Recursion (Part 3)

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)

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));
  }
}
```

Jump It

2. [AJ, p 685, Q4]

0	3	80	6	57	10

- board of n squares, $n > 2$
- start at the first square on left
- on each move you can move 1 or 2 squares to the right
- each square you land on has a cost (the value in the square)
  - costs are always positive
- goal is to reach the rightmost square with the lowest cost

Jump It

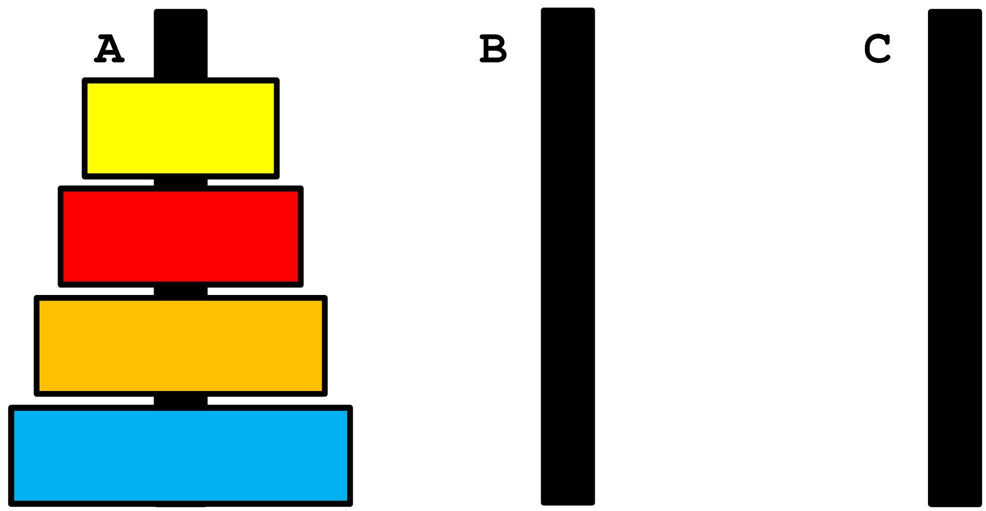
- sketch a small example of the problem
  - it will help you find the base cases
  - it might help you find the recursive cases

Jump It

```
public static int cost(List<Integer> board) {
  if (board.size() == 2) {
    return board.get(0) + board.get(1);
  }
  if (board.size() == 3) {
    return board.get(0) + board.get(2);
  }
  List<Integer> afterOneStep = board.subList(1, board.size());
  List<Integer> afterTwoStep = board.subList(2, board.size());
  int c = board.get(0);
  return c + Math.min(cost(afterOneStep), cost(afterTwoStep));
}
```

Towers of Hanoi

3. [AJ, p 685, Q7]



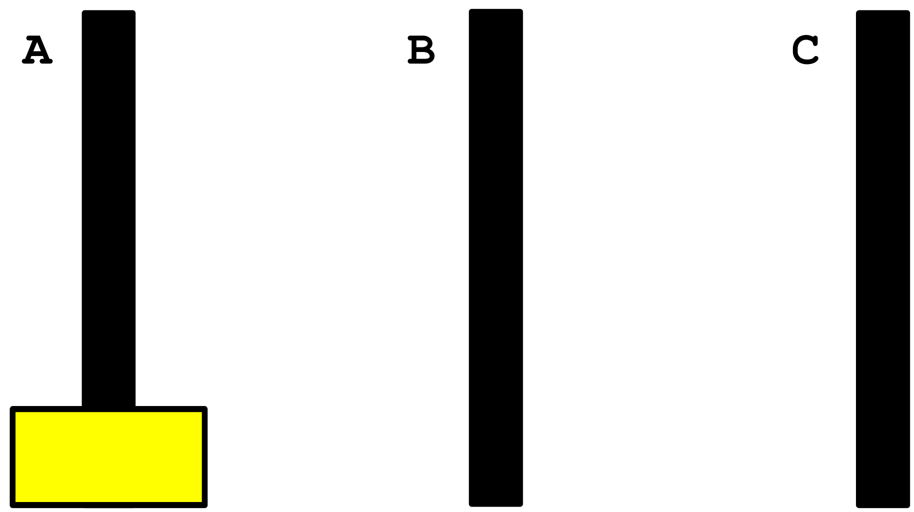
- move the stack of *n* disks from A to C
  - can move one disk at a time from the top of one stack onto another stack
  - cannot move a larger disk onto a smaller disk

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
  - Rise of the Planet of the Apes
  - Survior: South Pacific

Towers of Hanoi

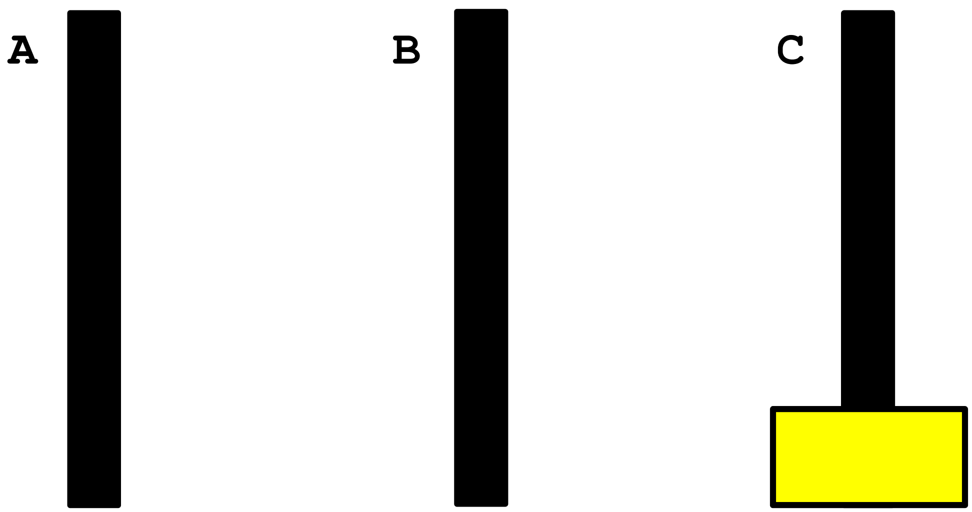
- n = 1



- move disk from A to C

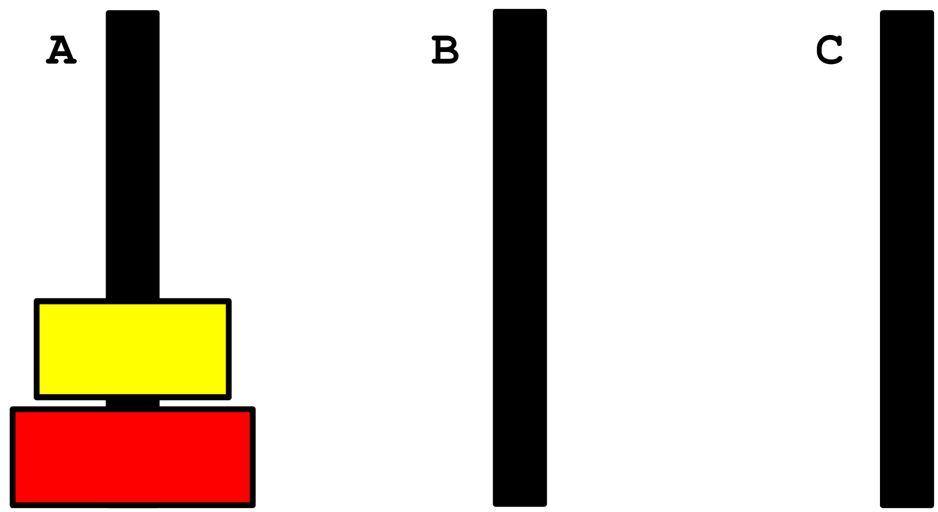
Towers of Hanoi

- n = 1



Towers of Hanoi

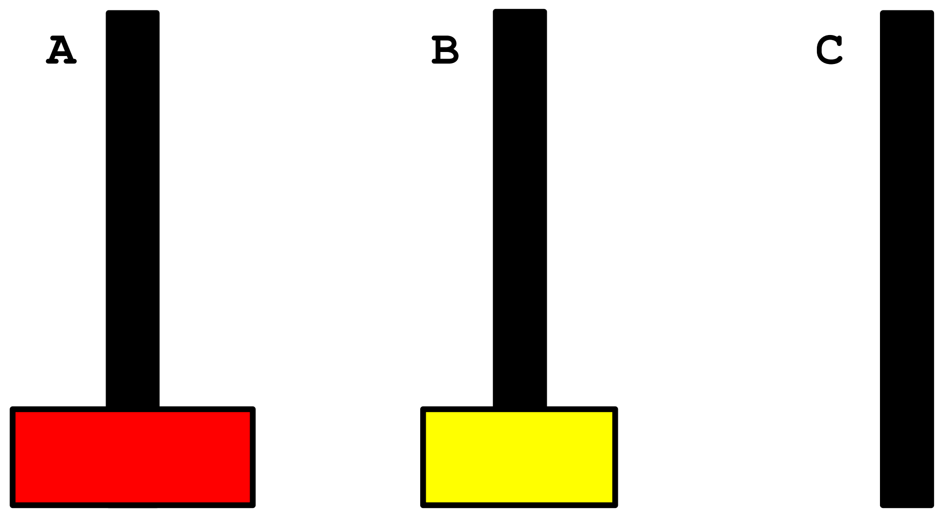
- n = 2



- move disk from A to B

Towers of Hanoi

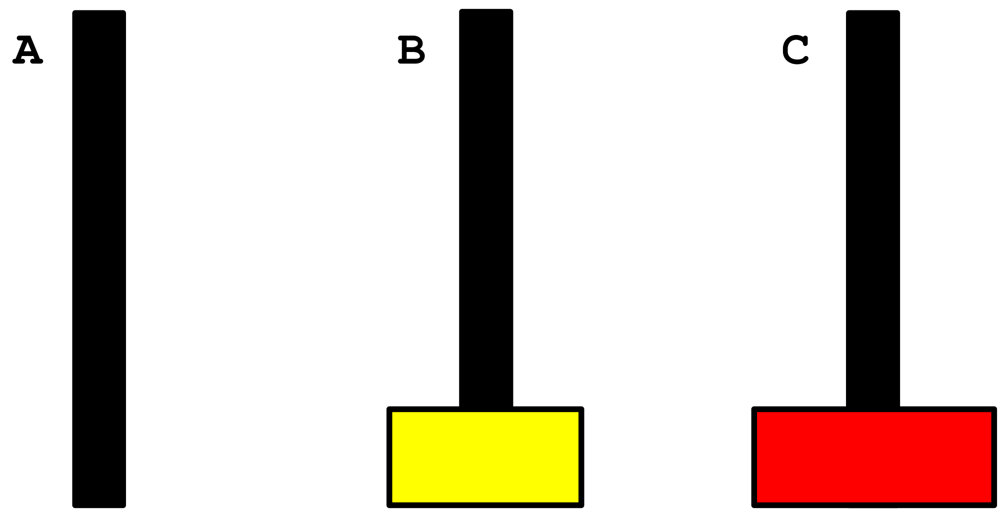
- n = 2



- move disk from A to C

Towers of Hanoi

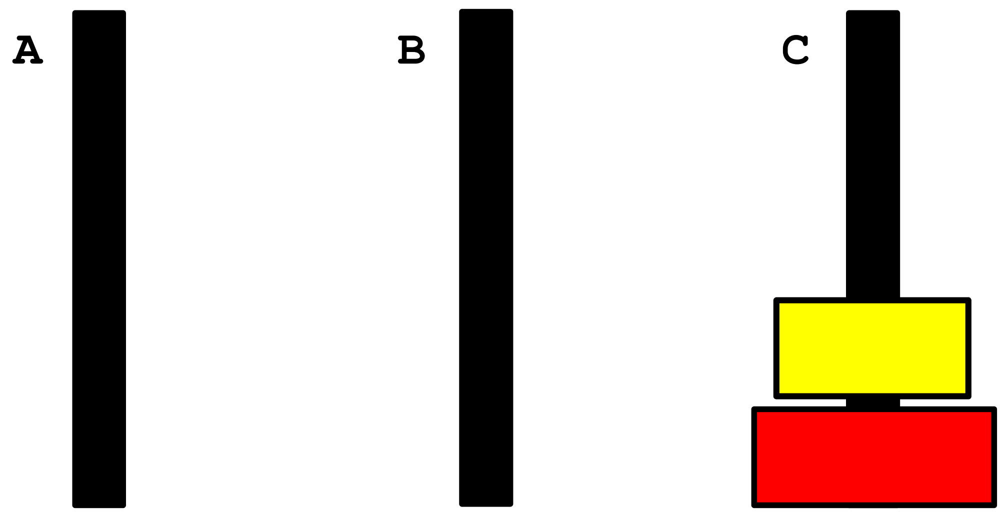
- n = 2



- move disk from B to C

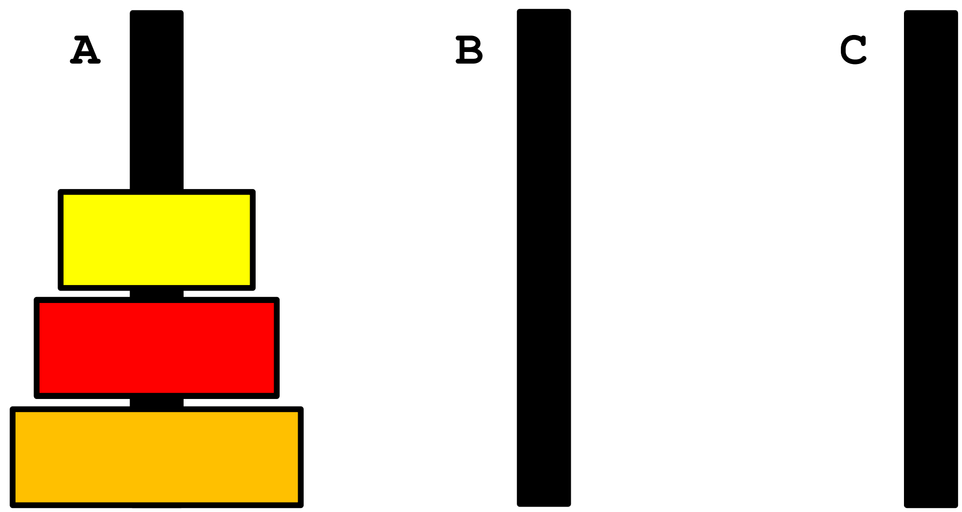
Towers of Hanoi

- n = 2



Towers of Hanoi

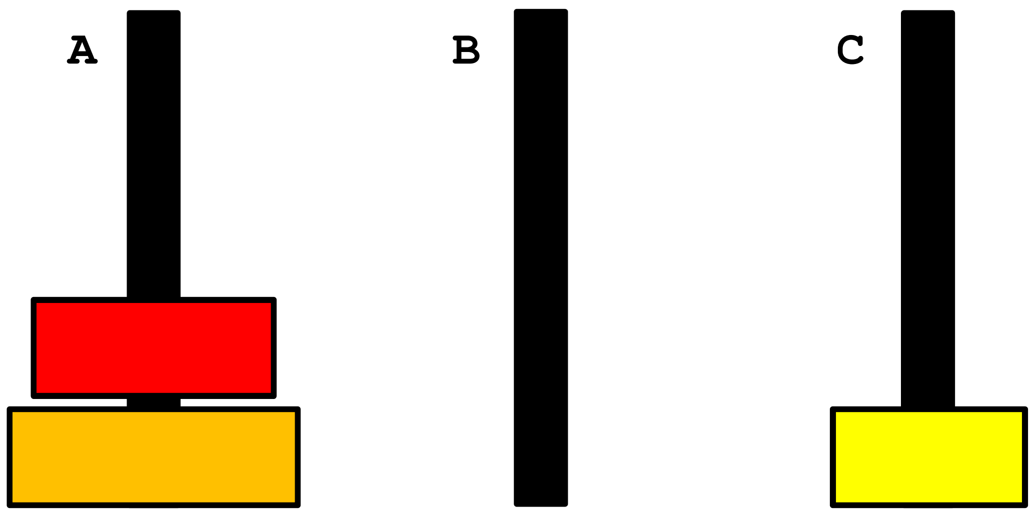
- n = 3



- move disk from A to C

Towers of Hanoi

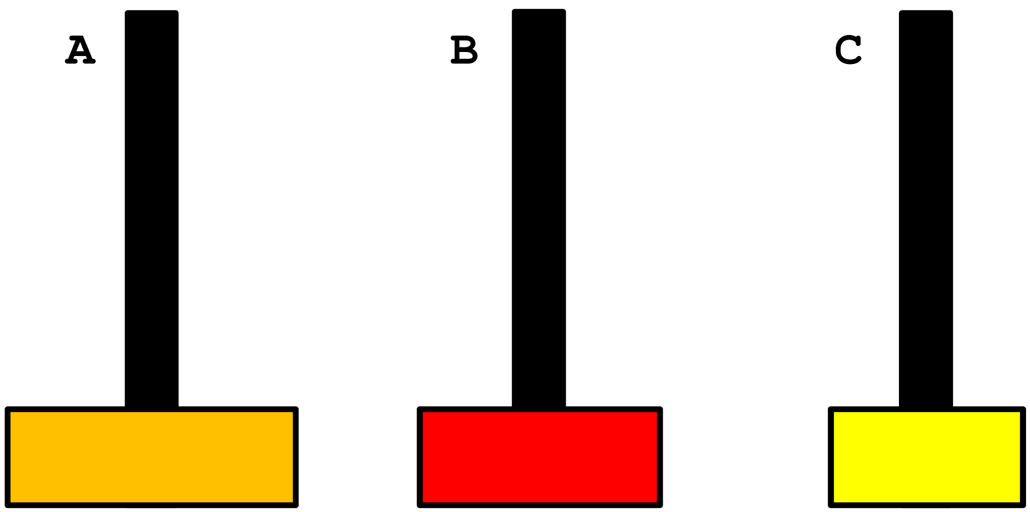
- n = 3



- move disk from A to B

Towers of Hanoi

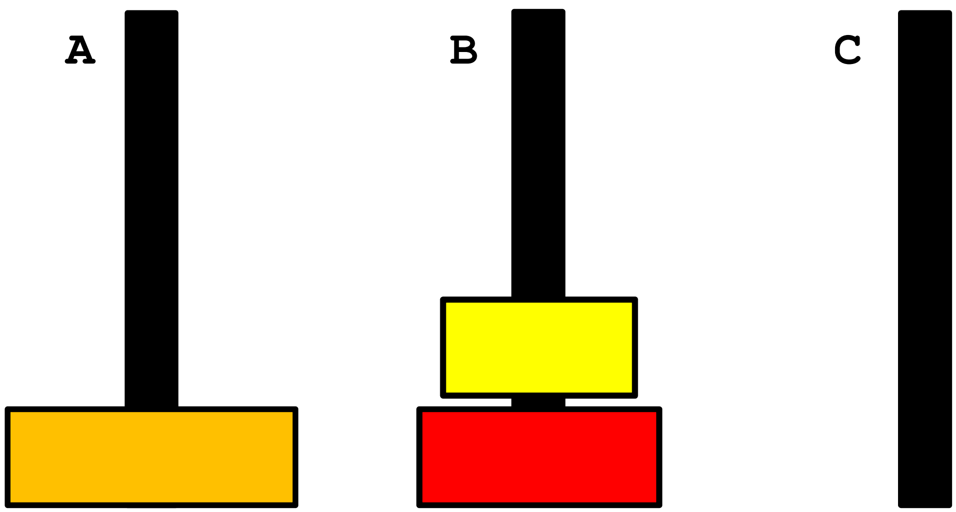
- n = 3



- move disk from C to B

Towers of Hanoi

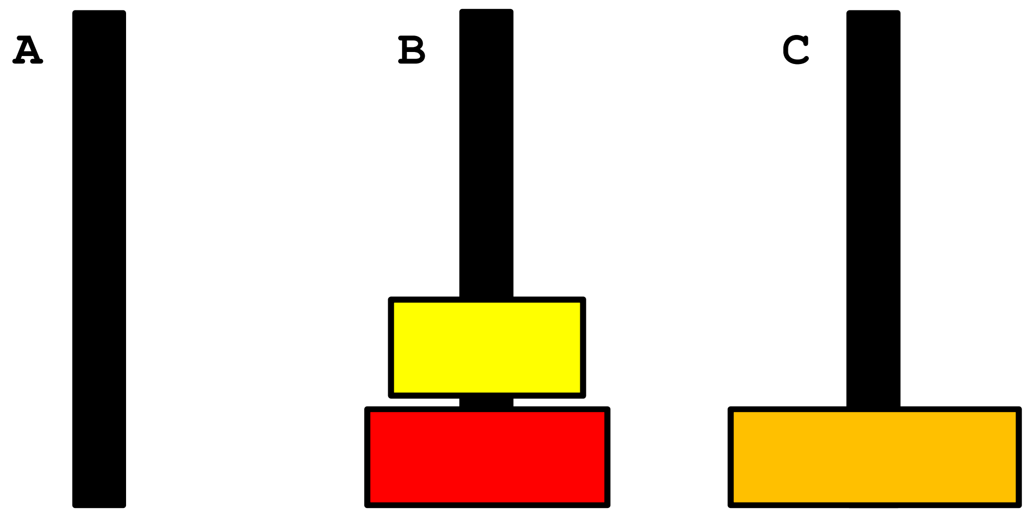
- n = 3



- move disk from A to C

Towers of Hanoi

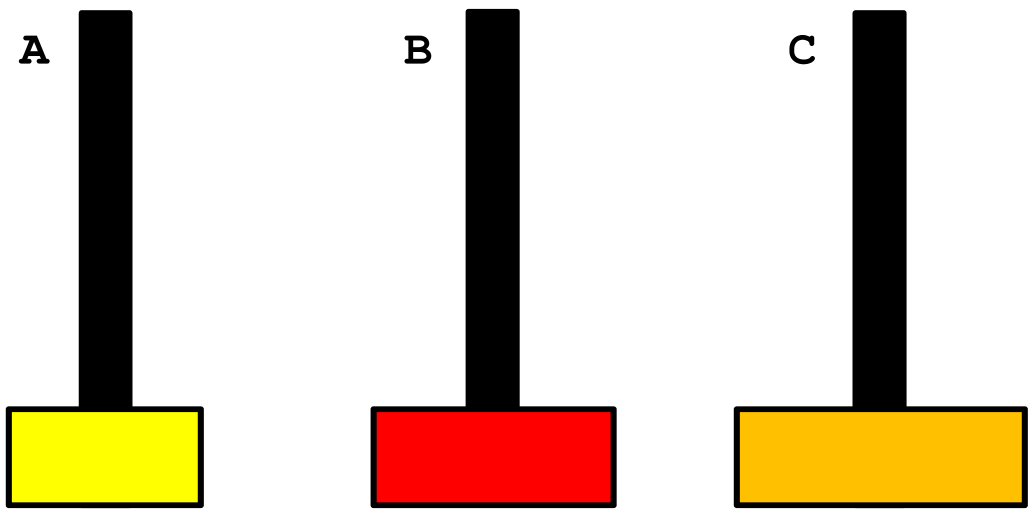
- n = 3



- move disk from B to A

Towers of Hanoi

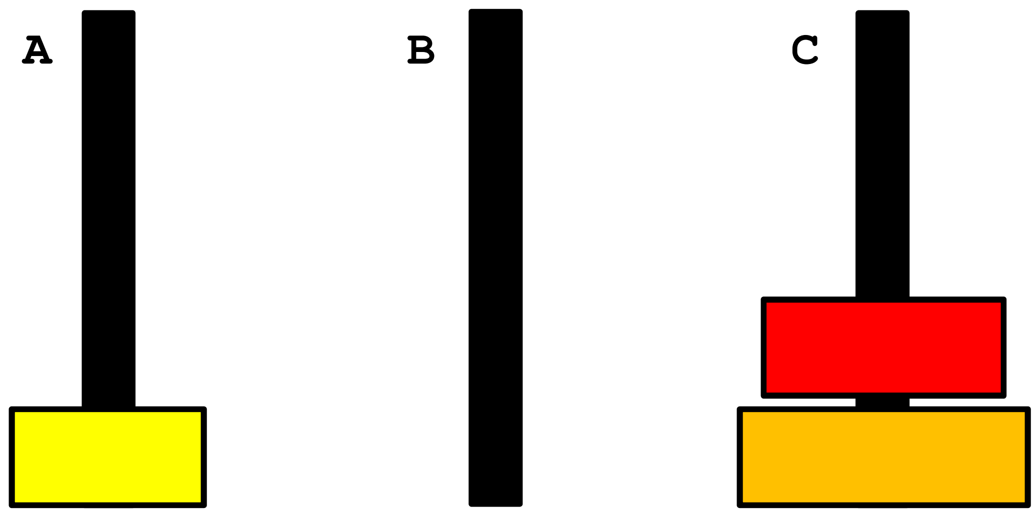
- n = 3



- move disk from B to C

Towers of Hanoi

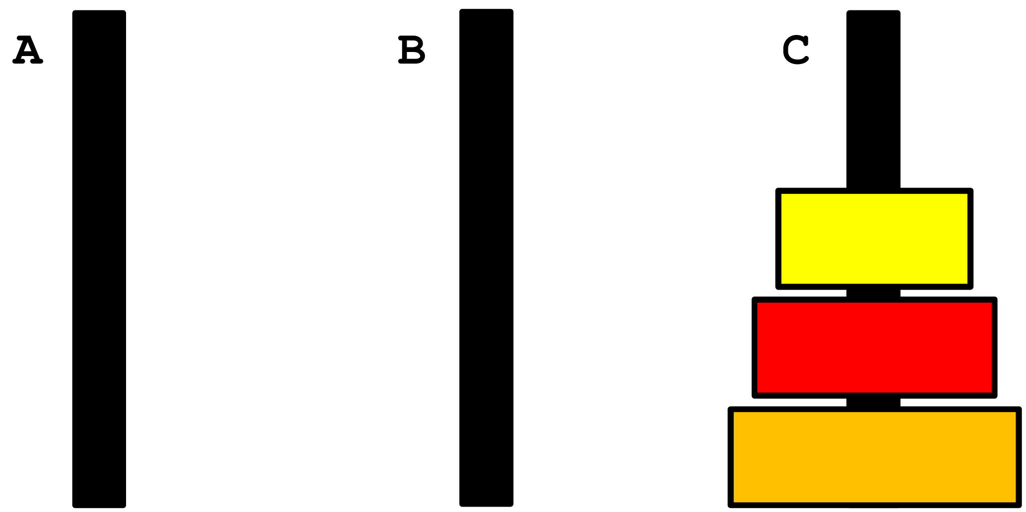
- n = 3



- move disk from A to C

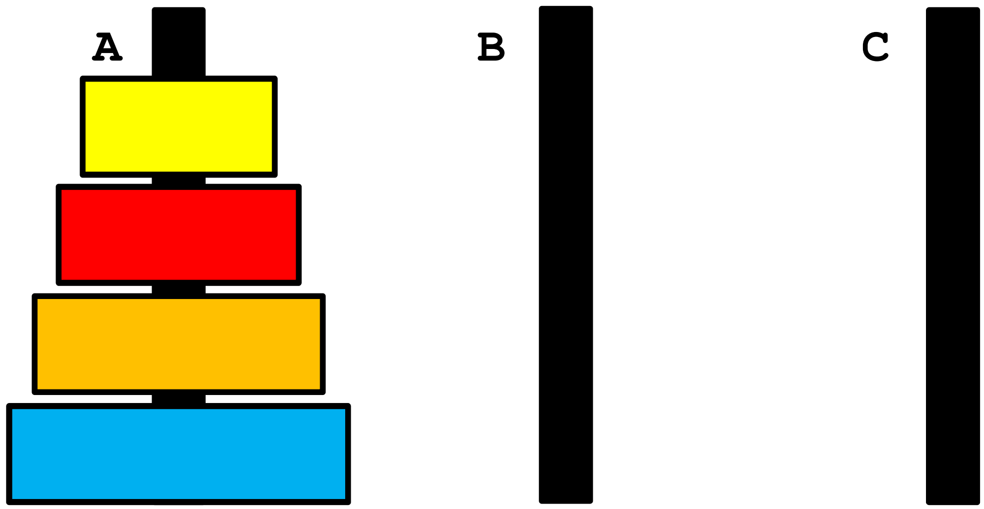
Towers of Hanoi

- n = 3



Towers of Hanoi

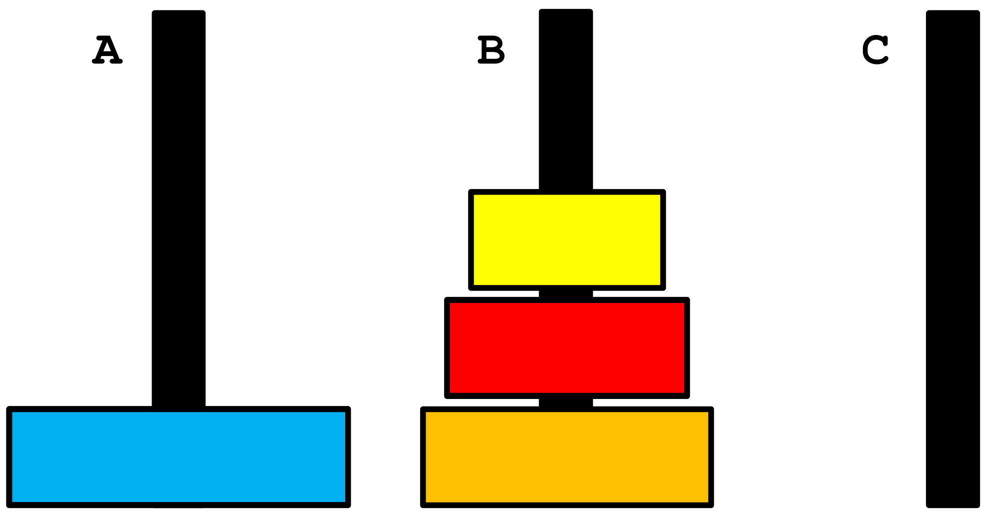
- n = 4



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

Towers of Hanoi

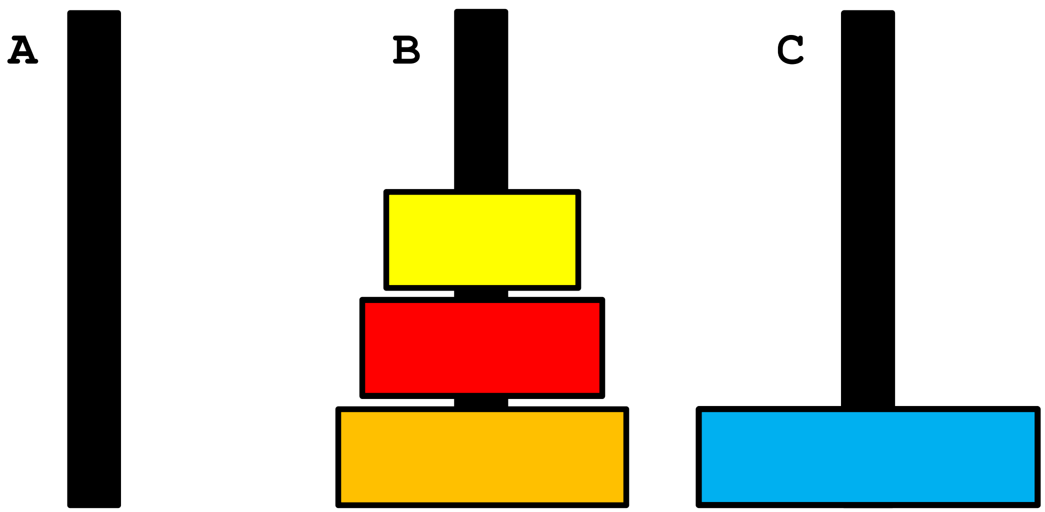
- n = 4



- move disk from A to C

Towers of Hanoi

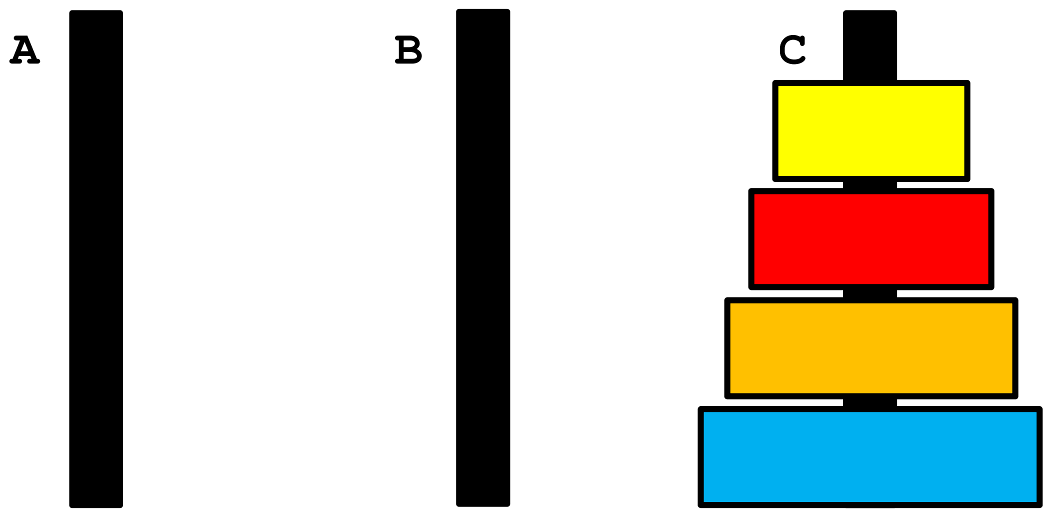
- n = 4



- 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);
  }
}
```

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

4. Prove that the recursive palindrome algorithm is correct and terminates.
5. Prove that the recursive Jump It algorithm is correct and terminates.
6. Prove the recursive Towers of Hanoi algorithm is correct and terminates.