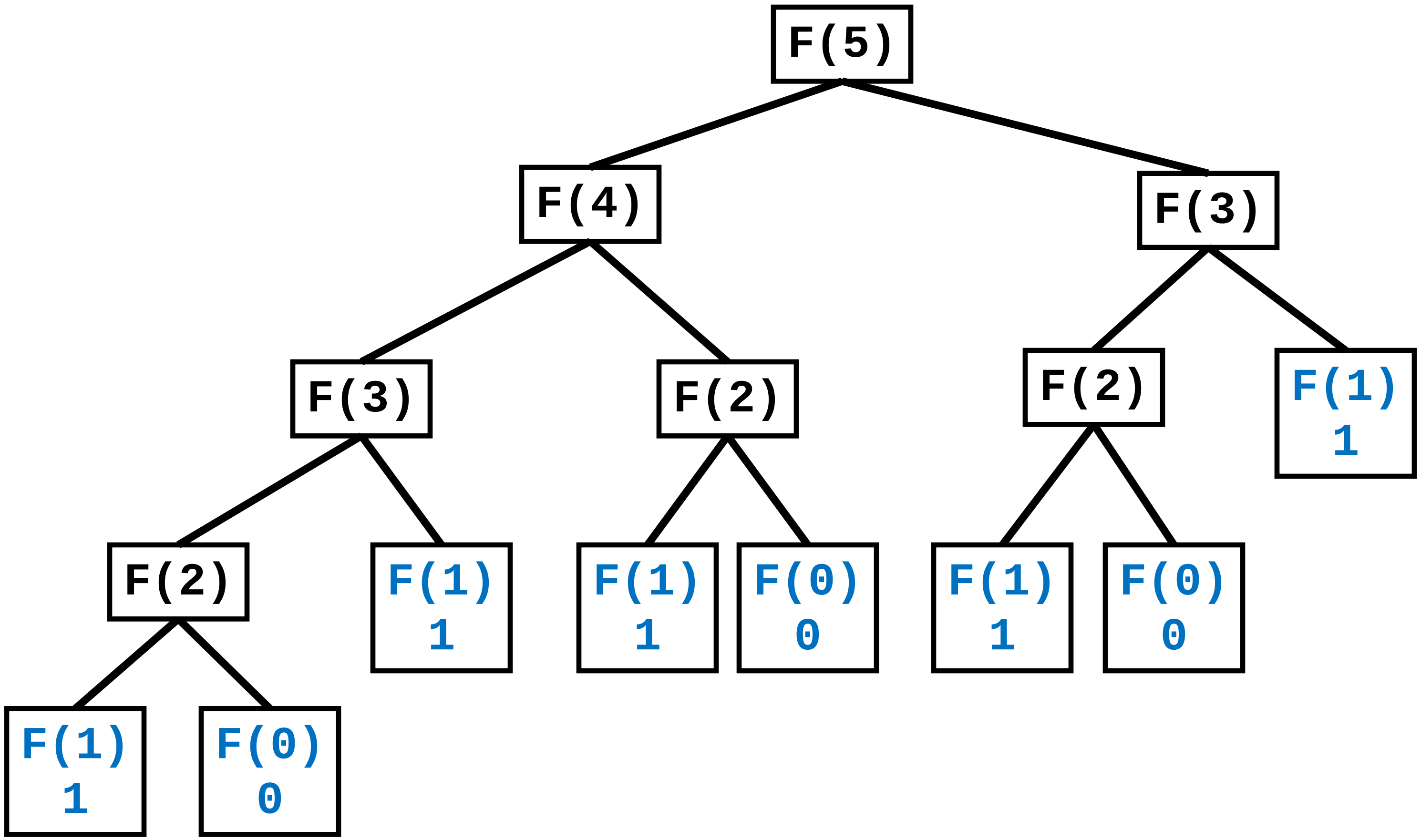
countZeros Call Stack

`callZeros( 800410L )`

last in first out

`callZeros( 8L )`	`0`
`callZeros( 80L )`	`1 + 0`
`callZeros( 800L )`	`1 + 1 + 0`
`callZeros( 8004L )`	`0 + 1 + 1 + 0`
`callZeros( 80041L )`	`0 + 0 + 1 + 1 + 0`
`callZeros( 800410L )`	`1 + 0 + 0 + 1 + 1 + 0`
	`= 3`

Fibonacci Call Tree



Compute Powers of 10

- Write a recursive method that computes $\mathbf{10^n}$ for any integer value **n**
- Recall:
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$
  - $10^{-n} = 1 / 10^n$

```
public static double powerOf10(int n)
 {
  if (n == 0)
  {
    // base case
    return 1.0;
  }
  else if (n > 0)
  {
    // recursive call for positive n
    return 10.0 * powerOf10(n - 1);
  }
  else
  {
    // recursive call for negative n
    return 1.0 / powerOf10(-n);
  }
}
```

Proving Correctness and Termination

- To show that a recursive method accomplishes its goal you must prove:
  1. That the base case(s) and the recursive calls are correct
  2. That the method terminates

Proving Correctness

- To prove correctness:
  1. Prove that each base case is correct
  2. Assume that the recursive invocation is correct and then prove that each recursive case is correct

printItToo

```
public static void printItToo(String s, int n)
 {
  if (n == 0)
  {
    return;
  }
  else
  {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

Correctness of printItToo

1. (prove the base case) If `n == 0` nothing is printed; thus the base case is correct.
2. Assume that `printItToo(s, n-1)` prints the string `s` exactly `(n – 1)` times. Then the recursive case prints the string `s` exactly `(n – 1)+1 = n` times; thus the recursive case is correct.

Proving Termination

- To prove that a recursive method terminates:
  1. Define the size of a method invocation; the size must be a non-negative integer number
  2. Prove that each recursive invocation has a smaller size than the original invocation

Termination of printIt

1. **`printIt(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printIt(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printIt(s, n-1)`** is **`n-1`** (by definition) which is smaller than the original size **`n`**.

countZeros

```
public static int countZeros(long n)
{
  if(n == 0L)
  {  // base case 1
    return 1;
  }
  else if(n < 10L)
  {  // base case 2
    return 0;
  }
  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero)
  {
    return 1 + countZeros(m);
  }
  else
  {
    return countZeros(m);
  }
}
```

Correctness of countZeros

1. (Base cases) If the number has only one digit then the method returns **`1`** if the digit is zero and **`0`** if the digit is not zero; therefore, the base case is correct.
2. (Recursive cases) Assume that **`countZeros(n/10L)`** is correct (it returns the number of zeros in the first **`(d – 1)`** digits of **`n`**). If the last digit in the number is zero, then the recursive case returns **`1 +`** the number of zeros in the first **`(d – 1)`** digits of **`n`**, otherwise it returns the number of zeros in the first **`(d – 1)`** digits of **`n`**; therefore, the recursive cases are correct.

Termination of countZeros

1. Let the size of **`countZeros(n)`** be **`d`** the number of digits in the number **`n`**.
2. The size of the recursive invocation **`countZeros(n/10L)`** is **`d-1`**, which is smaller than the size of the original invocation.

Decrease and Conquer

- A common strategy for solving computational problems
  - Solves a problem by taking the original problem and converting it to *one* smaller version of the same problem
    - Note the similarity to recursion
- Decrease and conquer, and the closely related divide and conquer method, are widely used in computer science
  - Allow you to solve certain complex problems easily
  - Help to discover efficient algorithms

Review of Recursion

- A recursive method calls itself
- To prevent infinite recursion you need to ensure that:
  1. The method reaches a base case
  2. Each recursive call makes progress towards a base case (i.e. reduces the size of the problem)
- To solve a problem with a recursive algorithm:
  1. Identify the base cases (the cases corresponding to the smallest version of the problem you are trying to solve)
  2. Figure out the recursive call(s)

Correctness and Termination

- Proving correctness requires that you do two things:
  1. Prove that each base case is correct
  2. Assume that the recursive invocation is correct and then prove that each recursive case is correct
- Proving termination requires that you do two things:
  1. Define the size of each method invocation
  2. Prove that each recursive invocation is smaller than the original invocation

Recursion Examples

- The subsequent slides present additional examples of problems that can be solved using recursion

- Depending on time, these examples may or may not be discussed in lecture.

Palindromes

1. A palindrome is a sequence of symbols that is the same forwards and backwards:
   - "level"
   - "yo banana boy"

Write a recursive algorithm that returns true if a string is a palindrome (and false if not); assume that the string has no spaces or punctuation marks.

Palindromes

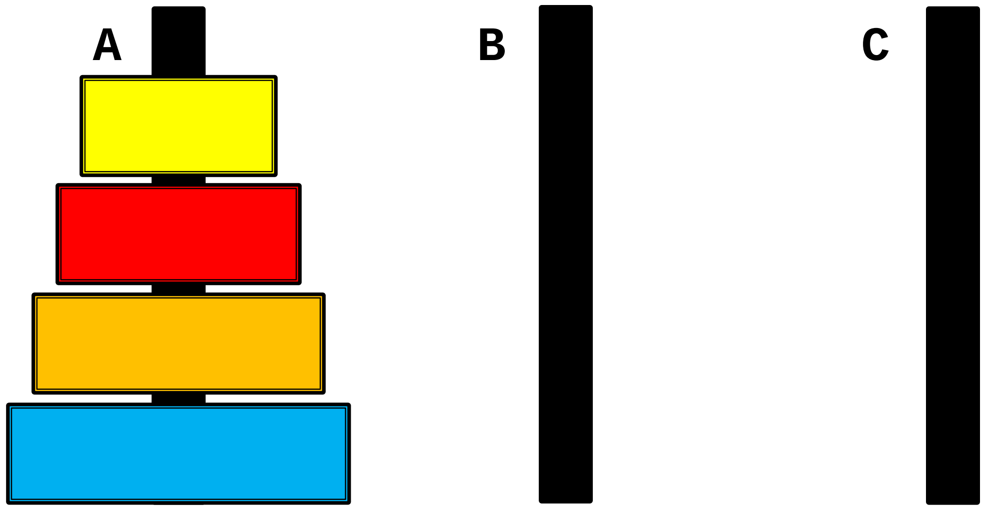
- Sketch a small example of the problem
  - It will help you find the base cases
  - It might help you find the recursive cases

Palindromes

```
public static boolean isPalindrome(String s)
 {
  if (s.length() < 2)
  {
    return true;
  }
  else
  {
    int first = 0;
    int last = s.length() - 1;
    return (s.charAt(first) == s.charAt(last)) &&
      isPalindrome(s.substring(first + 1, last));
  }
}
```

Towers of Hanoi

3. [AJ, p 685, Q7]



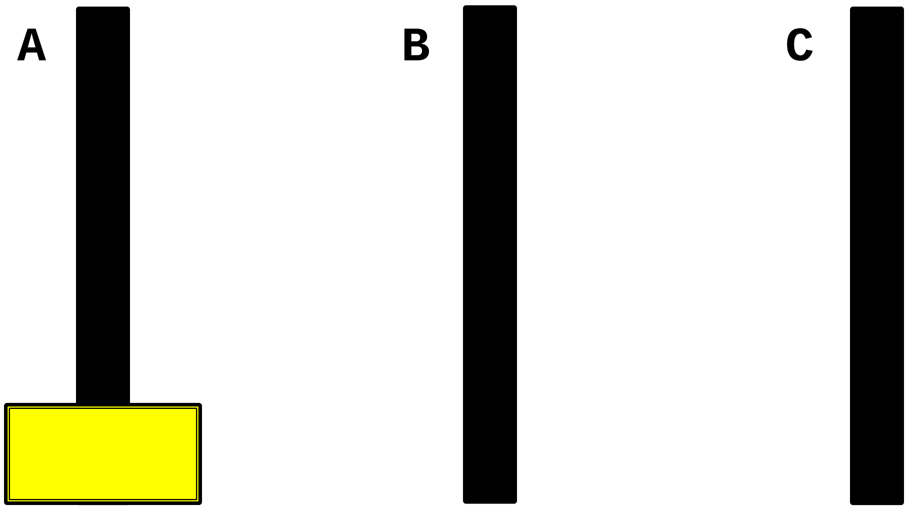
- Move the stack of *n* disks from A to C
  - Can move one disk at a time from the top of one stack onto another stack
  - Cannot move a larger disk onto a smaller disk

Towers of Hanoi

- Legend says that the world will end when a 64 disk version of the puzzle is solved
- Several appearances in pop culture
  - Doctor Who (TV series)
  - Rise of the Planet of the Apes (Movie)
  - Mass Effect (Video game)

Towers of Hanoi

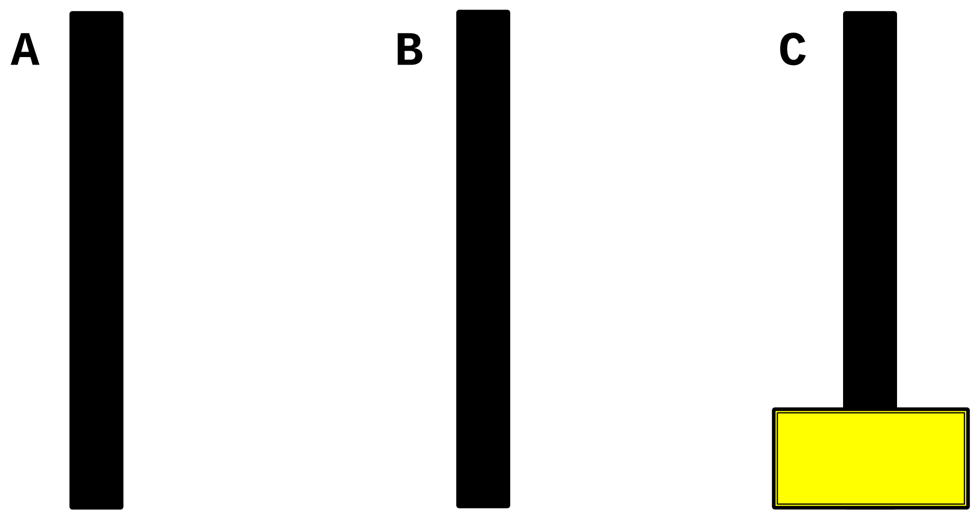
- $n = 1$



- Move disk from A to C

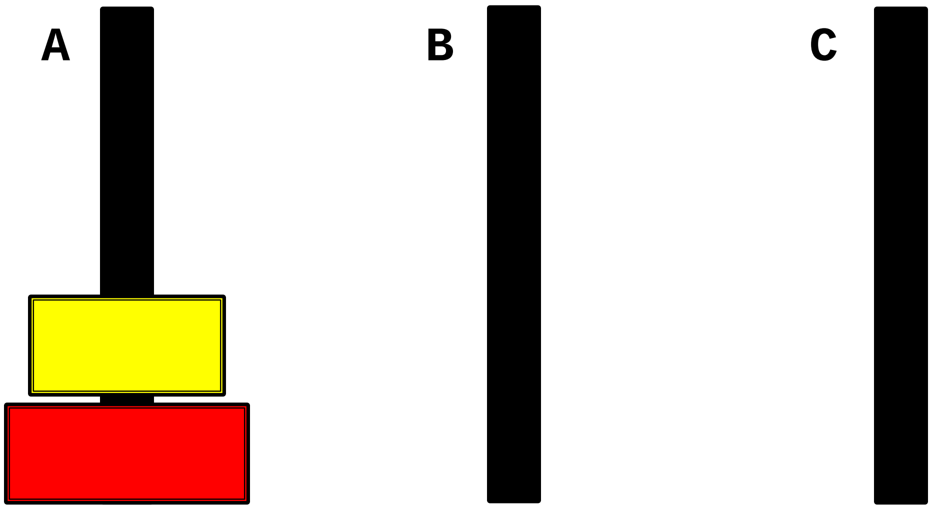
Towers of Hanoi

- $n = 1$



Towers of Hanoi

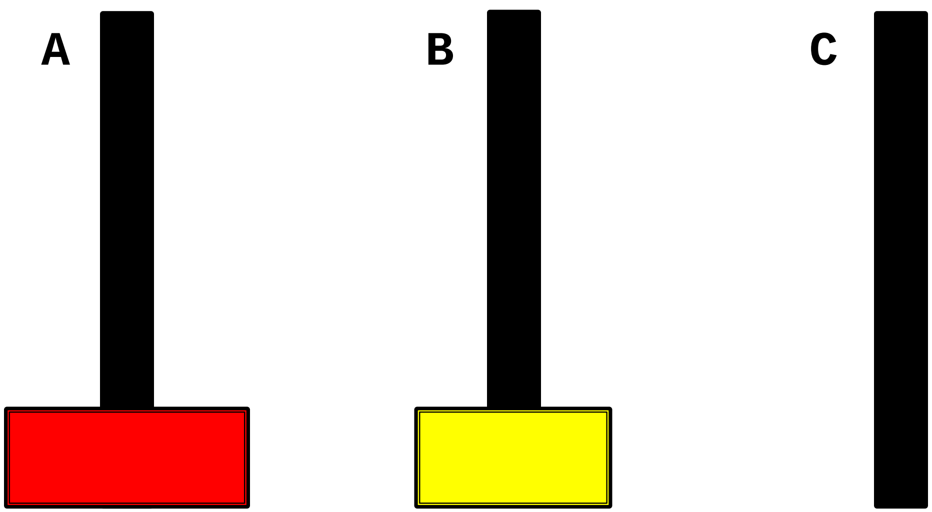
- $n = 2$



- Move disk from A to B

Towers of Hanoi

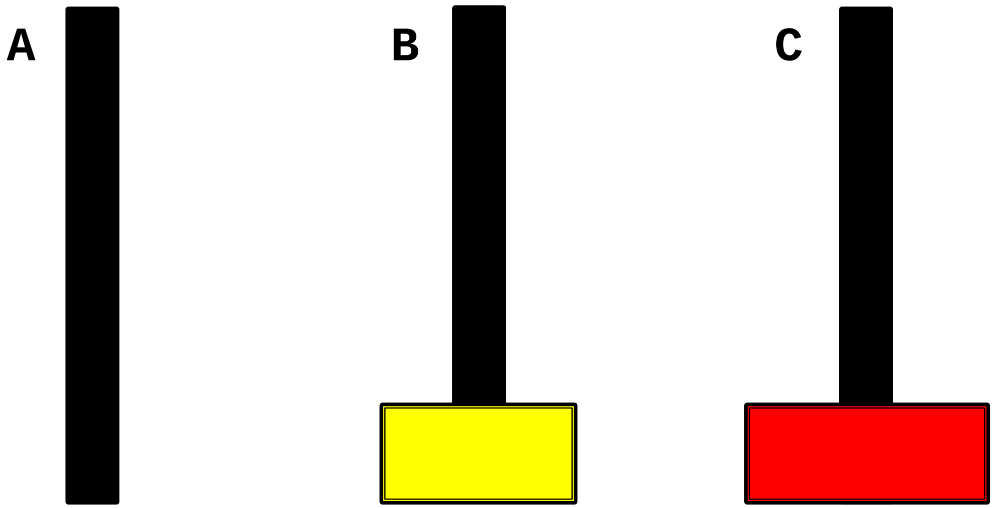
- $n = 2$



- Move disk from A to C

Towers of Hanoi

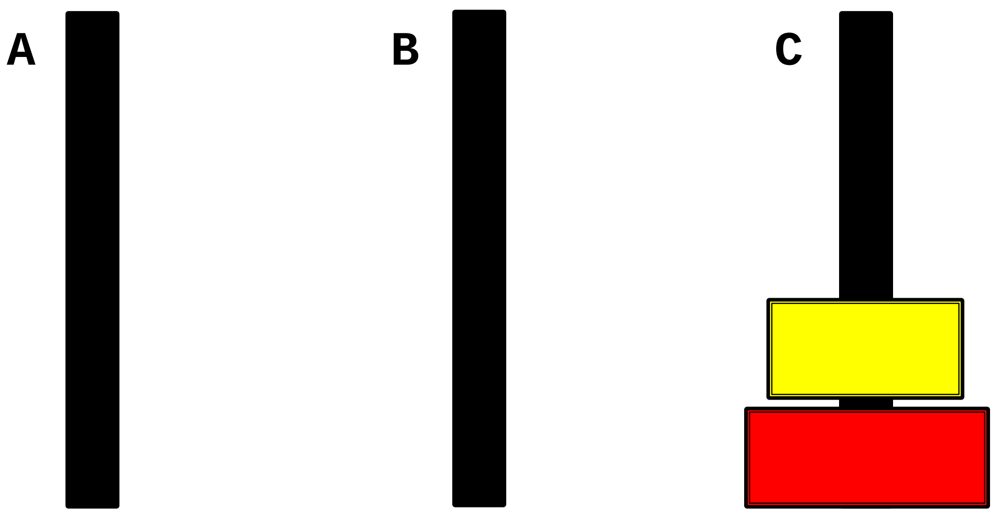
- $n = 2$



- Move disk from B to C

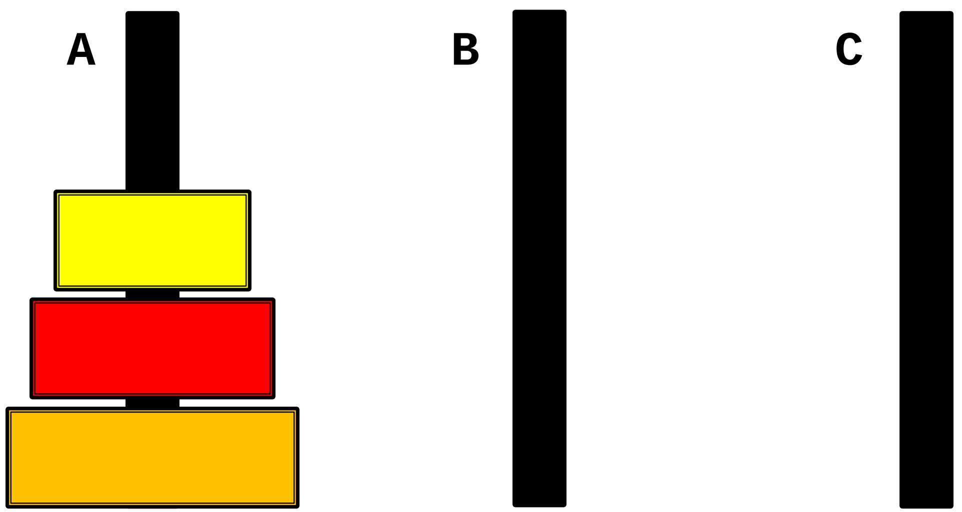
Towers of Hanoi

- $n = 2$



Towers of Hanoi

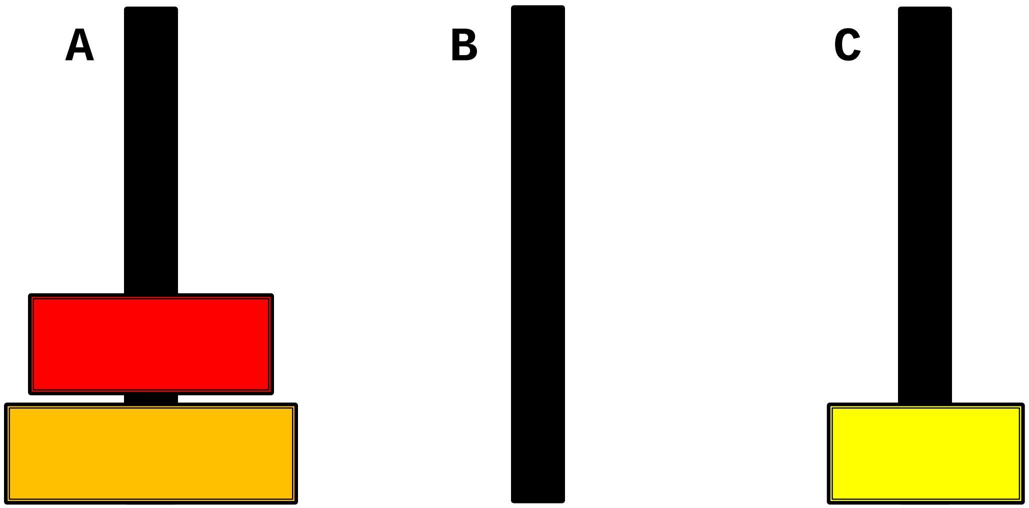
- $n = 3$



- Move disk from A to C

Towers of Hanoi

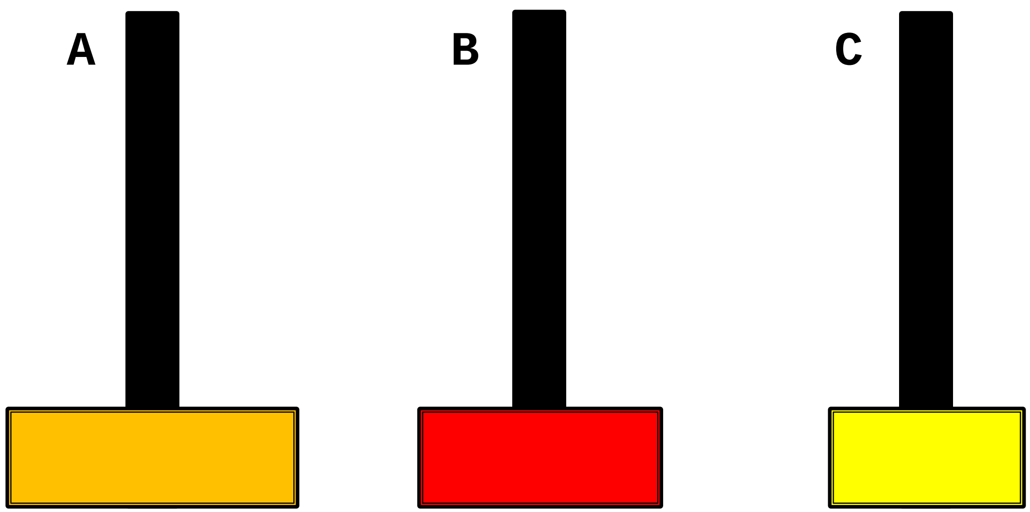
- $n = 3$



- Move disk from A to B

Towers of Hanoi

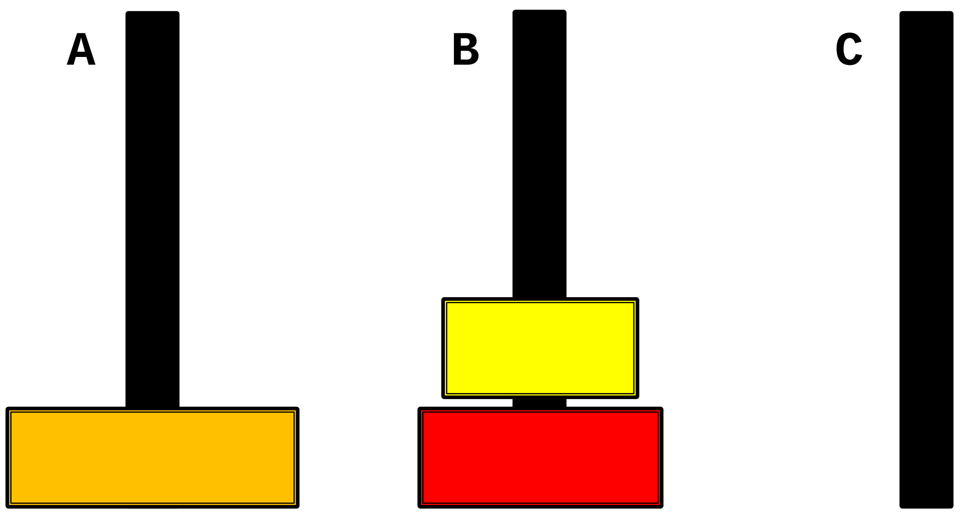
- $n = 3$



- Move disk from C to B

Towers of Hanoi

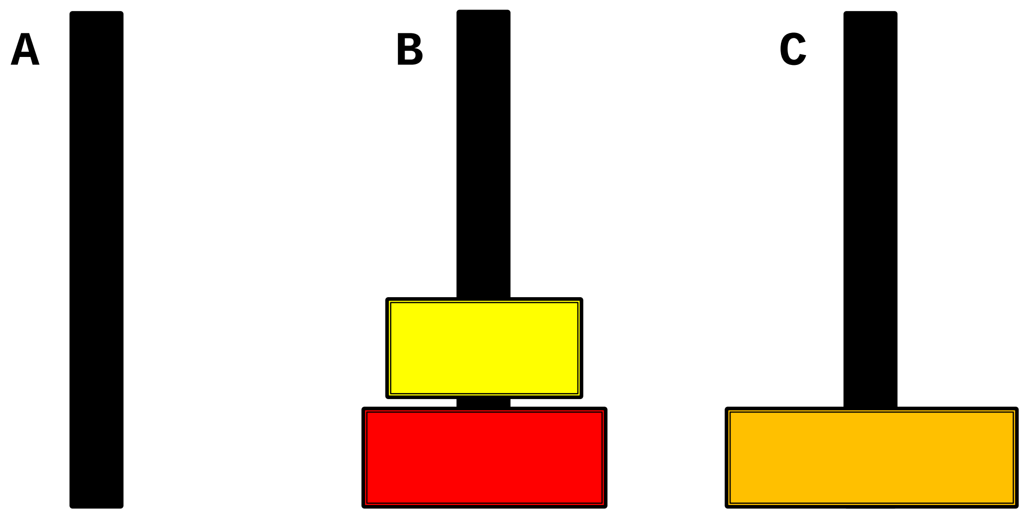
- $n = 3$



- Move disk from A to C

Towers of Hanoi

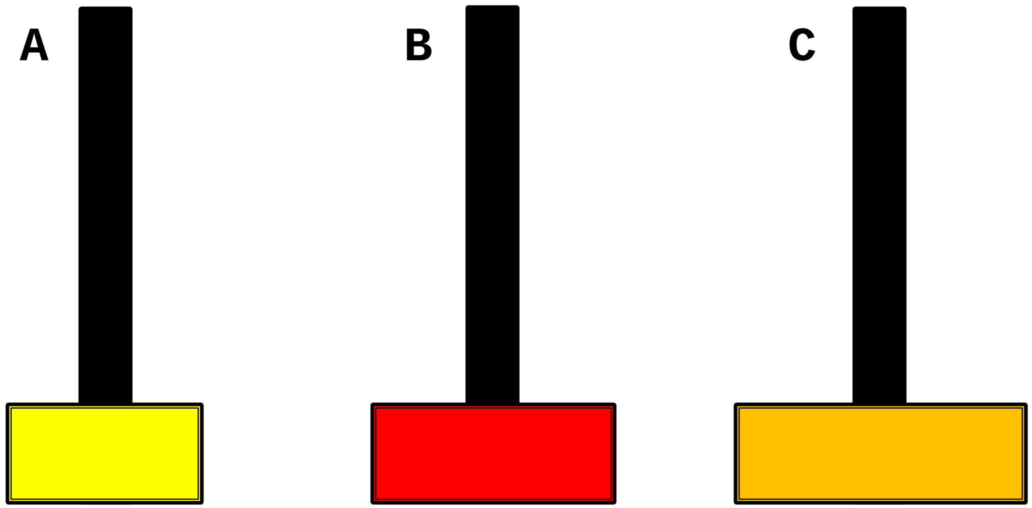
- $n = 3$



- Move disk from B to A

Towers of Hanoi

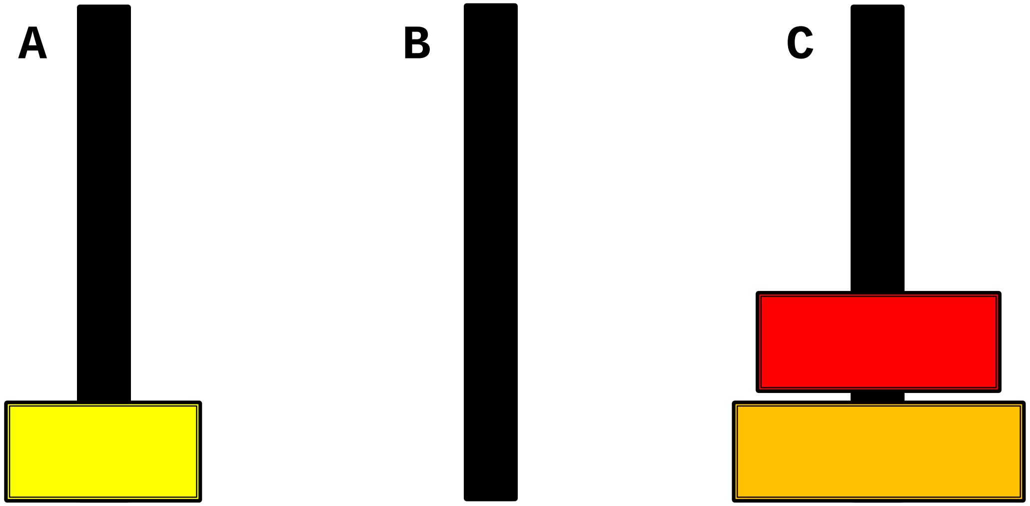
- $n = 3$



- Move disk from B to C

Towers of Hanoi

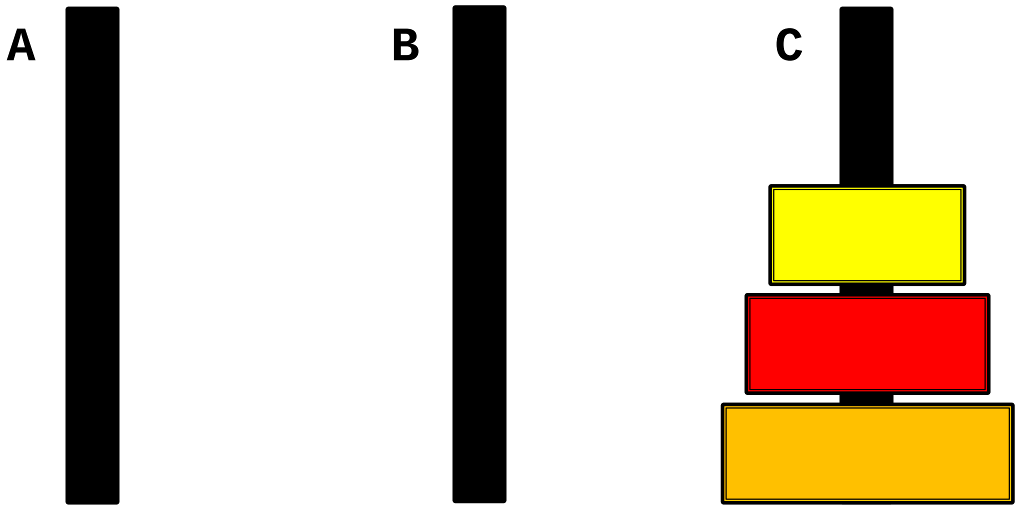
- $n = 3$



- Move disk from A to C

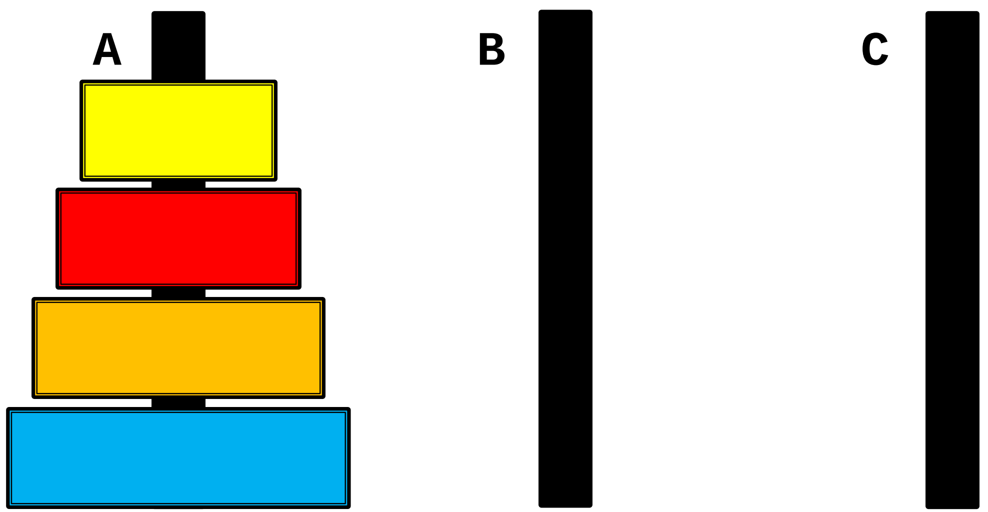
Towers of Hanoi

- $n = 3$



Towers of Hanoi

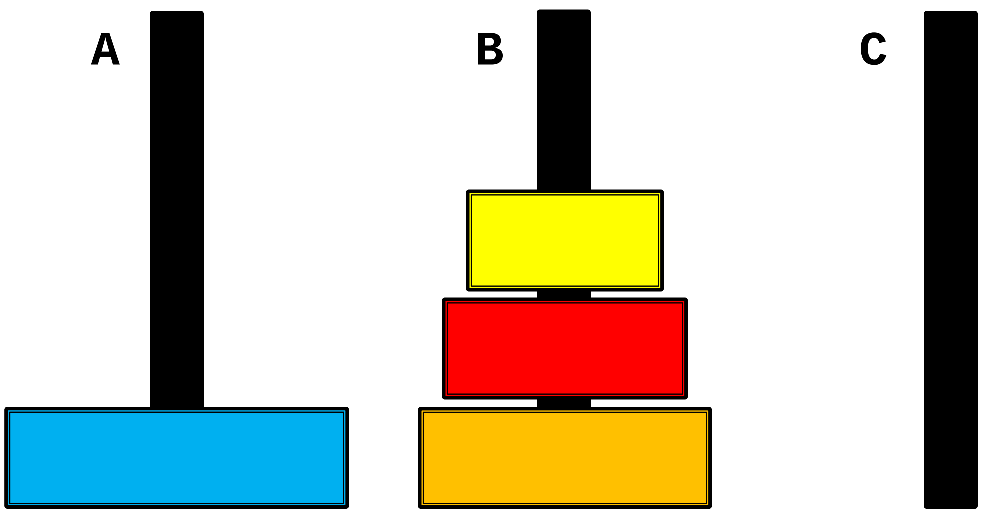
- $n = 4$



- Move (n – 1) disks from A to B using C

Towers of Hanoi

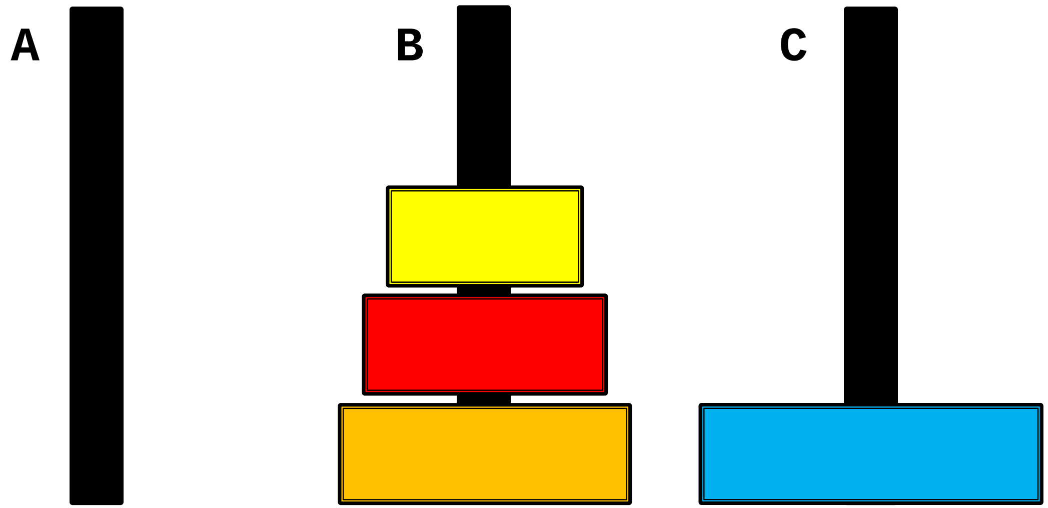
- $n = 4$



- Move disk from A to C

Towers of Hanoi

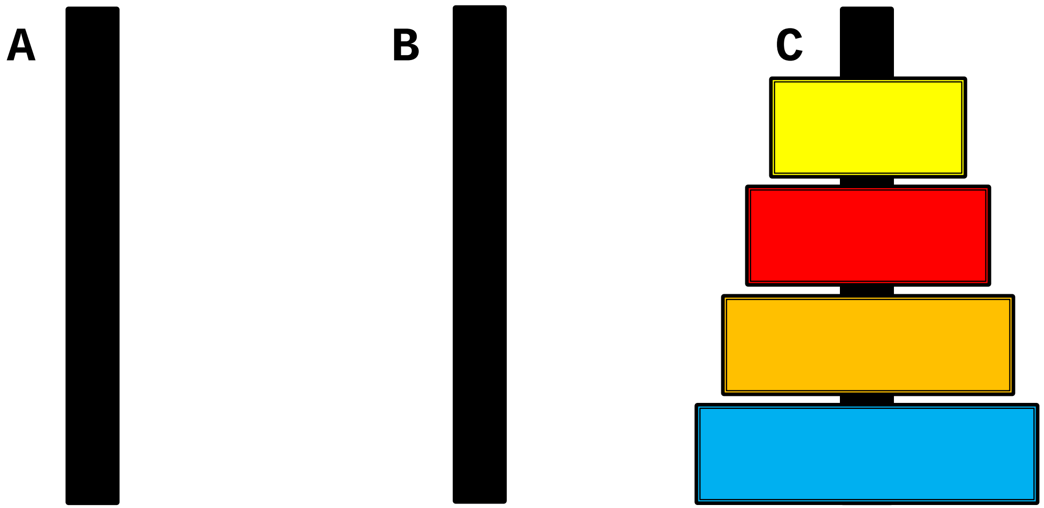
- $n = 4$



- Move (n – 1) disks from B to C using A

Towers of Hanoi

- $n = 4$



- Base case $n = 1$
  1. Move disk from A to C
- Recursive case
  1. Move $(n - 1)$ disks from A to B
  2. Move 1 disk from A to C
  3. Move $(n - 1)$ disks from B to C

Towers of Hanoi

```
public static void move(int n,
                        String from,
                        String to,
                        String using)
{
  if(n == 1)
  {
    System.out.println("move disk from " + from + " to " + to);
  }
  else
  {
    move(n - 1, from, using, to);
    move(1, from, to, using);
    move(n - 1, using, to, from);
  }
}
```