

CSC148 – Introduction to Computer Science

Lecture 5: Recursion

Sean Henderson

What is Recursion?

[Advanced Search](#)

Web [+ Show options...](#)

Results 1 - 10 of about

Did you mean: [recursion](#)

[Recursion - Wikipedia, the free encyclopedia](#)

A visual form of **recursion** known as the Droste effect. The woman in this image is holding an object which contains a smaller image of her holding the same ...

en.wikipedia.org/wiki/Recursion - [Cached](#) - [Similar](#) -   

[Recursion \(computer science\) - Wikipedia, the free encyclopedia](#)

Recursion in computer science is a way of thinking about and solving many types of problems. In fact, **recursion** is one of the central ideas of computer ...

[en.wikipedia.org/wiki/Recursion_\(computer_science\)](http://en.wikipedia.org/wiki/Recursion_(computer_science)) - [Cached](#) - [Similar](#) -   

[+ Show more results from en.wikipedia.org](#)

[Recursion -- from Wolfram MathWorld](#)

A **recursive** process is one in which objects are defined in terms of other objects of the same type. Using some sort of recurrence relation, the entire class ...

mathworld.wolfram.com/Recursion.html - [Cached](#) - [Similar](#) -   

What is Recursion?

- A function (or process, or data structure...) defined in terms of itself
- Two important components
 - A Base Case which terminates the recursion
 - A Recursive Step which moves toward the base case

Example

- Say we want to calculate $n!$
- Recall: $n! = 1 * 2 * 3 * \dots * (n-1) * n$
- Easy enough to do

```
def f(n):  
    fact = 1  
    for i in range(n):  
        fact = fact * (i + 1)  
    return fact
```

Example

- Let's say we wanted to calculate $n!$ recursively
- Base cases?
- Recursive call?

Example

- Defined base case $n = 0$
- Recursive call $n! = n * (n - 1)!$

```
def f(n):  
    if n == 0:  
        # Base case  
        return 1  
    else:  
        # Recursive call  
        return n * f(n - 1)
```

Example

- Fibonacci numbers; always seem to have a way of showing up
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- $F_n = F_{n-1} + F_{n-2}$
- $F_0 = 1$
- $F_1 = 1$
- Every term is the sum of the two previous terms

Why Recursion?

- Also very easy to implement with a for loop
- Are there any problems where recursion is necessary, and not just masking a loop?
- Not really
- Recursion is exactly as powerful as a Stack (although significantly simpler in most cases)

More Meaningful Example

- Exponentiation: compute a^b
 - non-negative integer b
- Simple solution using for loop

```
def power(a, b):  
    ret = 1  
    for i in range(b):  
        ret = ret * a  
    return ret
```

A Better Way

- If b is even:
 - $a^b = a^{b/2} * a^{b/2}$
- If b is odd:
 - $a^b = a * a^{b/2} * a^{b/2}$
- Using integer division
- Of course, $a^0 = 1$ (base case)

Runtime

- What is the runtime of those two algorithms?
- The iterative solution had a for loop from 0 to $b-1$ ($O(b)$)
- What about the recursive solution?
- Each call did some constant amount of work, then made a recursive call with $b/2$
- Stopped when $b = 0$
- How many calls could have been made?

Runtime

- At most $\log_2 b$ (by convention, $\lg = \log_2$)
- So our runtime is bounded above by $c \lg b$ (for some constant c)
- $O(\lg b)$
- Other algorithm was exponentially worse!

Application

- Exponentiation not just useful for real values
- Also works for matrices, where each multiplication is much more expensive
- Naïve multiplication of two square matrices of size n is $O(n^3)$
- Using this method, raising a square matrix to the power of b has runtime $O(n^3 \lg b)$ instead of $O(n^3 b)$, a significant savings

Fibonacci Revisited

- Consider the following:

$$\begin{bmatrix} F_{k+2} \\ F_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

- Which implies that:

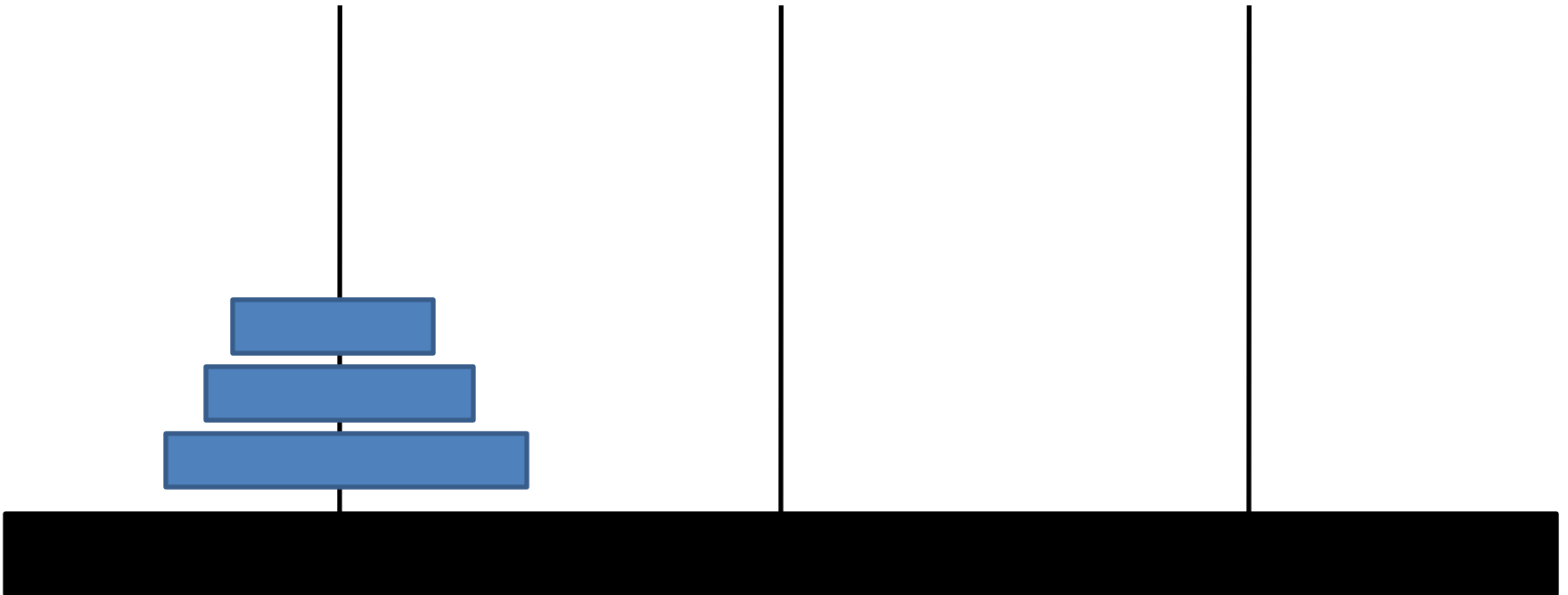
$$\begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Fibonacci Revisited

- Using this idea, we can calculate F_n in time $O(\lg n)$ instead of $O(n)$ (exponential savings!)

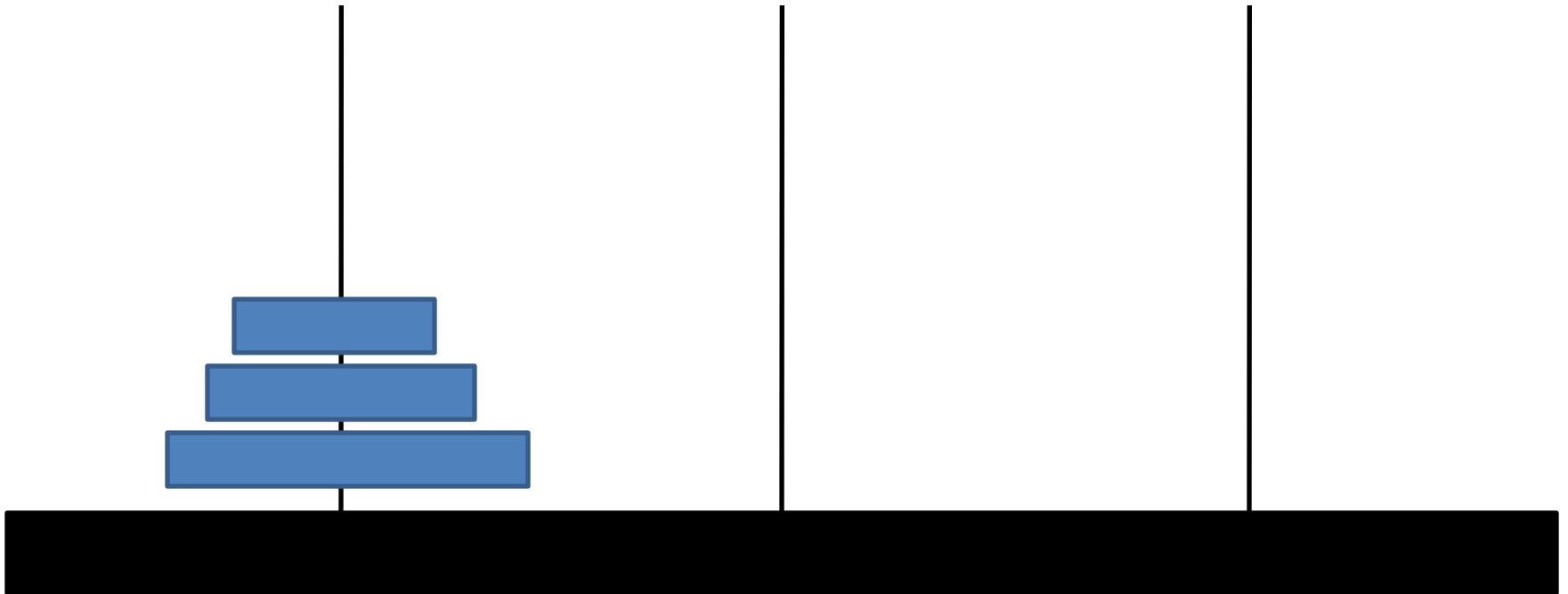
Towers of Hanoi

- Classic puzzle/game



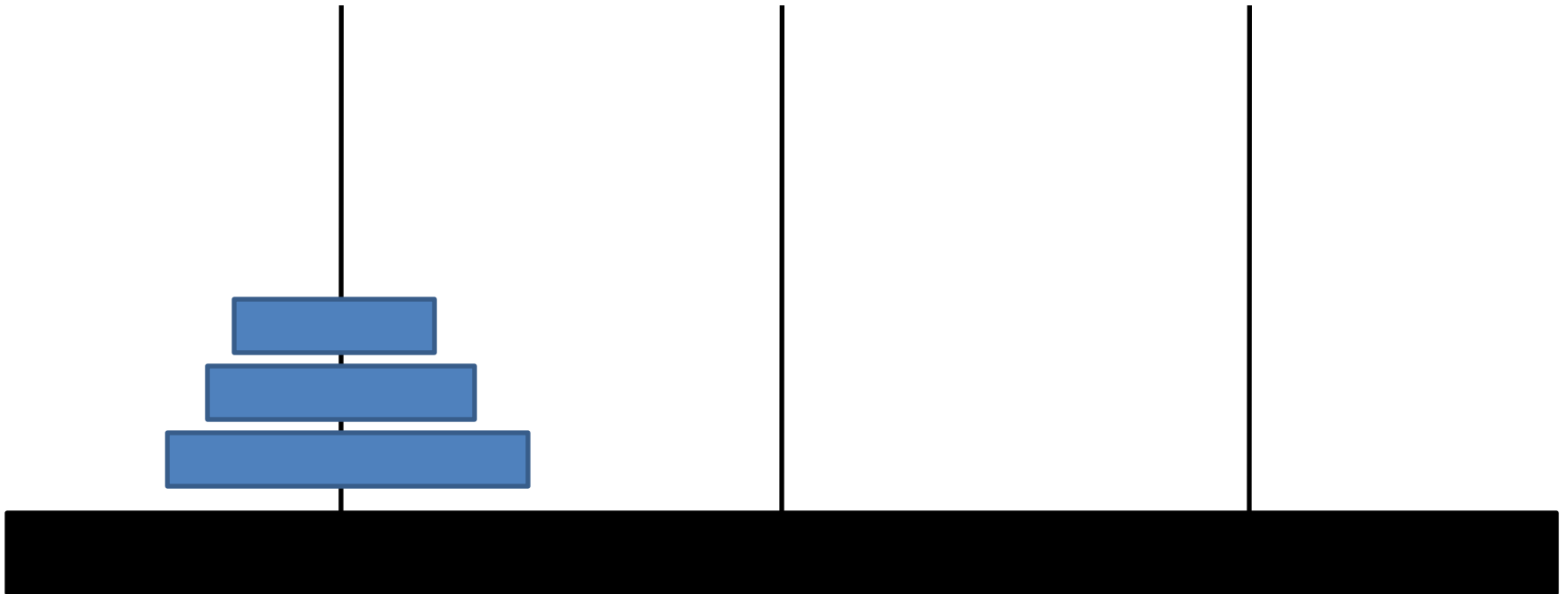
Towers of Hanoi

- 3 pegs, and n disks (here, $n = 3$)



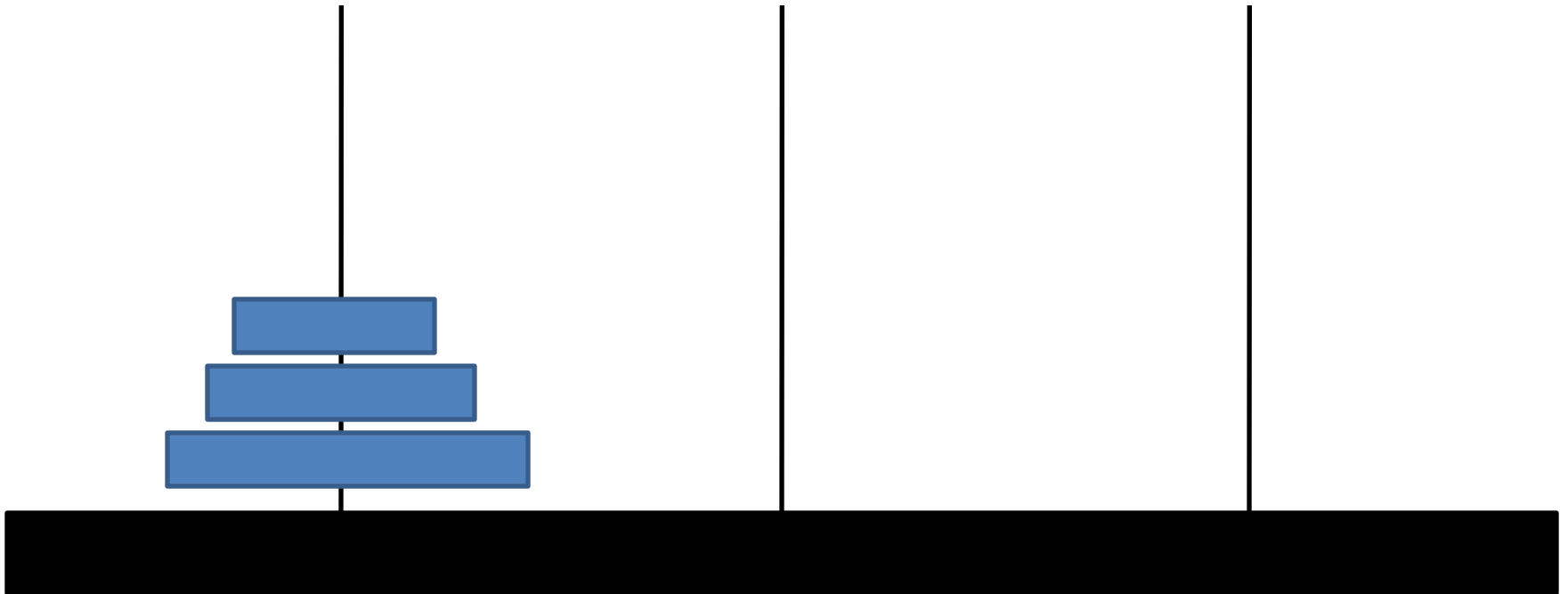
Towers of Hanoi

- Each disk is a different size



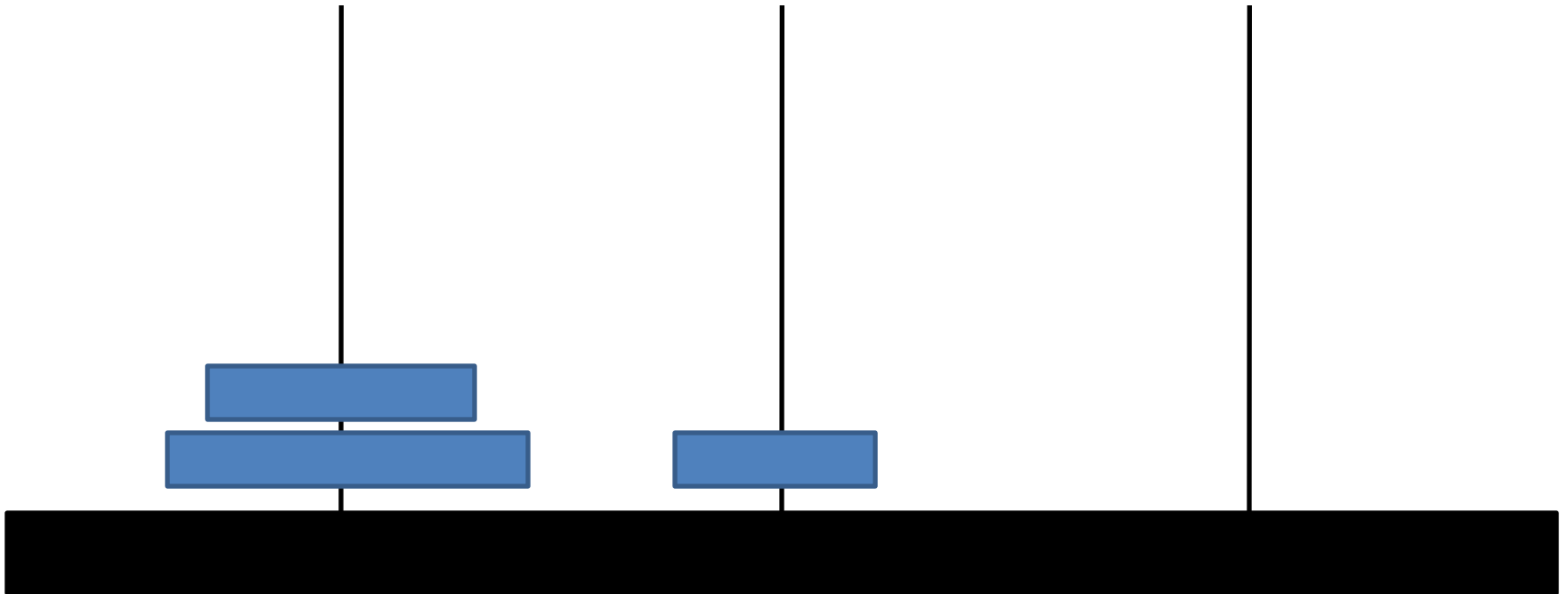
Towers of Hanoi

- Can move disks from the top of one stack to the top of another (pegs are stacks)



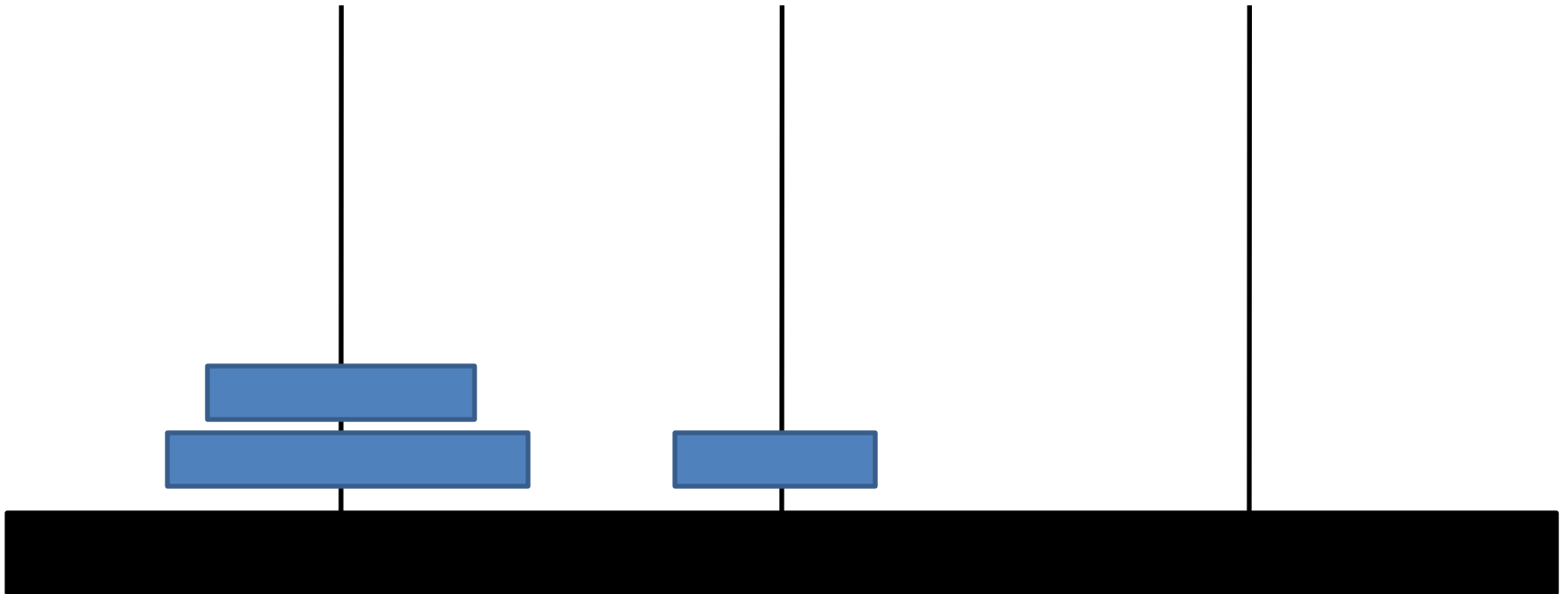
Towers of Hanoi

- Can move disks from the top of one stack to the top of another (pegs are stacks)



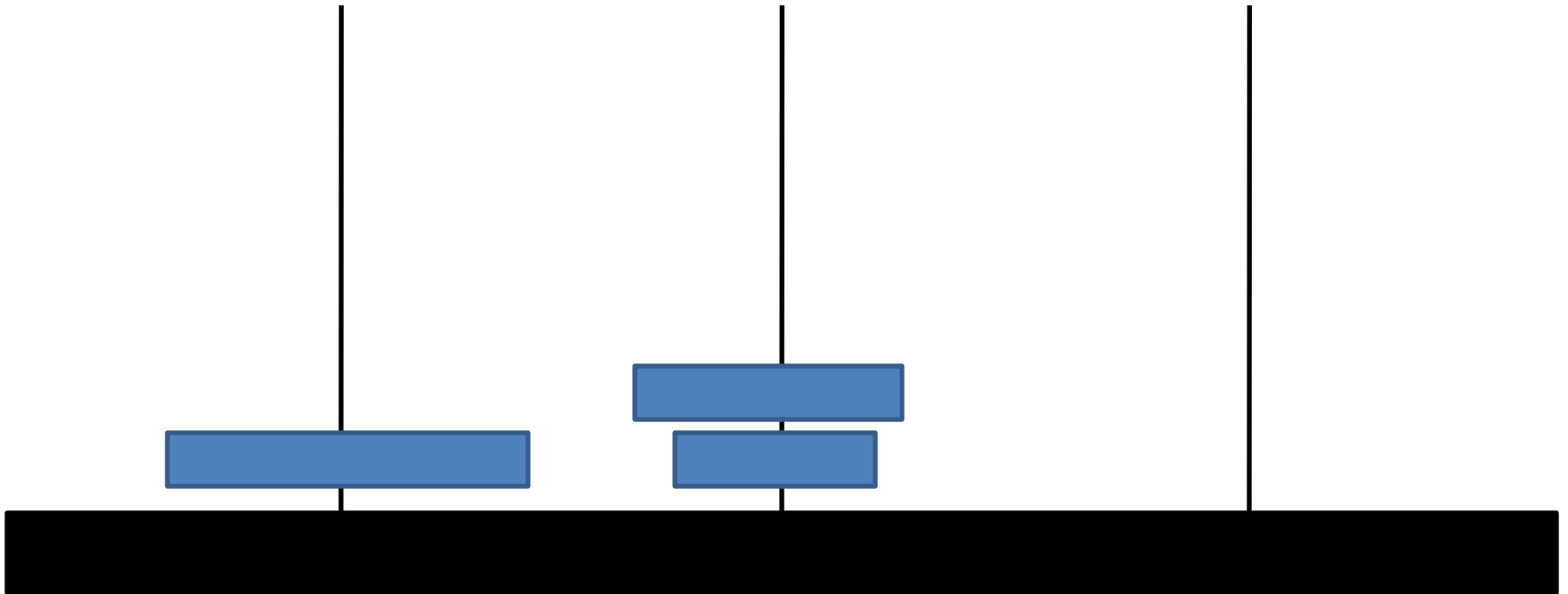
Towers of Hanoi

- Cannot place a disk on top of one that is smaller



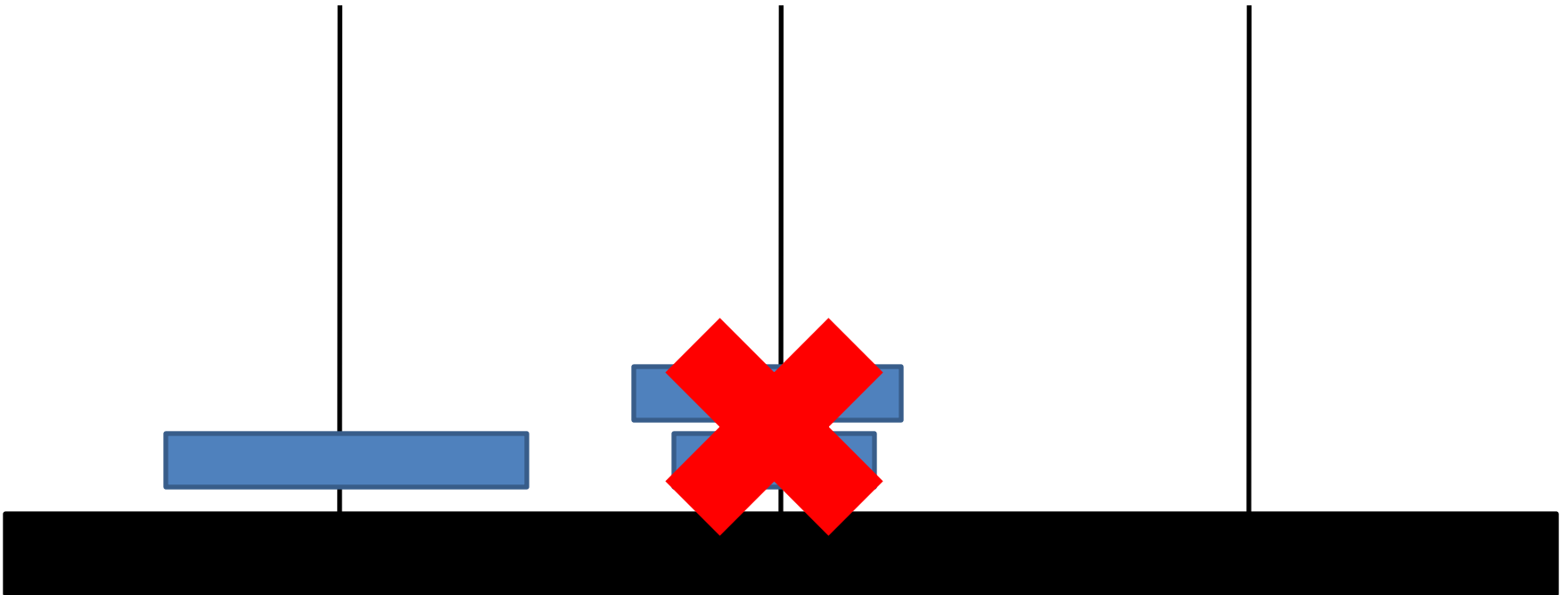
Towers of Hanoi

- Cannot place a disk on top of one that is smaller



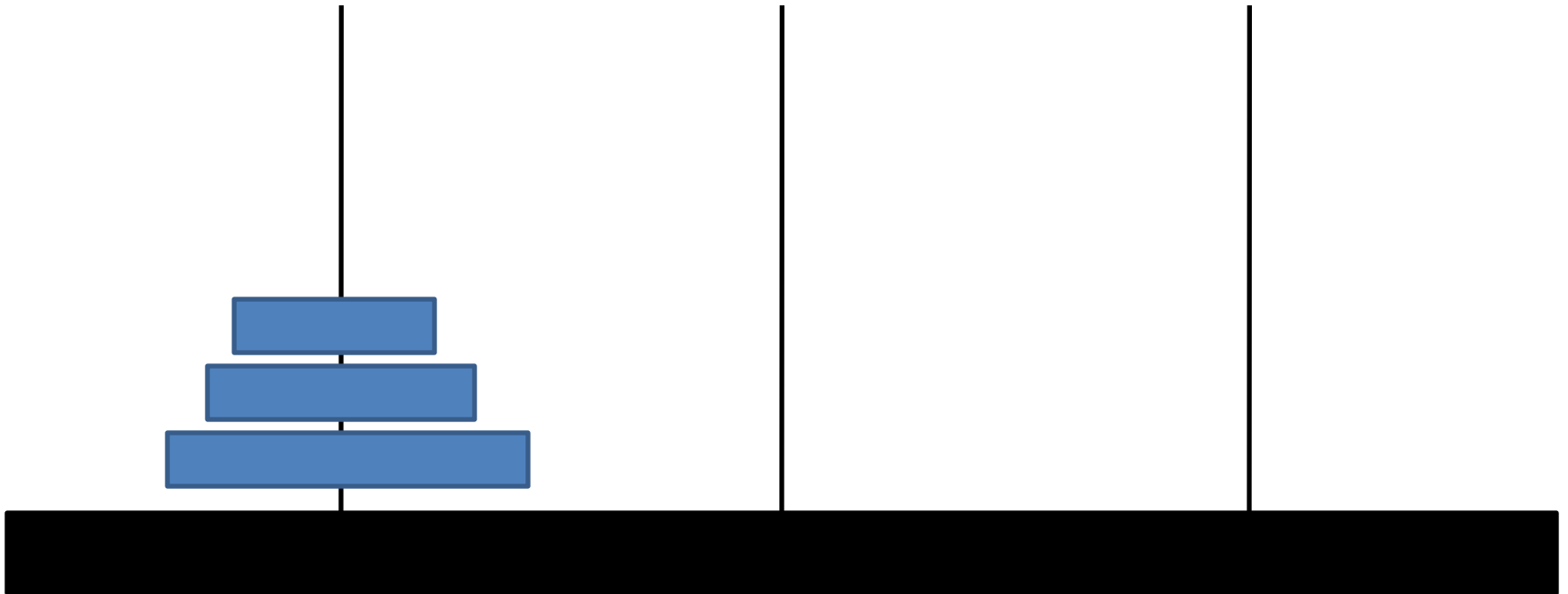
Towers of Hanoi

- Cannot place a disk on top of one that is smaller



Towers of Hanoi

- Goal is to move all the disks from peg 1 to peg 3 following these rules

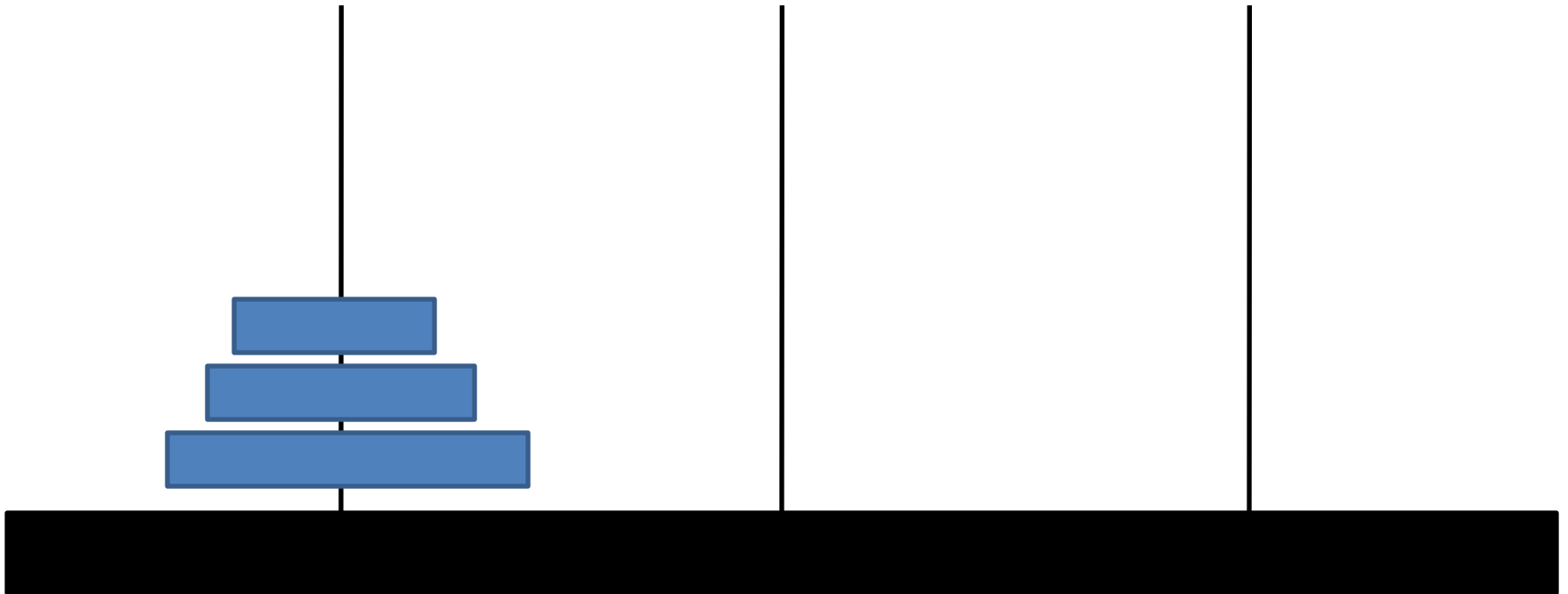


Observations

- The biggest disk (disk n) cannot be placed on top of any others
- So, in order to move that disk from peg 1 to peg 3, every other disk must be on peg 2
- So we need to move $n-1$ disks from peg 1 to peg 2, then disk n from peg 1 to peg 3, then $n-1$ disks from peg 2 to peg 3
- Recursion!

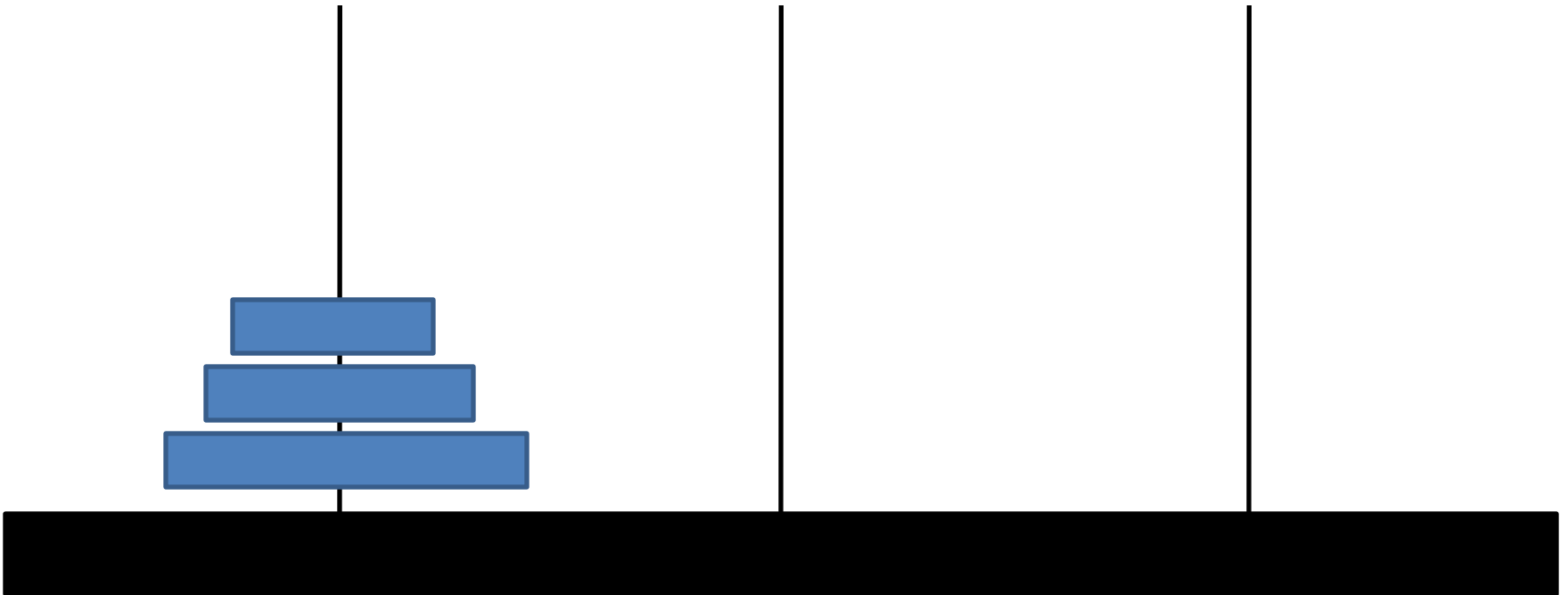
Towers of Hanoi

- Move top 2 disks to peg 2



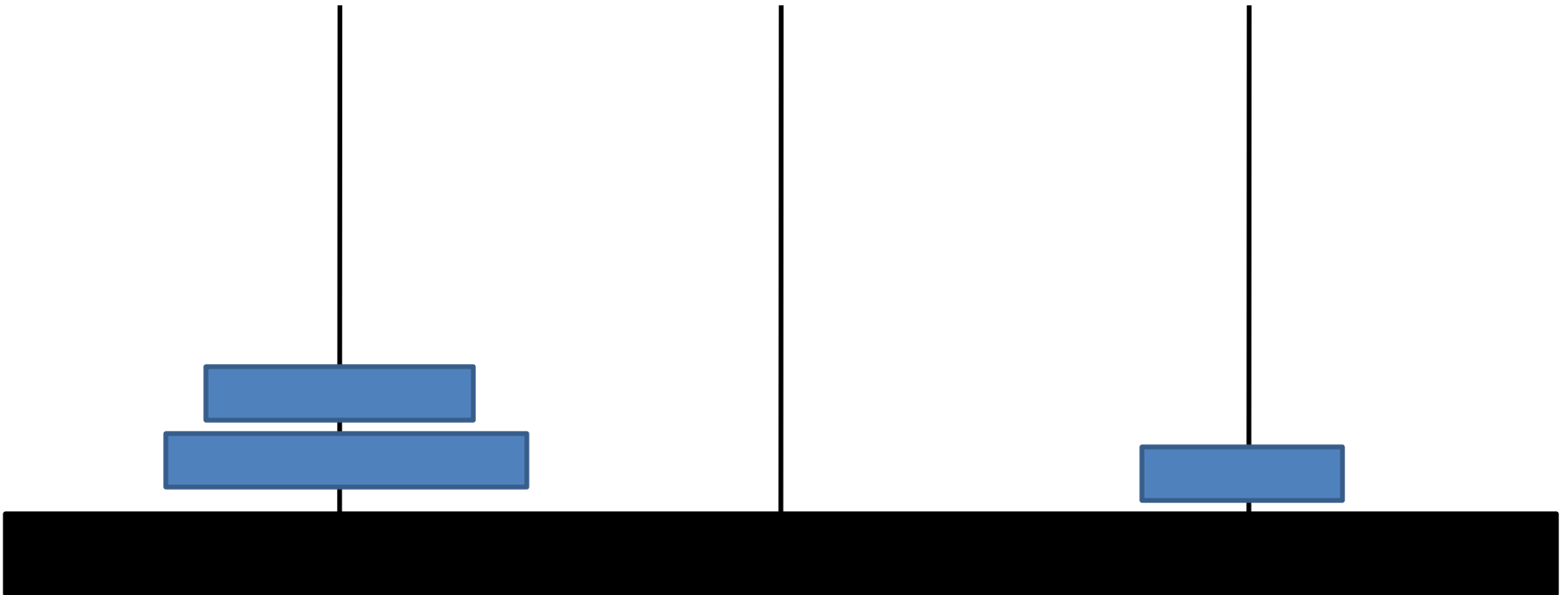
Towers of Hanoi

- Move top 2 disks to peg 2
- Move top disk to peg 3



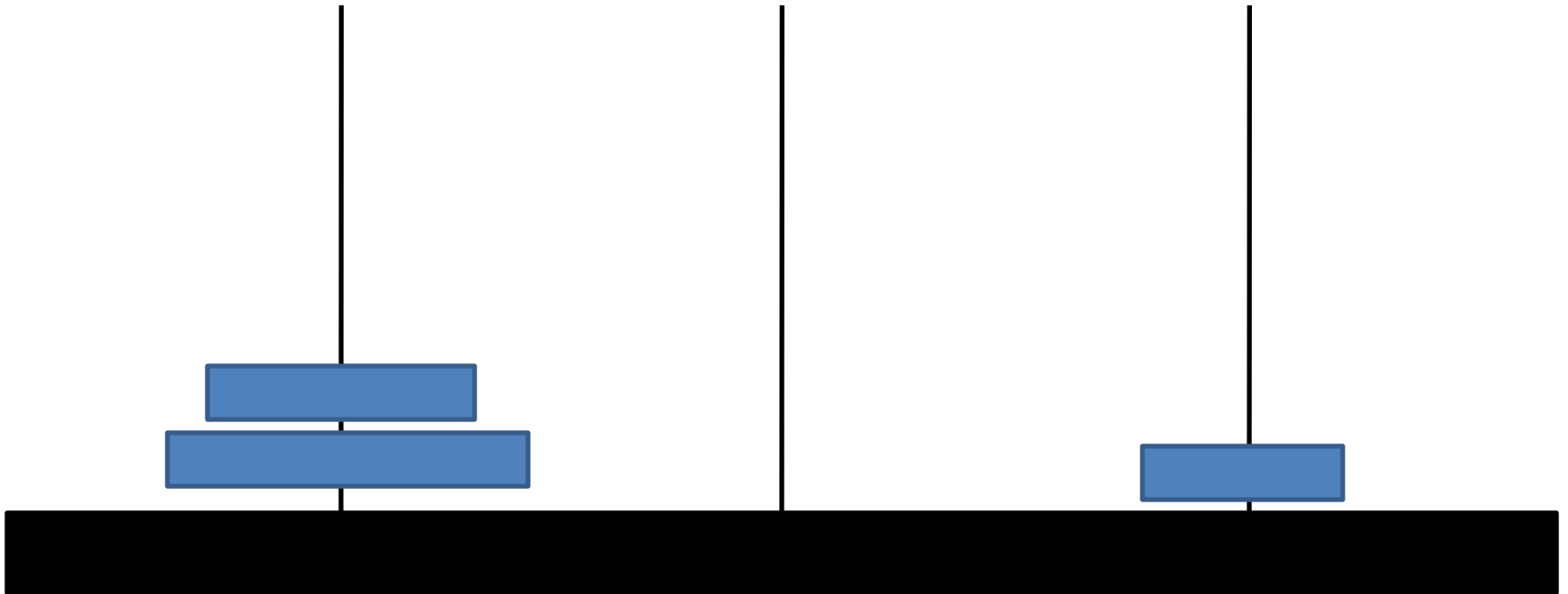
Towers of Hanoi

- Move top 2 disks to peg 2
- Move top disk to peg 3



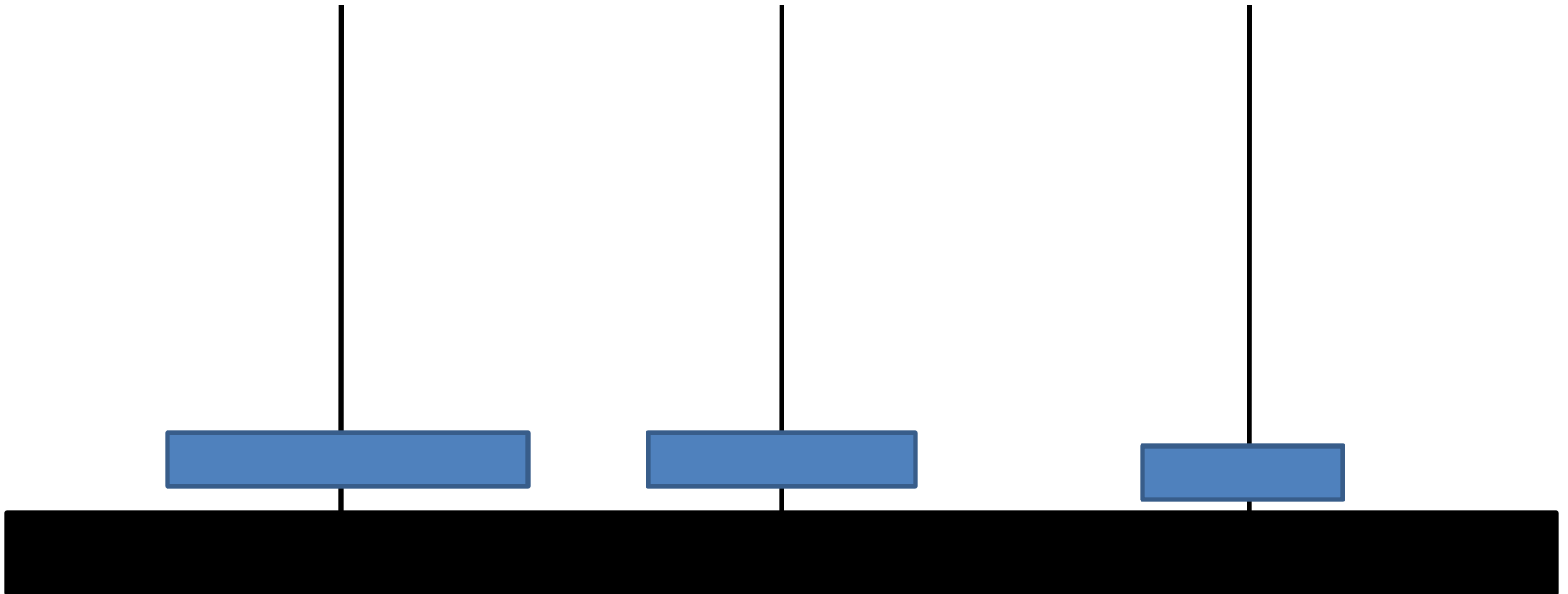
Towers of Hanoi

- Move disk 2 to peg 2



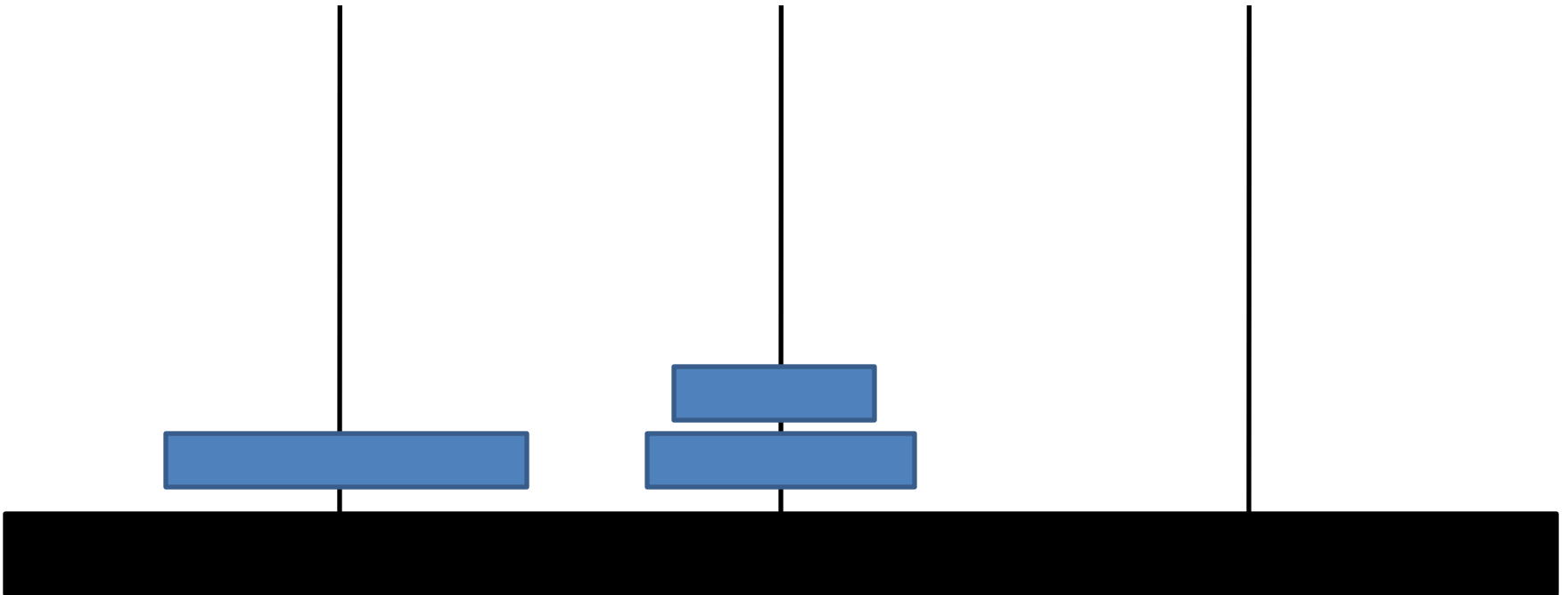
Towers of Hanoi

- Move disk 2 to peg 2
- Move disk 1 to peg 2



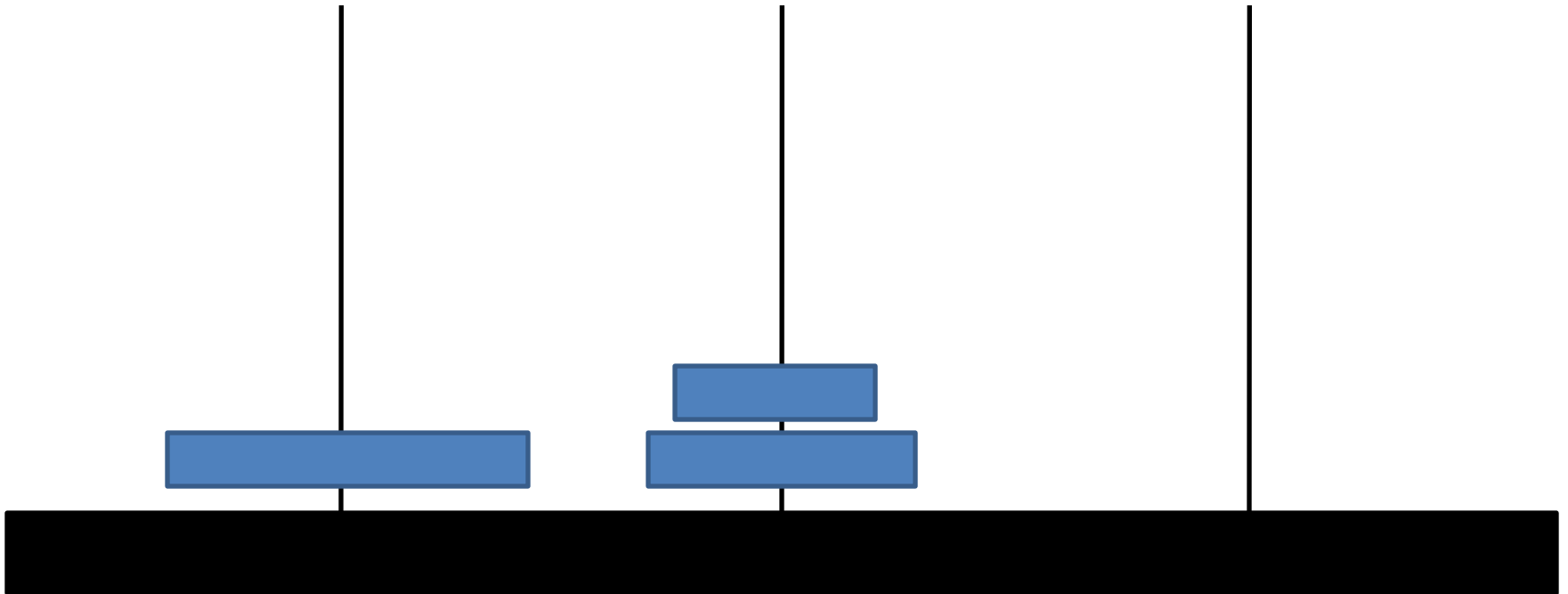
Towers of Hanoi

- Move disk 2 to peg 2
- Move disk 1 to peg 2



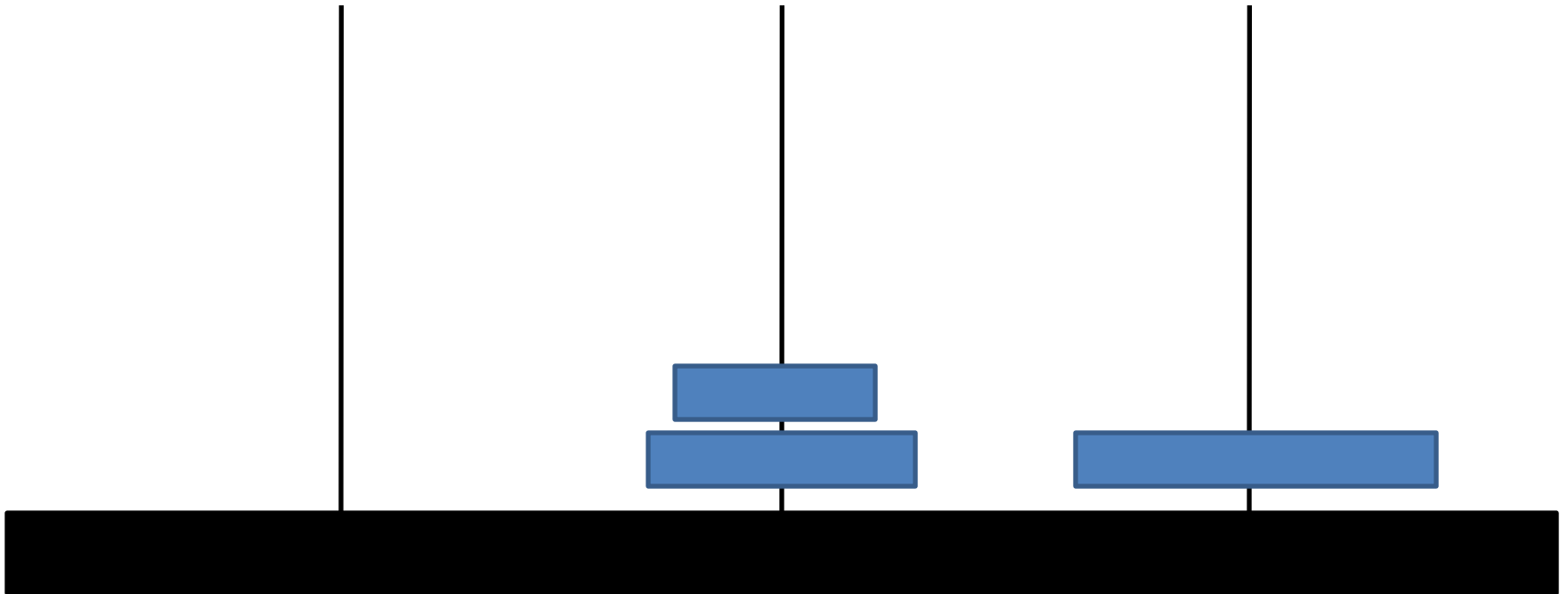
Towers of Hanoi

- Have now moved the top 2 disks to peg 2
- Now we move disk 3 to peg 3



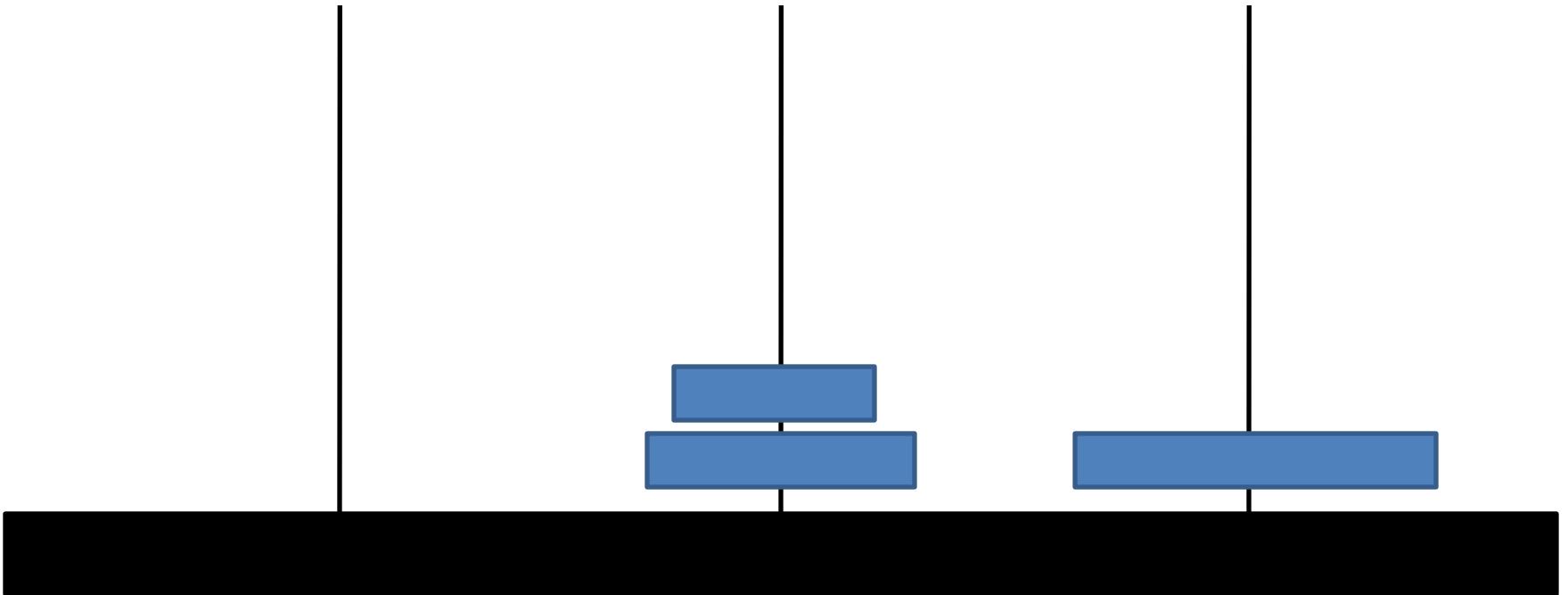
Towers of Hanoi

- Have now moved the top 2 disks to peg 2
- Now we move disk 3 to peg 3



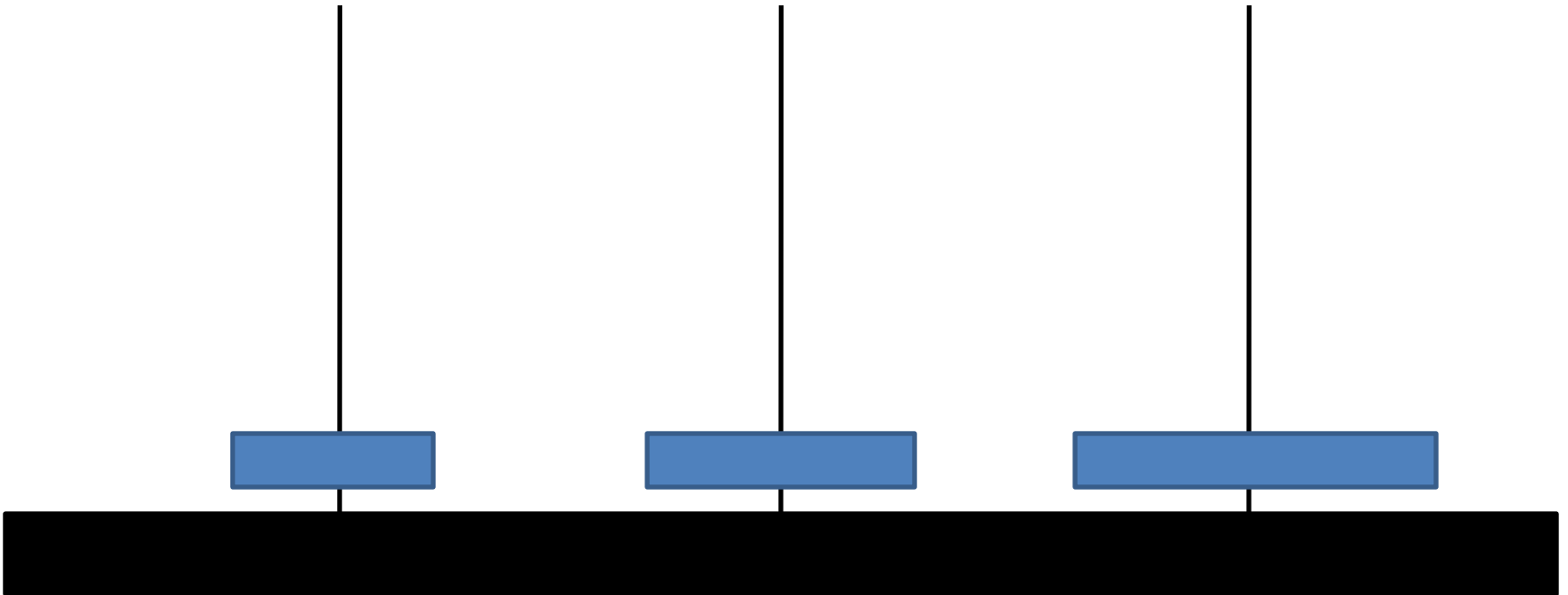
Towers of Hanoi

- Finally, move 2 disks from peg 2 to peg 3



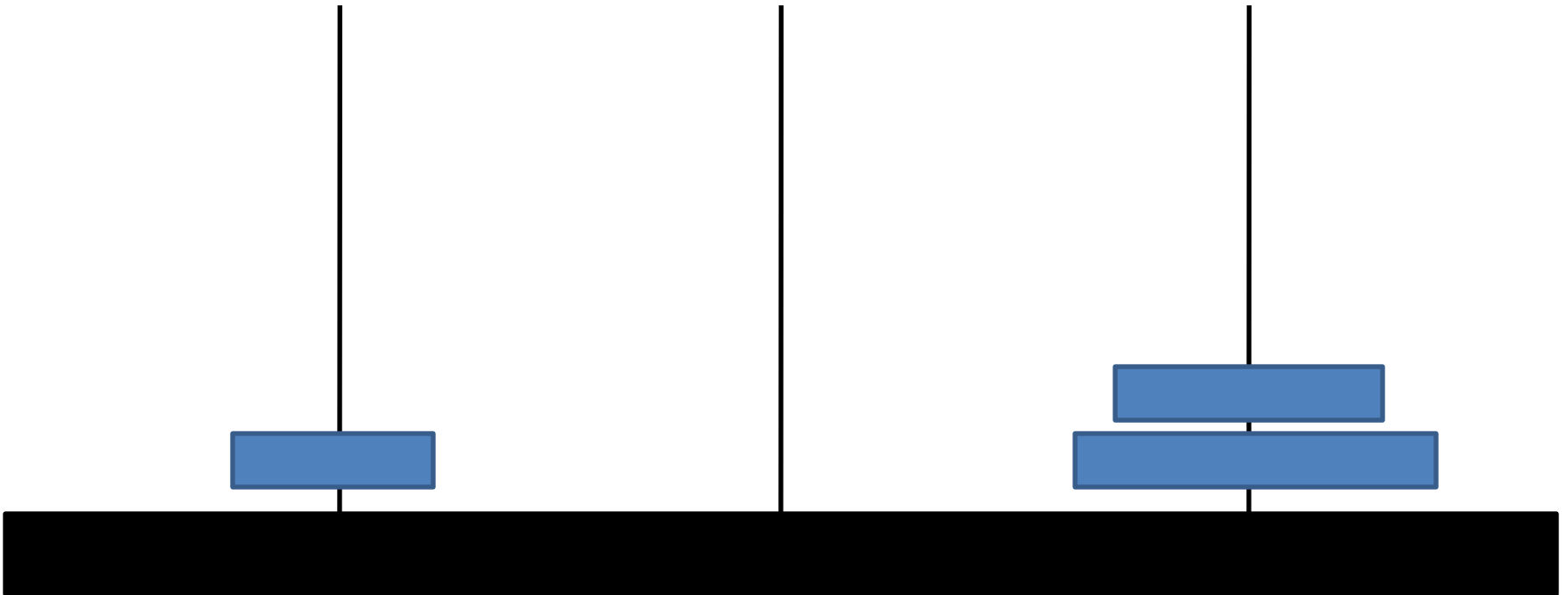
Towers of Hanoi

- Finally, move 2 disks from peg 2 to peg 3



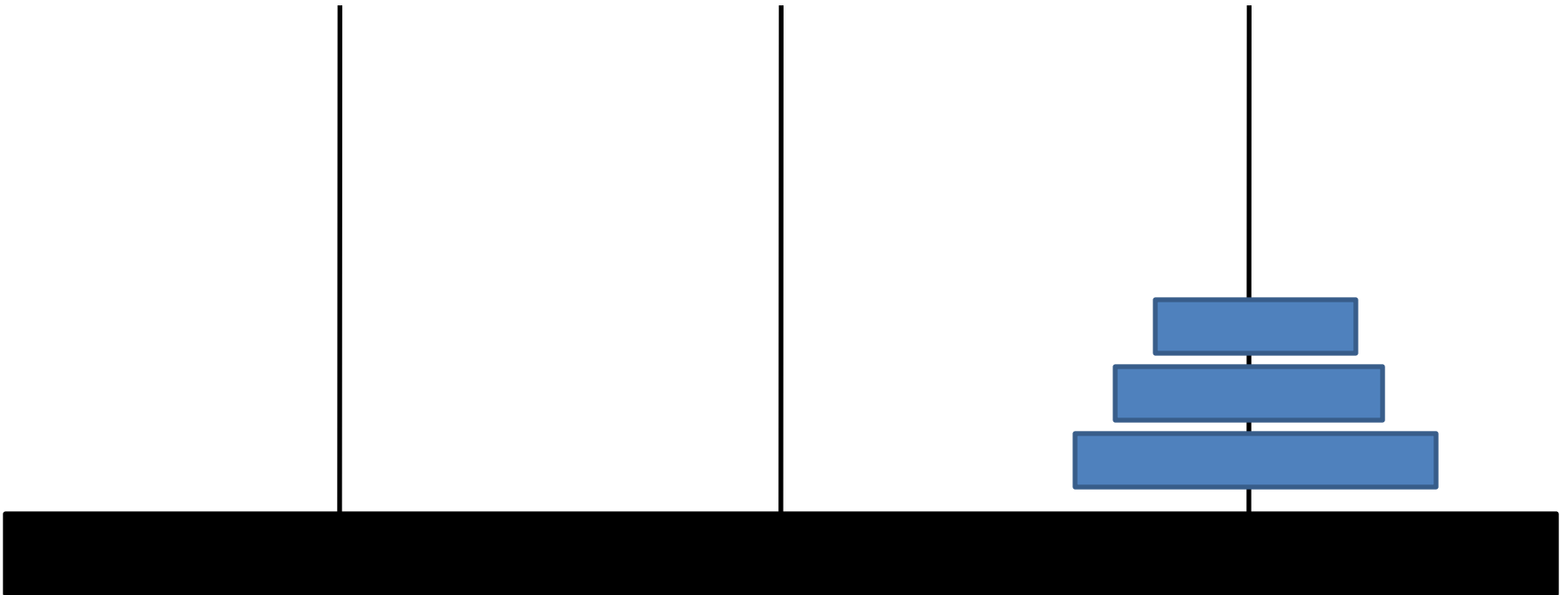
Towers of Hanoi

- Finally, move 2 disks from peg 2 to peg 3



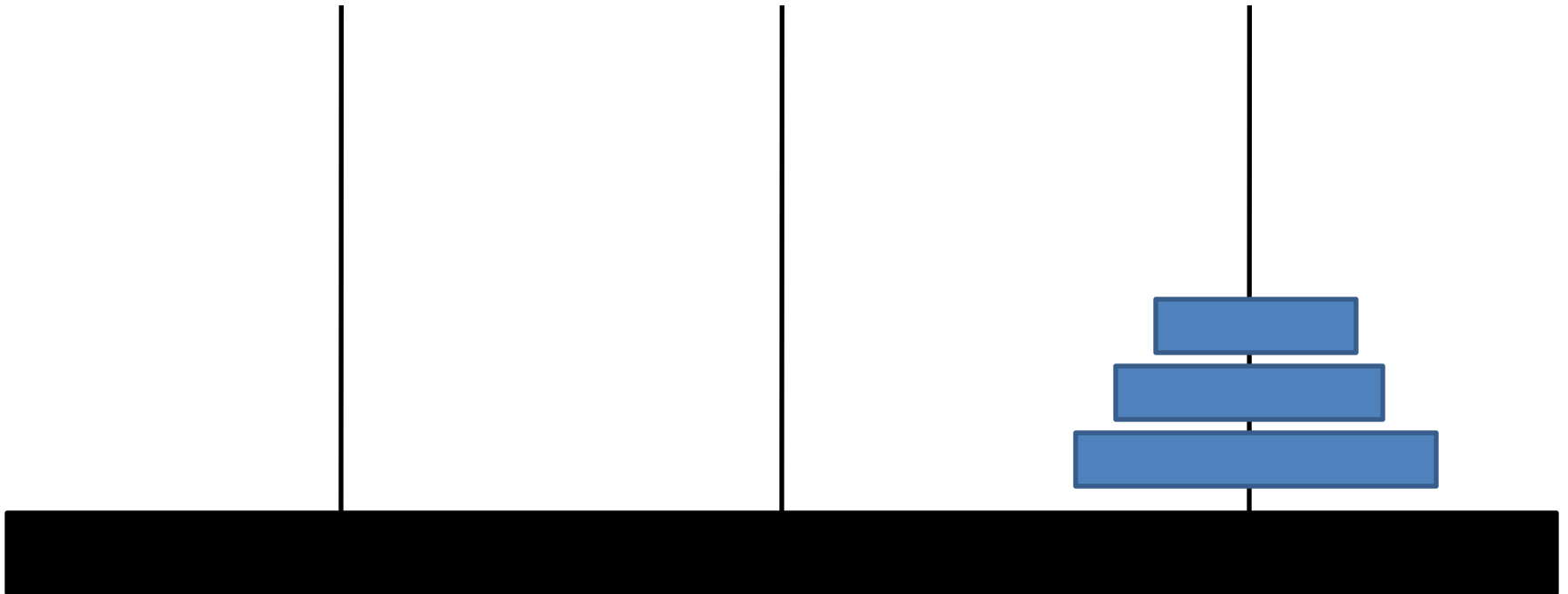
Towers of Hanoi

- Finally, move 2 disks from peg 2 to peg 3



Towers of Hanoi

- Success!



How Many Steps?

- How many moves does it take to move a stack of n disks between two pegs?
- Our n -disk recursive algorithm moves $n-1$ disks between two pegs, then moves 1 disk, then $n-1$ between two pegs
- $T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1)+1$
- $T(1) = 1$ (base case)
- $1, 3, 7, 15, 31, \dots, 2^n - 1$

Searching

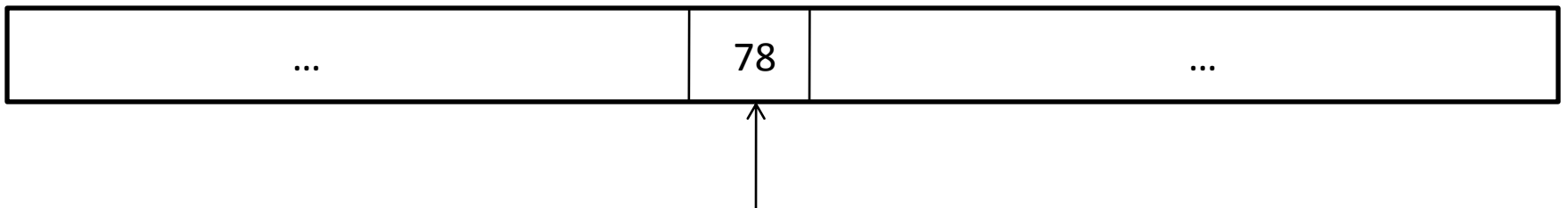
- Suppose we have some list with n elements in it
- Want to check if some value is in the list
- Easy enough, iterate over each element i and check to see if $i == \text{value}$
- Runtime: $O(?)$

Searching a Sorted List

- In general, can't do better than $O(n)$
- What if we have more information about how things are stored in the list?
- Suppose our list is sorted
 - Real world example: phone books, dictionaries, etc. are all sorted lists

Searching a Sorted List

- Suppose we have a list $a[0..(n-1)]$
- The list is sorted
 - $a[0] \leq a[1] \leq \dots \leq a[n-2] \leq a[n-1]$
- We are searching for the number 53 in the list
- What can we conclude if $a[n/2] = 78$?



Searching a Sorted List

- We can throw away the right half of the list
- Only need to consider $a[0:(n/2)]$
- Algorithm (Binary Search):
 - look at the middle element
 - If less than target, throw away left half
 - If greater than target, throw away right half

