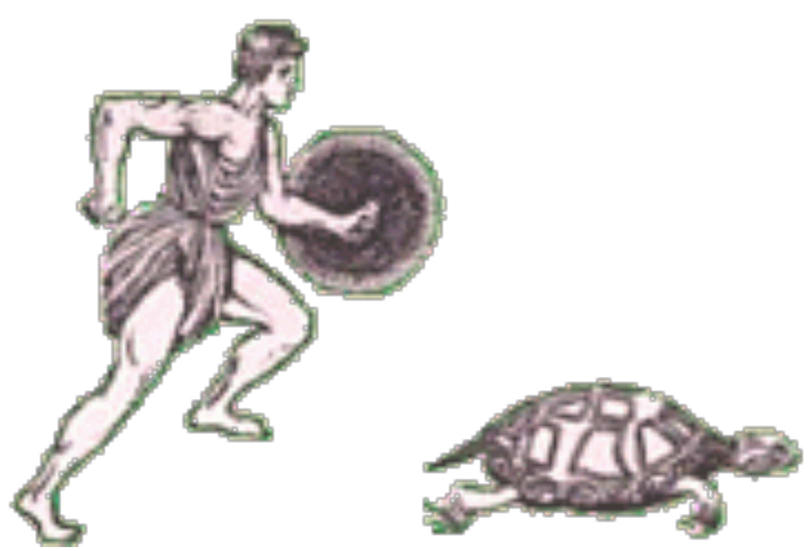
Data Structures in Java

Lecture 4: Introduction to Algorithm Analysis and Recursion

9/21/2015



Daniel Bauer

Algorithms

- An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.
- Algorithm Analysis — Questions:
    - Does the algorithm terminate?
    - Does the algorithm solve the problem? (correctness)
    - What resources does the algorithm use?
        - Time / Space

Analyzing Runtime: Basics

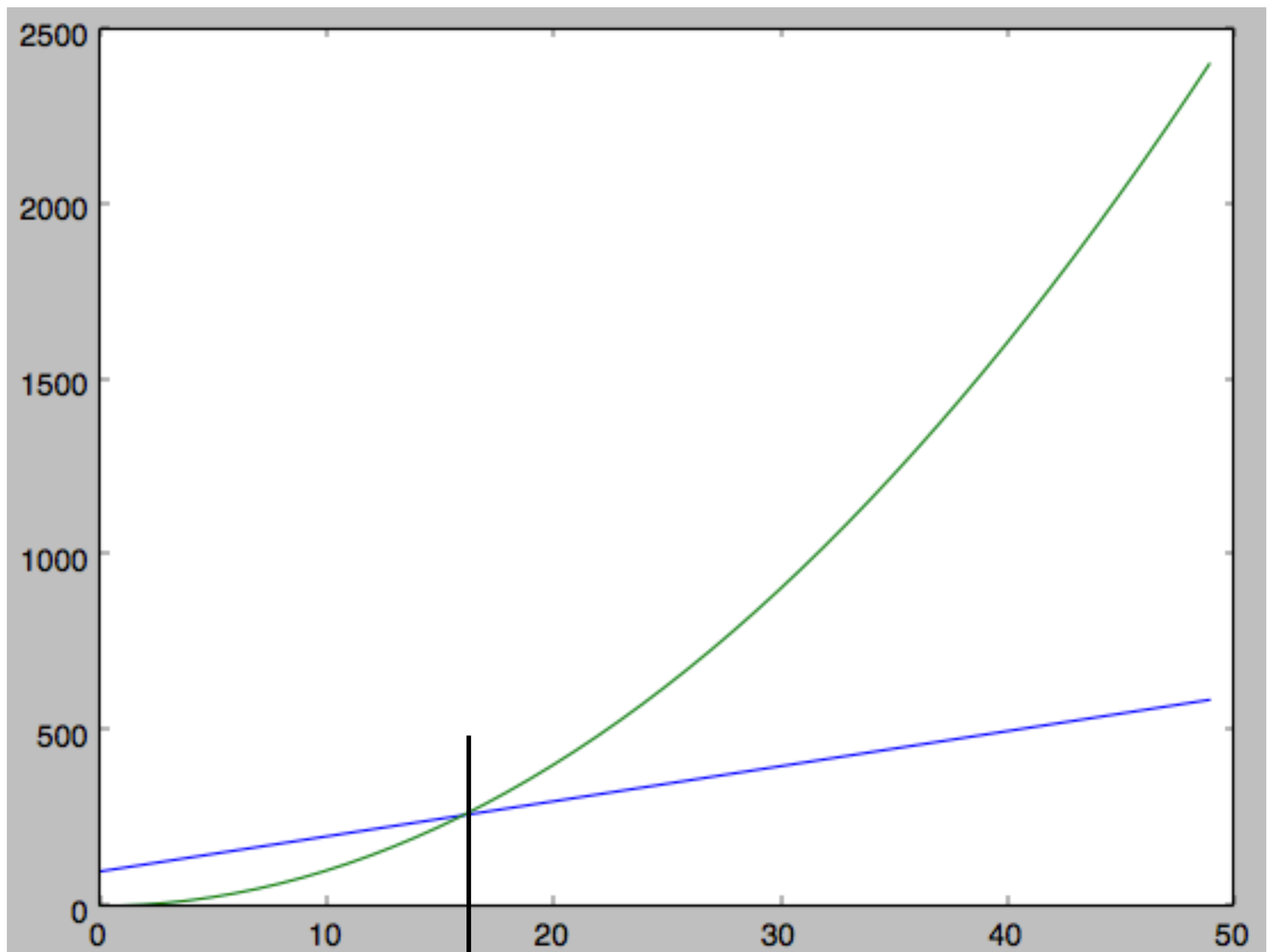
- We usually want to compare several algorithms.
  - Compare between different algorithms how the runtime *T(N)* grows with increasing input sizes *N.*
- We are using Java, but the same algorithms could be implemented in any language on any machine.
- How many basic operations/“steps” does the algorithm take? All operations assumed to have the same time.

Worst and Average case

- Usually the runtime depends on the type of input (e.g. sorting is easy if the input is already sorted).
- $T_{worst}(N)$: *worst case* runtime for the algorithm on ANY input. The algorithm is **at least** this fast.
- $T_{average}(N)$: *Average case analysis* — expected runtime on typical input.
- $T_{best}(N)$: Occasionally we are interested in the best case analysis.

Comparing Function Growth: Big-Oh Notation

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.



$f(N) = N^2 + 2$

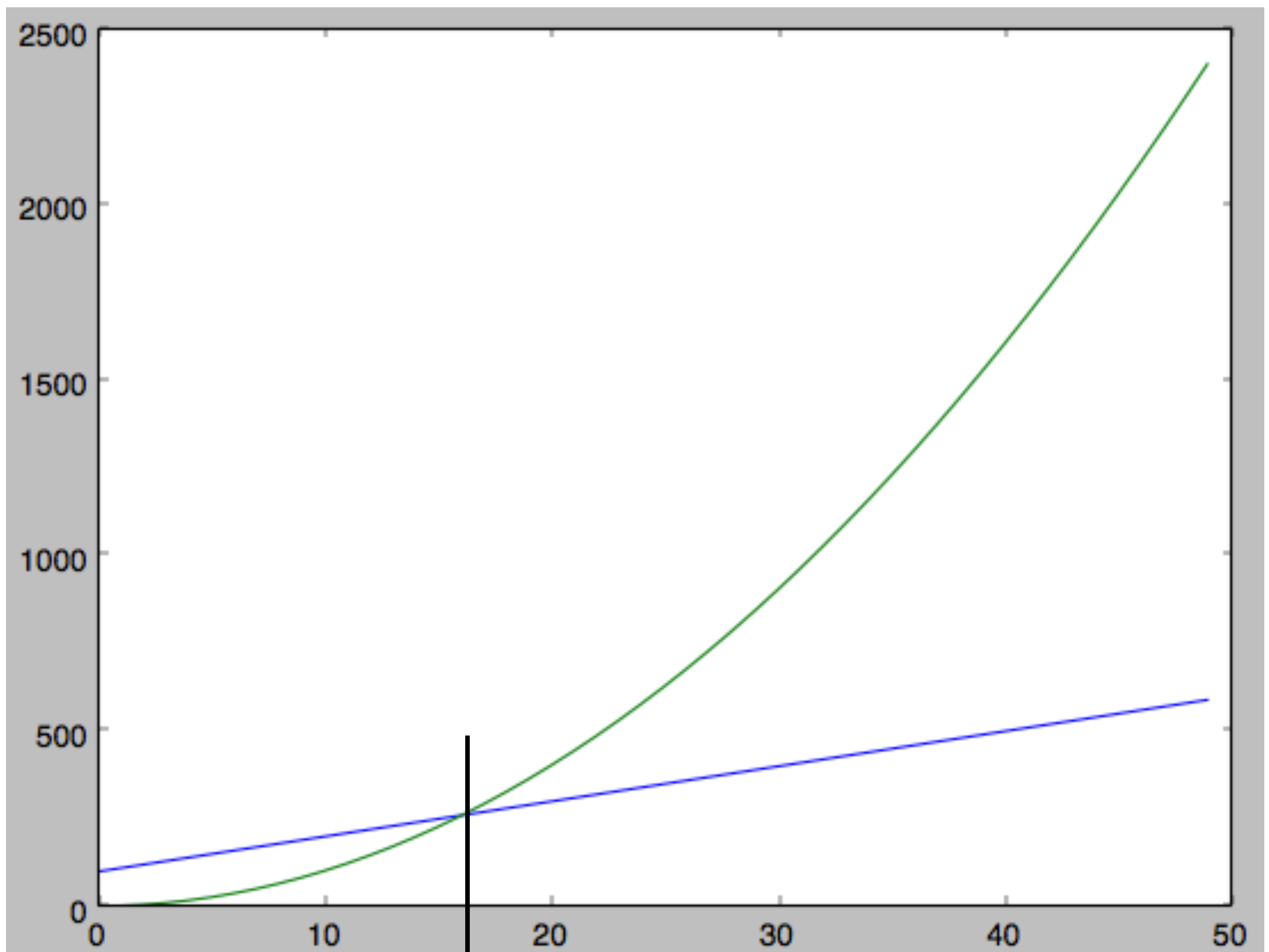
$T(N) = 10N + 100$

e.g. $c = 1$, $n_0 = 16.1$

Comparing Function Growth: Big-Oh Notation

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le cf(N)$ when $N \ge n_0$.

"T(N) is in the order of f(N)"



$f(N) = N^2 + 2$

$T(N) = 10N + 100$

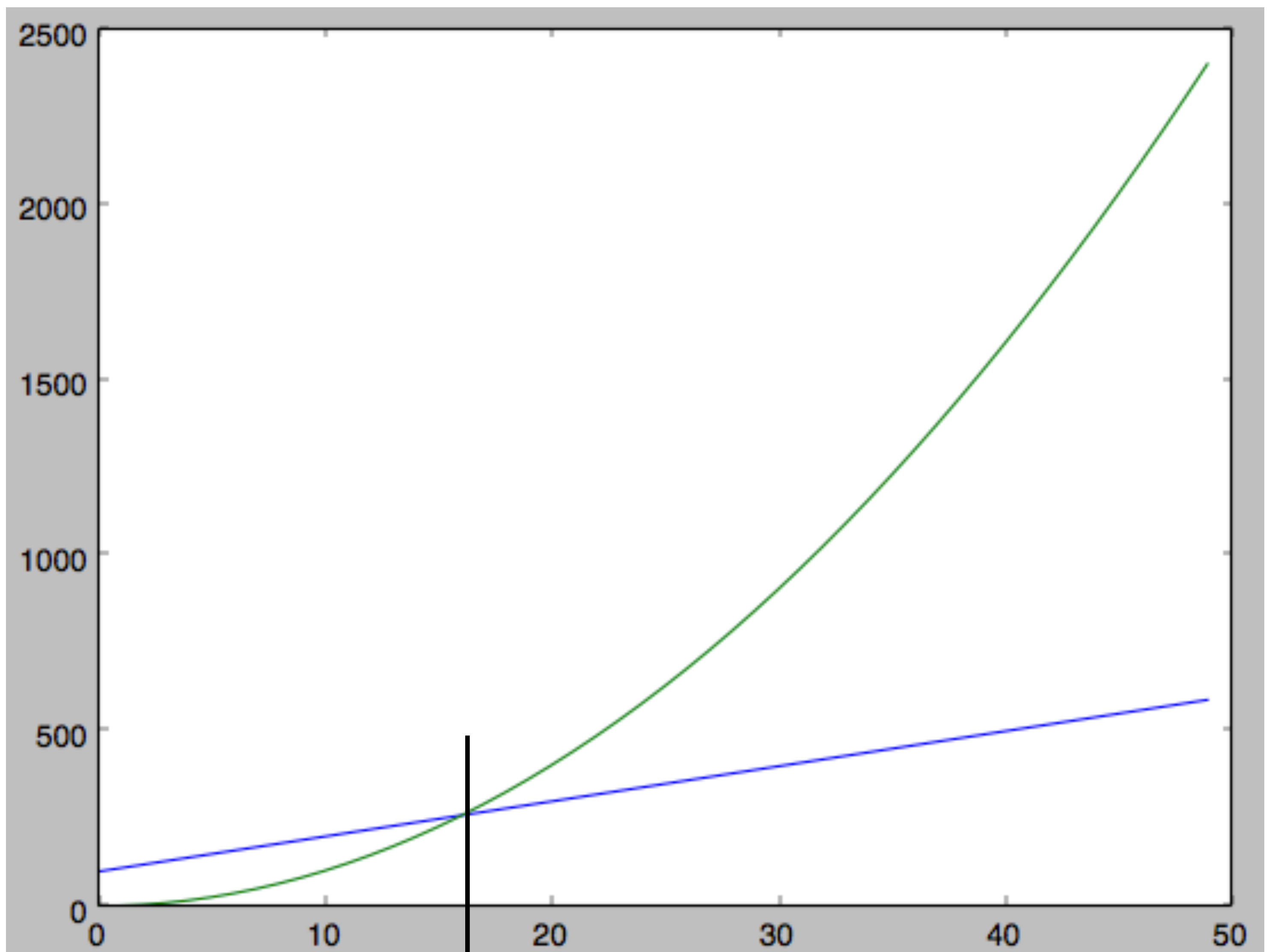
e.g. $c = 1$, $n_0 = 16.1$

Comparing Function Growth: Big-Oh Notation

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le cf(N)$ when $N \ge n_0$.

"T(N) is in the order of f(N)"

"f(N) is an upper bound on T(N)"



$f(N) = N^2 + 2$

$T(N) = 10N + 100$

e.g. $c = 1$, $n_0 = 16.1$

Comparing Function Growth: Additional Notations

- Lower Bound:

$T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \geq cf(N)$ when $N \geq n_0$.

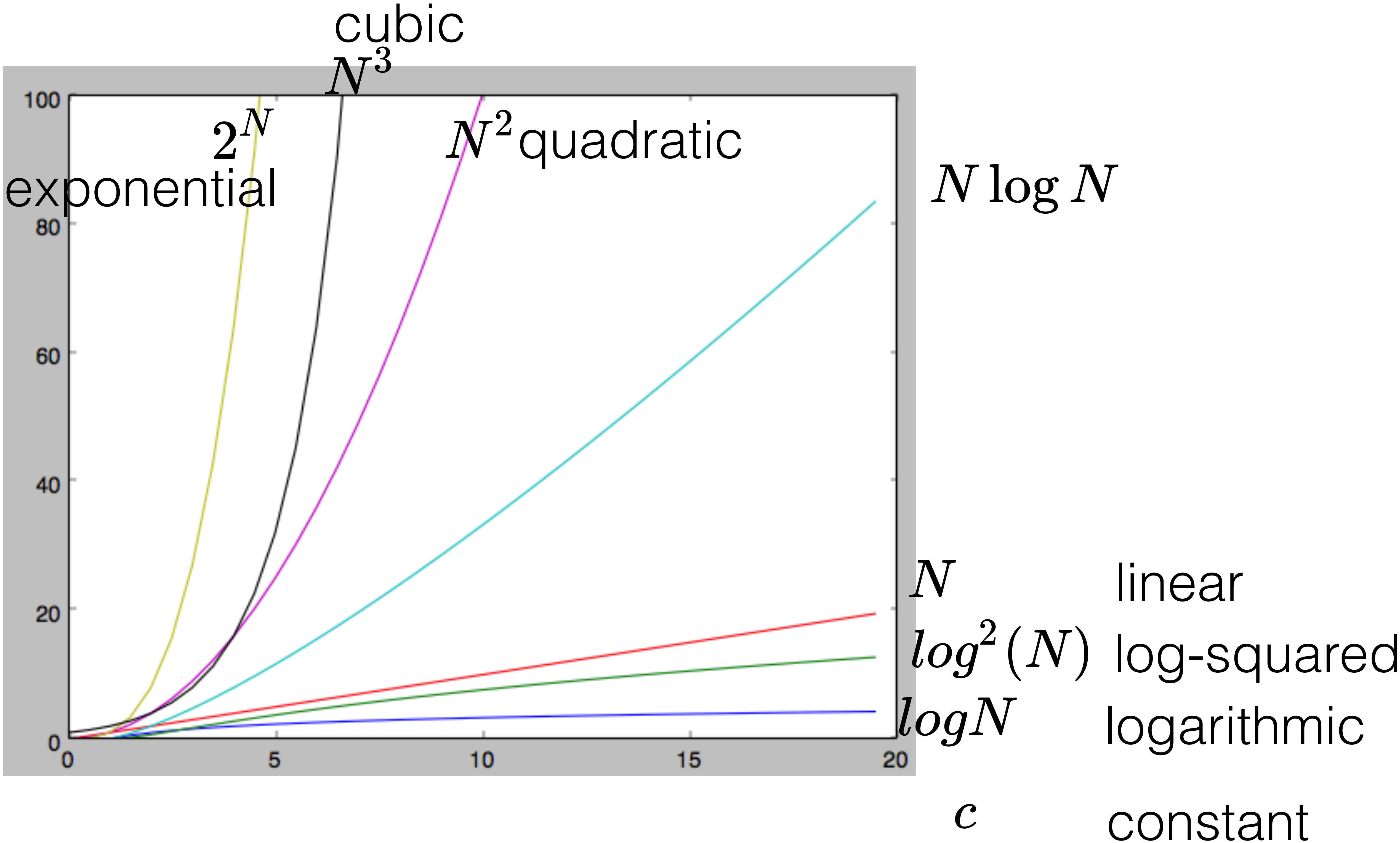
- Tight Bound: T(N) and f(N) grow at the same rate

$T(N) = \Theta(f(N))$ if $T(N) = \Omega(f(N))$ and $T(N) = O(f(N))$.

- Strict Upper Bound:

$T(N) = o(f(N))$ if for all positive constants c there is some n_0 such that $T(N) < cf(N)$ when $N > n_0$.

Typical Growth Rates



Rules for Big-Oh (1)

If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ then

1. $T_1(N) + T_2(N) = O(f(N) + g(N))$

   $O(\max(f(N), g(N))$

2. $T_1(N) * T_2(N) = O(f(N) * g(N))$

Rules for Big-Oh (2)

If $T(N)$ is a polynomial of degree k then

$$T(N) = \Theta(N^k)$$

For instance: $9N^3 + 12N^2 - 5 = \Theta(N^3)$

$log^k(N) = O(N)$ for any k .

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;

 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0; 1 step

 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum; 1 step
}
```

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;
 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

1 step

N iterations

4 steps each

1 step

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;
 
 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

1 step

N iterations

2 steps each

4 steps each

1 step

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;
  
 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

1 step

1 step (initialization)

+1 step for last test

N iterations

2 steps each

4 steps each

1 step

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;
  for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

1 step

1 step (initialization)

+1 step for last test

N iterations

2 steps each

4 steps each

1 step

T(N) = 6 N + 4 = O(N)

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```
public static int sum(int n){
  int partialSum = 0;
  
 for (int i = 1; i <= n; i++)
      partialSum += i * i * i;
  return partialSum;
}
```

1 step

1 step (initialization)

+1 step for last test

N iterations

2 steps each

4 steps each

1 step

T(N) = 6 N + 4 = O(N)

(running time of statements in the loop) X (iterations)

General Rules: Basic *for*-loops

Compute $\sum_{i=1}^{N} i^3$

```java
public static int sum(int n){
  int partialSum = 0;
 for (int i = 1; i <= n; i++)
     partialSum += i * i * i;
  return partialSum;
}
```

1 step

1 step (initialization)

+1 step for last test

N iterations

2 steps each

4 steps each

1 step

T(N) = 6 N + 4 = O(N)

(running time of statements in the loop) X (iterations)

If loop runs a constant number of times: *O(c)*

General Rules: Nested *L*oops

Analyze inside-out.

```
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
    k++;
```

General Rules: Nested *L*oops

Analyze inside-out.

```
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
    k++;
```

1 step each $O(c)$

General Rules: Nested *L*oops

Analyze inside-out.

```
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
    k++;
```

N iterations $O(N)$

1 step each $O(c)$

General Rules: Nested *L*oops

Analyze inside-out.

```
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
    k++;
```

N iterations $O(N) * O(N) = O(N^2)$

N iterations $O(N)$

1 step each $O(c)$

General Rules: Consecutive Statements

```
for (i = 0; i < n; i++)
  a[i] = 0;
for (i=0; i < n; i++)
  for (j = 0; j < n; j++)
      a[i] += a[j] + i + j;
```

General Rules: Consecutive Statements

```
for (i = 0; i < n; i++)
  a[i] = 0;
for (i=0; i < n; i++)
  for (j = 0; j < n; j++)
      a[i] += a[j] + i + j;
```

$O(N)$

$O(N^2)$

General Rules: Consecutive Statements

```
for (i = 0; i < n; i++)
  a[i] = 0;
for (i=0; i < n; i++)
  for (j = 0; j < n; j++)
      a[i] += a[j] + i + j;
```

$O(N)$

$O(N^2)$

$$O(N) + O(N^2) = O(N^2)$$

Basic Rules: *if/else* conditionals

```
if (condition)
    S1
else
    S2
```

$T_{\text{test}}(N)$

$T_{S_1}(N)$

$T_{S_2}(N)$

Basic Rules:
if/else conditionals

$T_{\text{test}}(N)$

```
if (condition)
    S1
else
    S2
```

$T_{S_1}(N)$

$T_{S_2}(N)$

$$T(N) = O(\max(T_{S_1}(N), T_{S_2}(N)) + T_{\text{test}}(N))$$

Logarithms in the Runtime

```
public static int binarySearch(int[] a, int x) {
  int low = 0;
  int  high = a.length - 1;

  while ( low <= high) {
    int mid = (low + high) / 2;
     if (a[mid] < x)
       low = mid + 1;
     else if(a[mid] > x)
       high = mid - 1;
     else
       return mid; // found
  }
  return -1; // Not found.
}
```

How many iterations of the *while* loop?

Every iteration cuts remaining partition in half.

Recursion

- A recursive algorithm uses a function (or method) that calls itself.
- Need to make sure there is some *base case* (otherwise causing an infinite loop).
- The recursive call needs to *make progress* towards the base case.
  - Reduces the problem to a simpler subproblem.

Recursive Binary Search

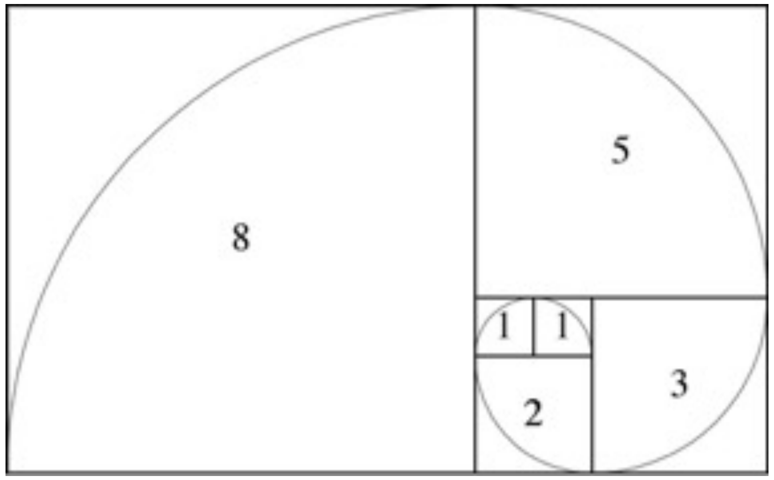
Fibonacci Sequence

Fibonacci Sequence

- 1, 1, 2, 3, 5, 8, 13, 21, …

Fibonacci Sequence

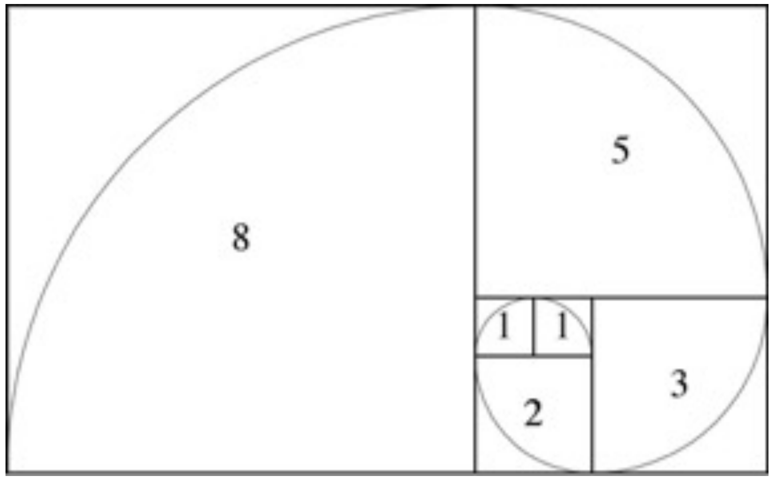
- 1, 1, 2, 3, 5, 8, 13, 21, …



$$F_1 = 1$$
$$F_2 = 1$$
$$F_{k+1} = F_k + F_{k-1}$$

Recursive Definition

Fibonacci Sequence



- 1, 1, 2, 3, 5, 8, 13, 21, …

$$F_1 = 1$$
$$F_2 = 1$$
$$F_{k+1} = F_k + F_{k-1}$$

Recursive Definition

- Closed form solution is complicated.

- Instead easier to compute this algorithmically.

Fibonacci Sequence in Java

```
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

Fibonacci in Java

```java
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

Fibonacci in Java

```
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

Base case

Fibonacci in Java

```
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

Base case

Recursive call - making progress

How many steps does the algorithm need?

```java
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

How many steps does the algorithm need?

```
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

Base case: 1 step $T(1) = O(c)$, $T(2) = O(c)$

How many steps does the algorithm need?

```
public class Fibonacci {

  public static void main(String[] args) {
    Fibonacci fib = new Fibonacci();
    int k =  Integer.parseInt(args[0]);
    System.out.println(fib.fibonacci(k));
  }

  public int fibonacci(int k) throws IllegalArgumentException{
    if (k < 1) {
      throw new IllegalArgumentException("Expecting a positive integer.");
    }
    if (k == 1 | k == 2) {
      return 1;
    } else {
      return fibonacci(k-1) + fibonacci(k-2);
      }
    }
}
```

Base case: 1 step $T(1) = O(c)$, $T(2) = O(c)$

Recursive calls: $T(k) = O(T(k-1) + T(k-2))$

Analyzing the Recursive Fibonacci Solution

Recursive calls: $T(k) = O(T(k-1) + T(k-2))$

Base case: $T(1) = O(c)$, $T(2) = O(c)$

Analyzing the Recursive Fibonacci Solution

Recursive calls: $T(k) = O(T(k-1) + T(k-2))$
Base case: $T(1) = O(c)$, $T(2) = O(c)$

Recurrence Relation.
How do we solve this?

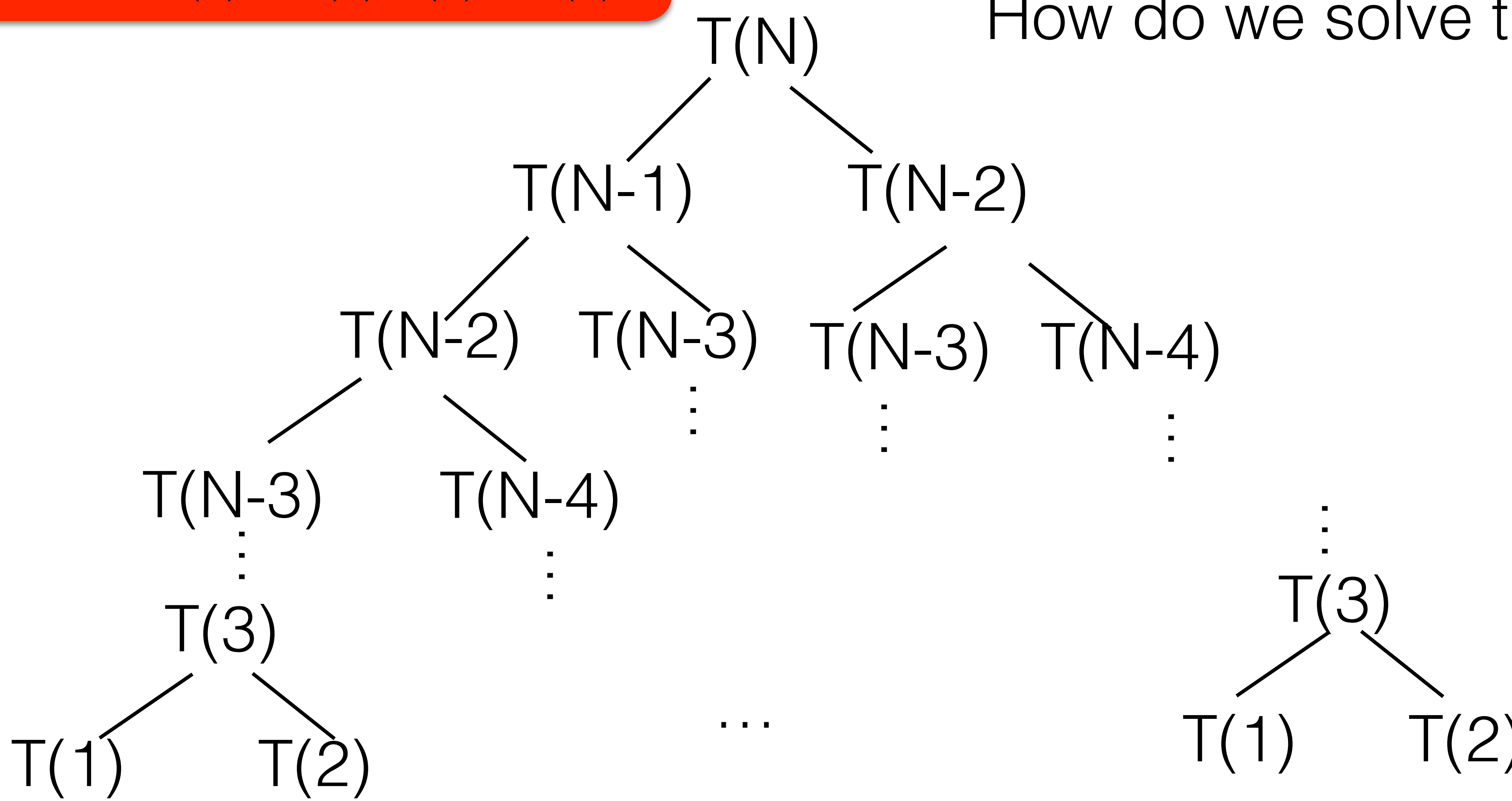
Analyzing the Recursive Fibonacci Solution

Recursive calls: $T(k) = O(T(k-1) + T(k-2))$

Base case: $T(1) = O(c)$, $T(2) = O(c)$

Recurrence Relation.

How do we solve this?



Fibonacci Sequence v.2

```java
public int fibonacci(int k) throws IllegalArgumentException{

  if (k < 1) {
    throw new IllegalArgumentException("Expecting a positive integer.");
  }
  int b = 1; //k-2
  int a = 1; //k-1
  for (int i=3; i<=k; i++) {
    int new_fib = a + b;
    b = a;
    a = new_fib;
  }
  return a;
}
```

Dynamic programming: Cache intermediate solutions so they can be re-used.

Fibonacci Sequence v.2

```
public int fibonacci(int k) throws IllegalArgumentException{

  if (k < 1) {
    throw new IllegalArgumentException("Expecting a positive integer.");
  }
  int b = 1; //k-2
  int a = 1; //k-1
  for (int i=3; i<=k; i++) {
    int new_fib = a + b;
    b = a;
    a = new_fib;
  }
  return a;
}
```

$T(N) = O(N)$

Dynamic programming: Cache intermediate solutions so they can be re-used.

Rules for Recursion

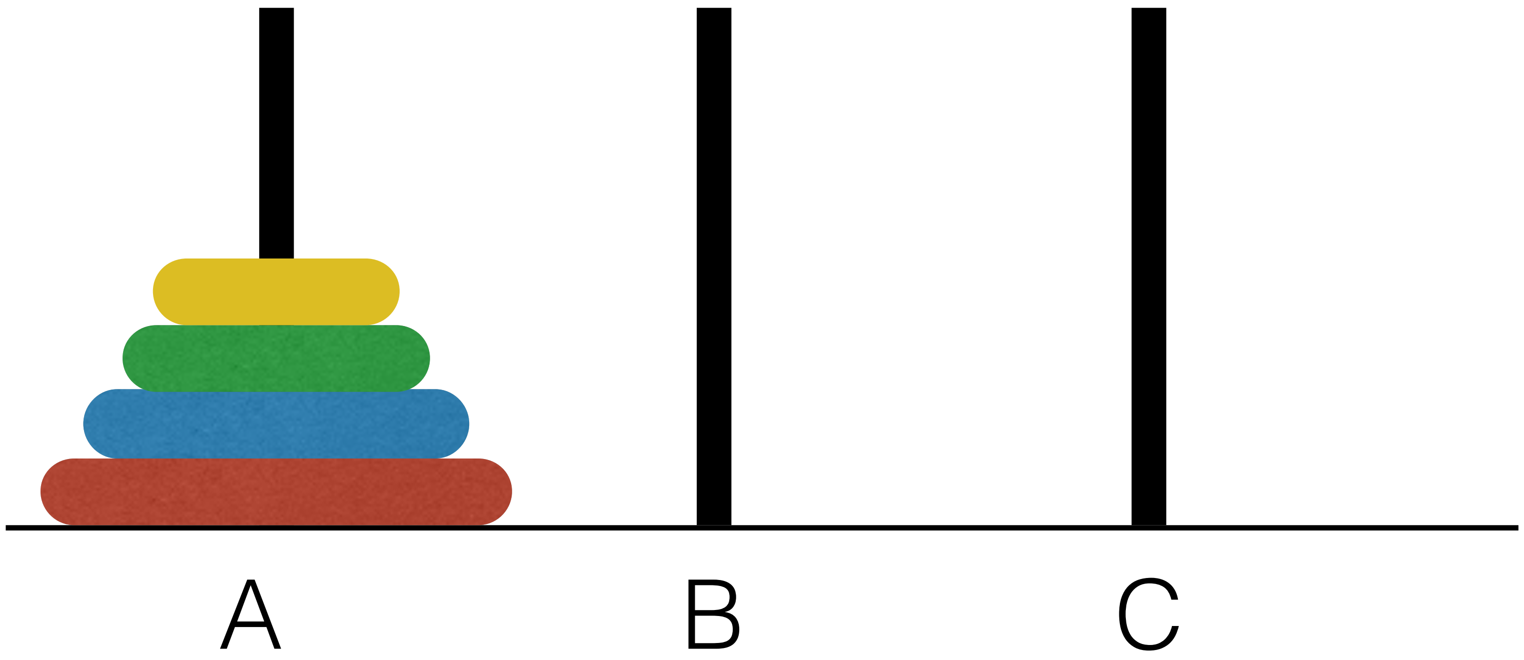
1. B*ase Case*

2. *Making Progress*

3. *Design Rule* - Assume all recursive calls work.

4. *Compound Interest Rules* - Never duplicate work by solving the same instance of a problem in separate recursive calls.

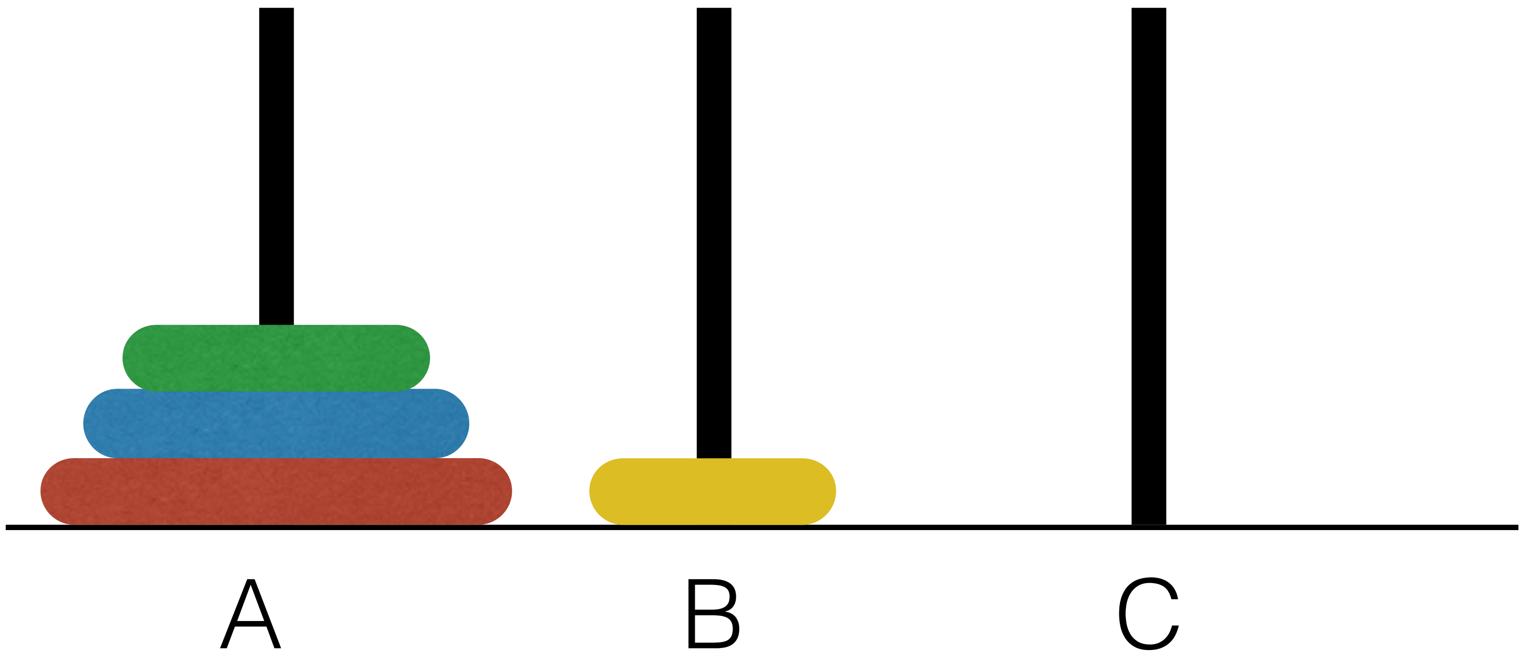
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

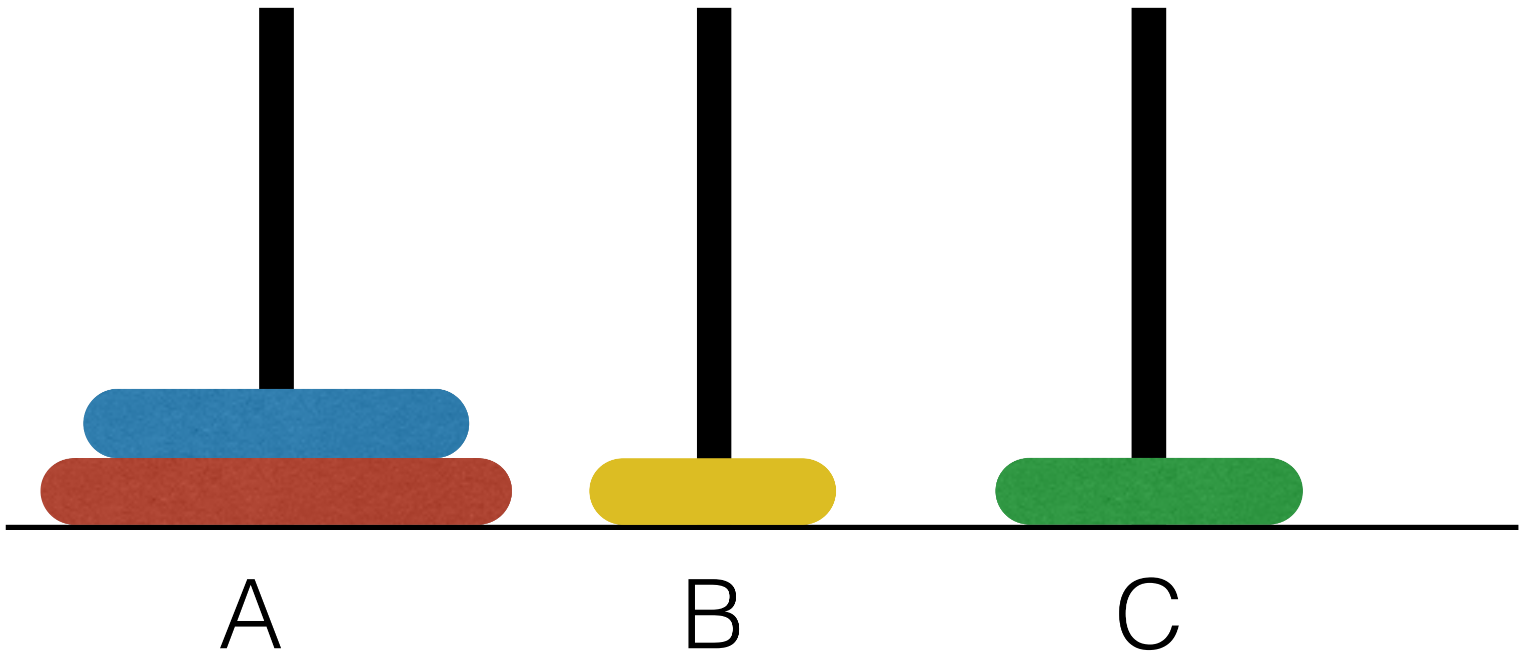
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

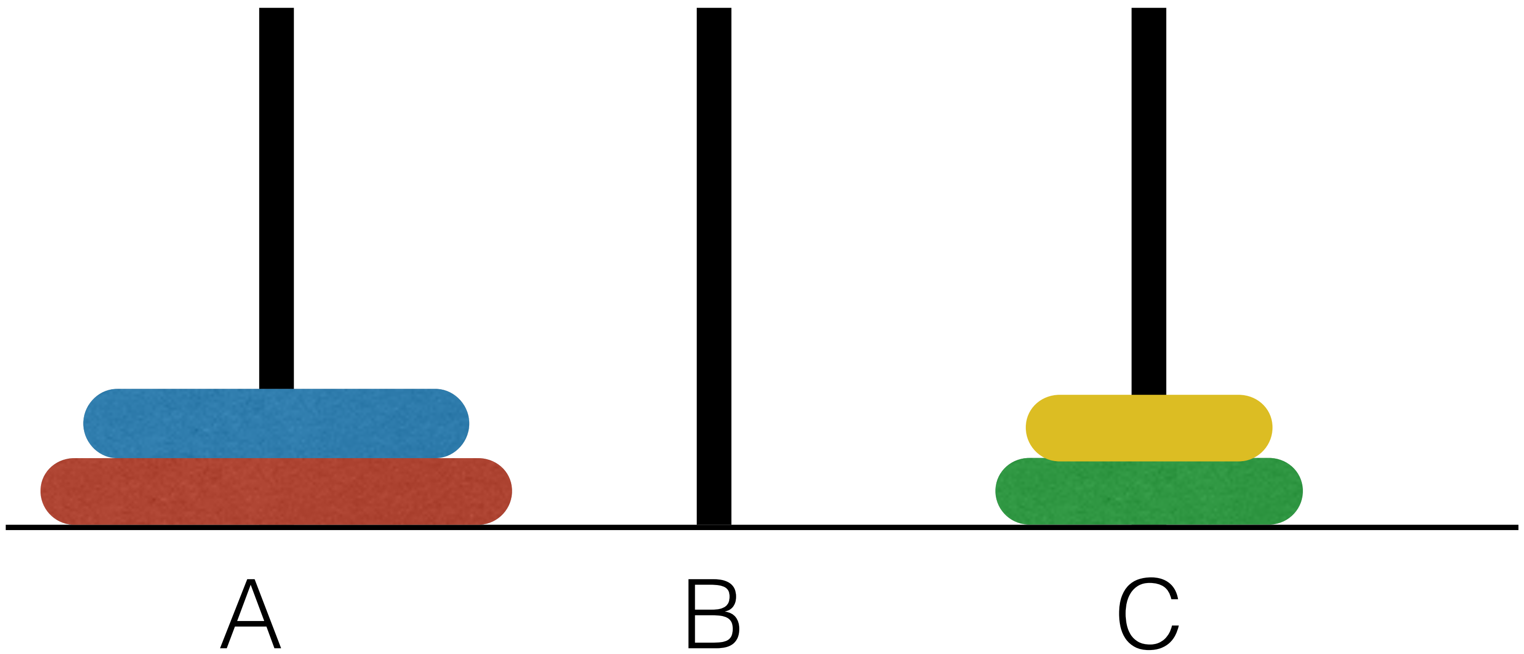
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

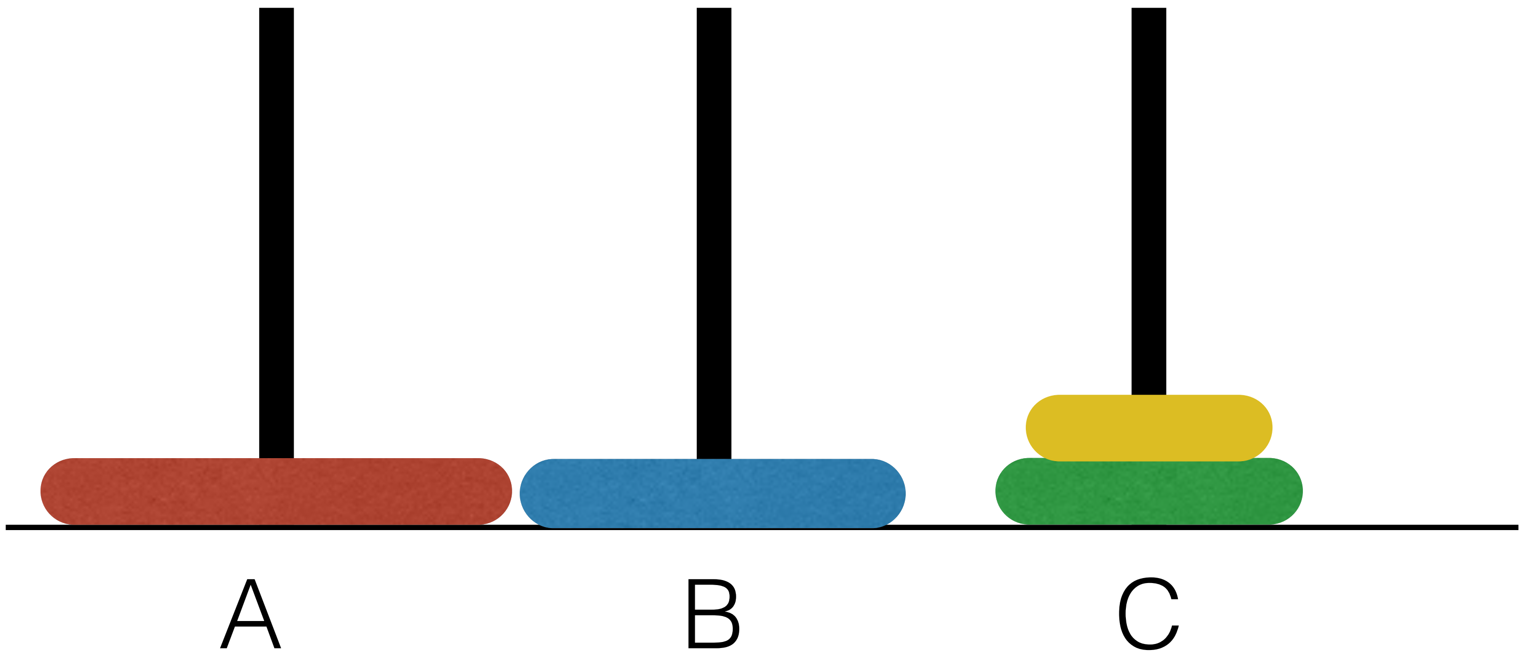
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

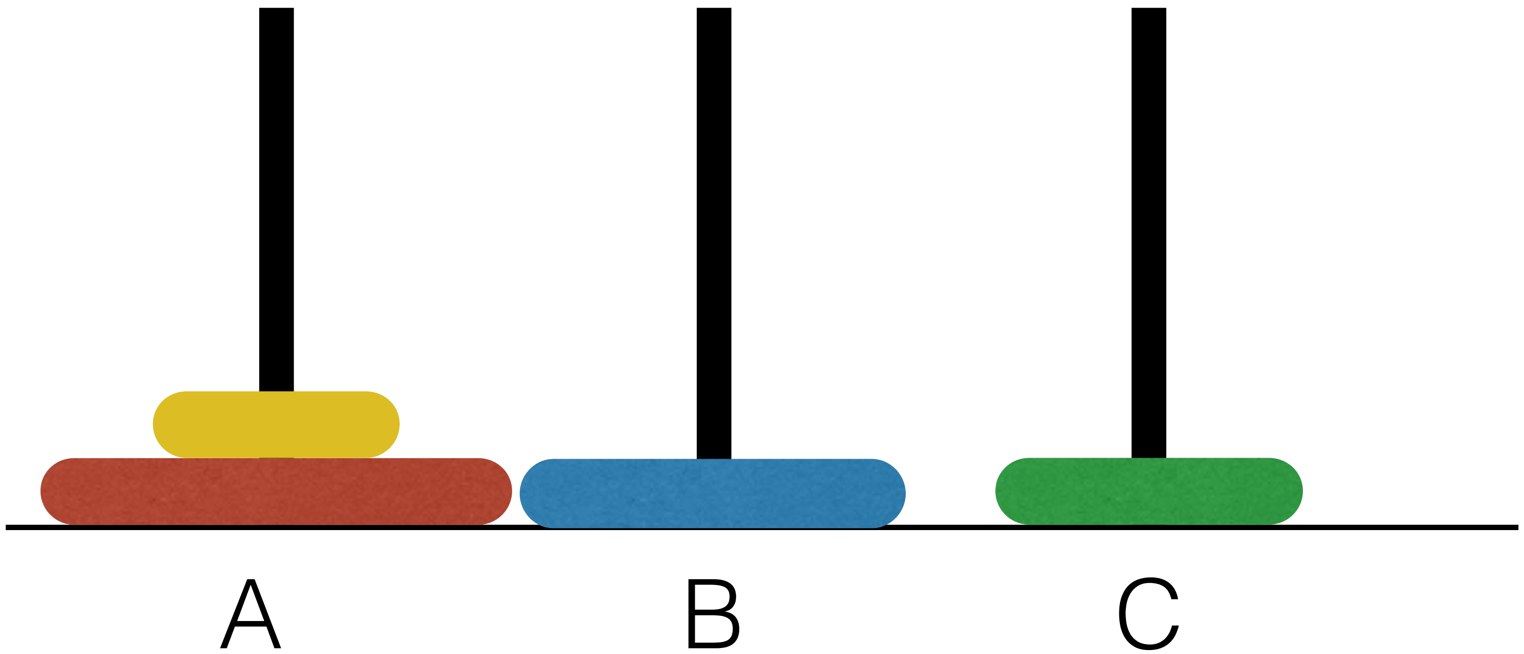
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

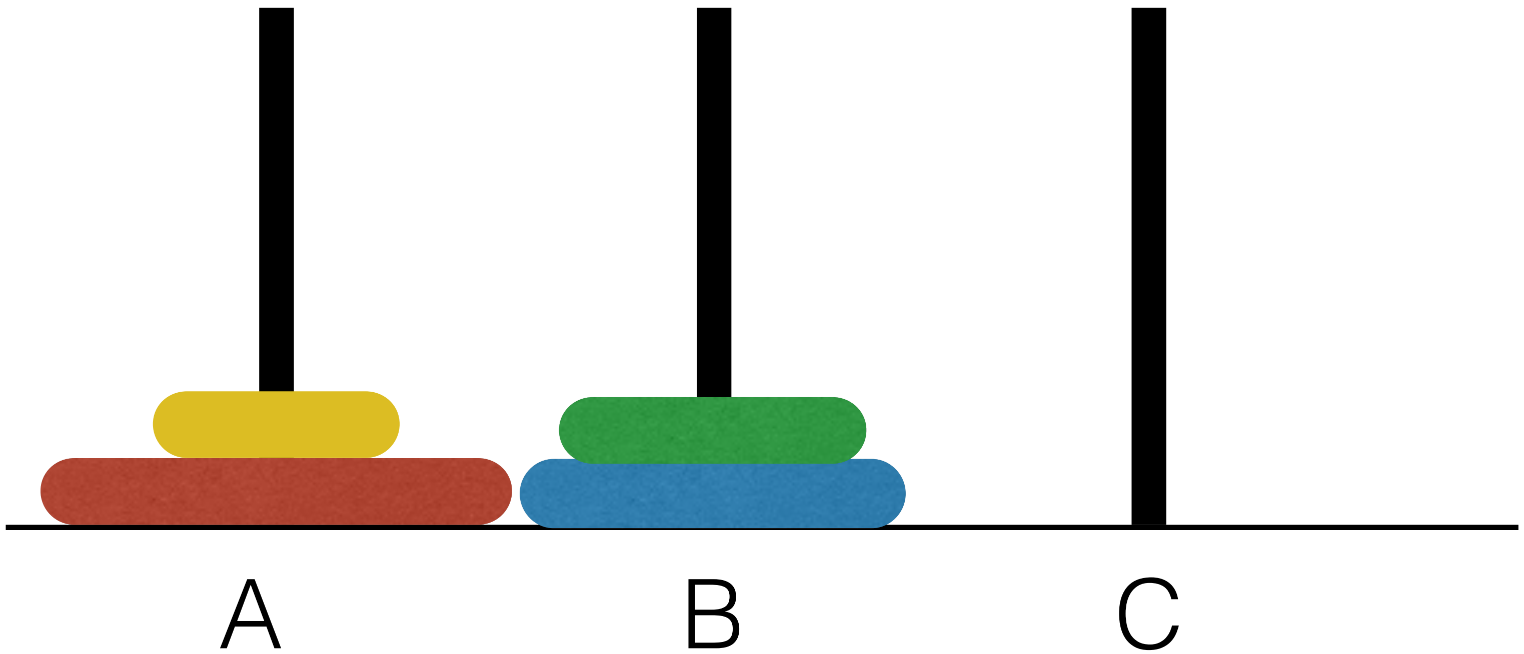
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

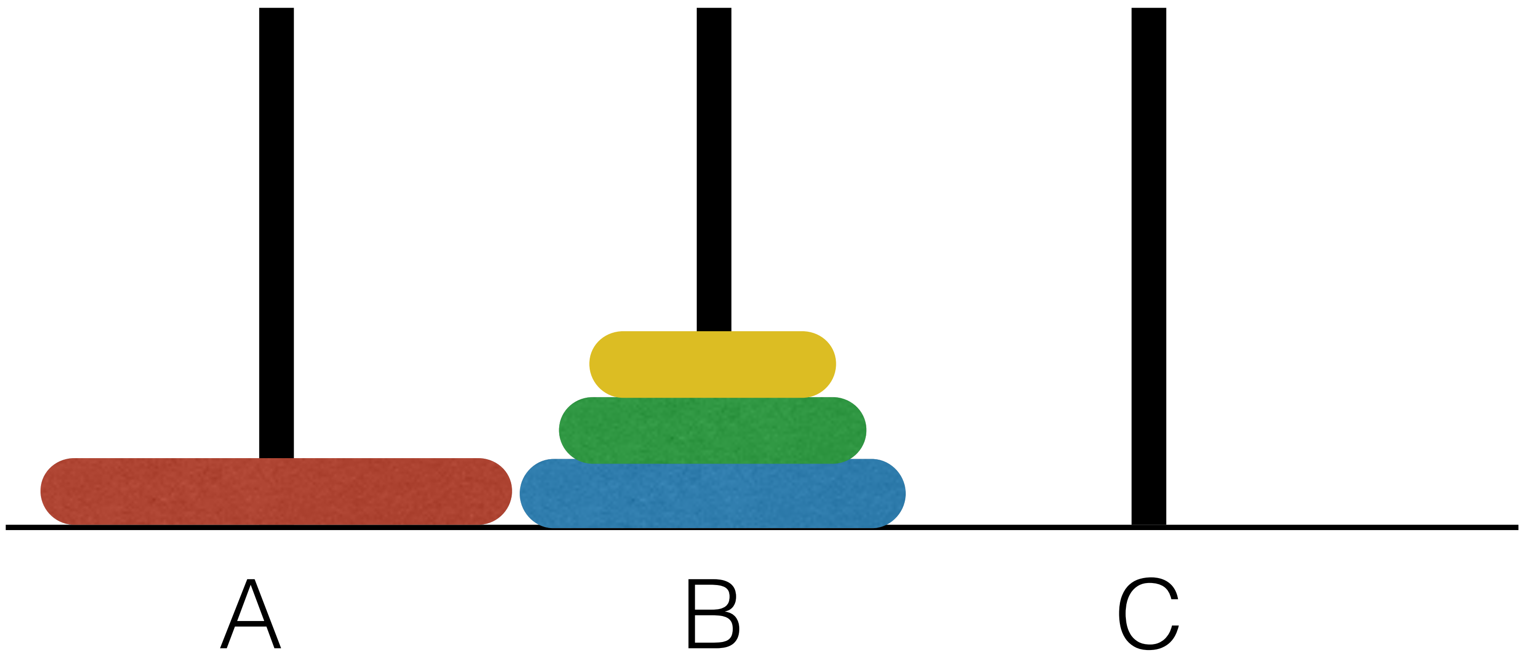
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

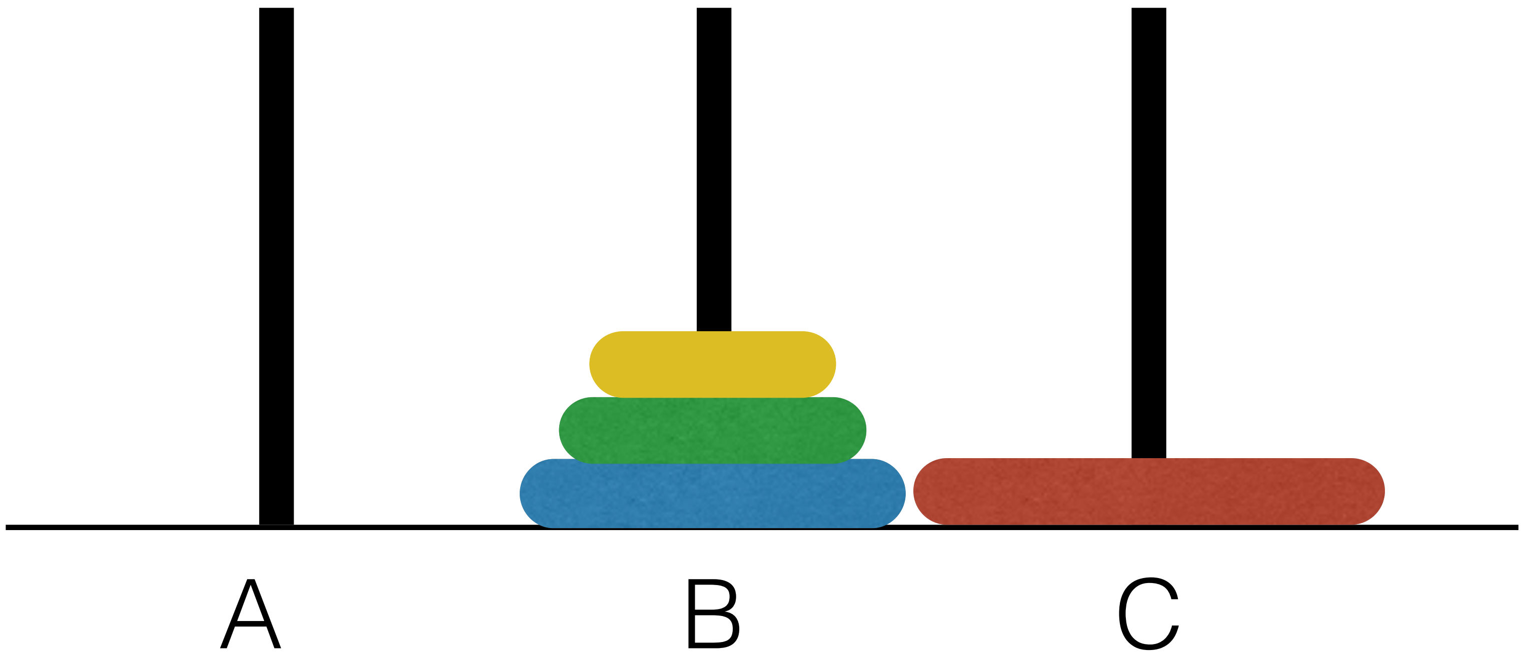
The Towers of Hanoi



Goal: Move all disks to the right peg

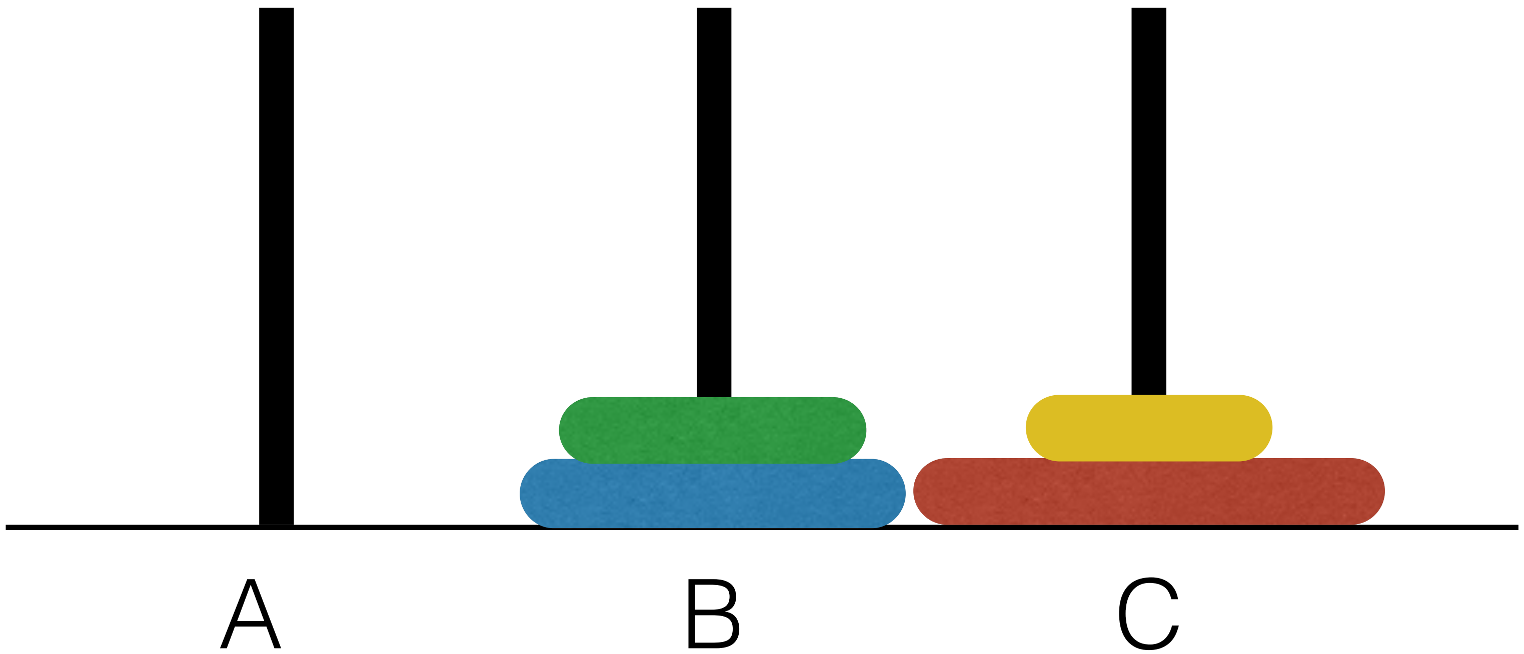
Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

The Towers of Hanoi



Goal: Move all disks to the right peg
Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

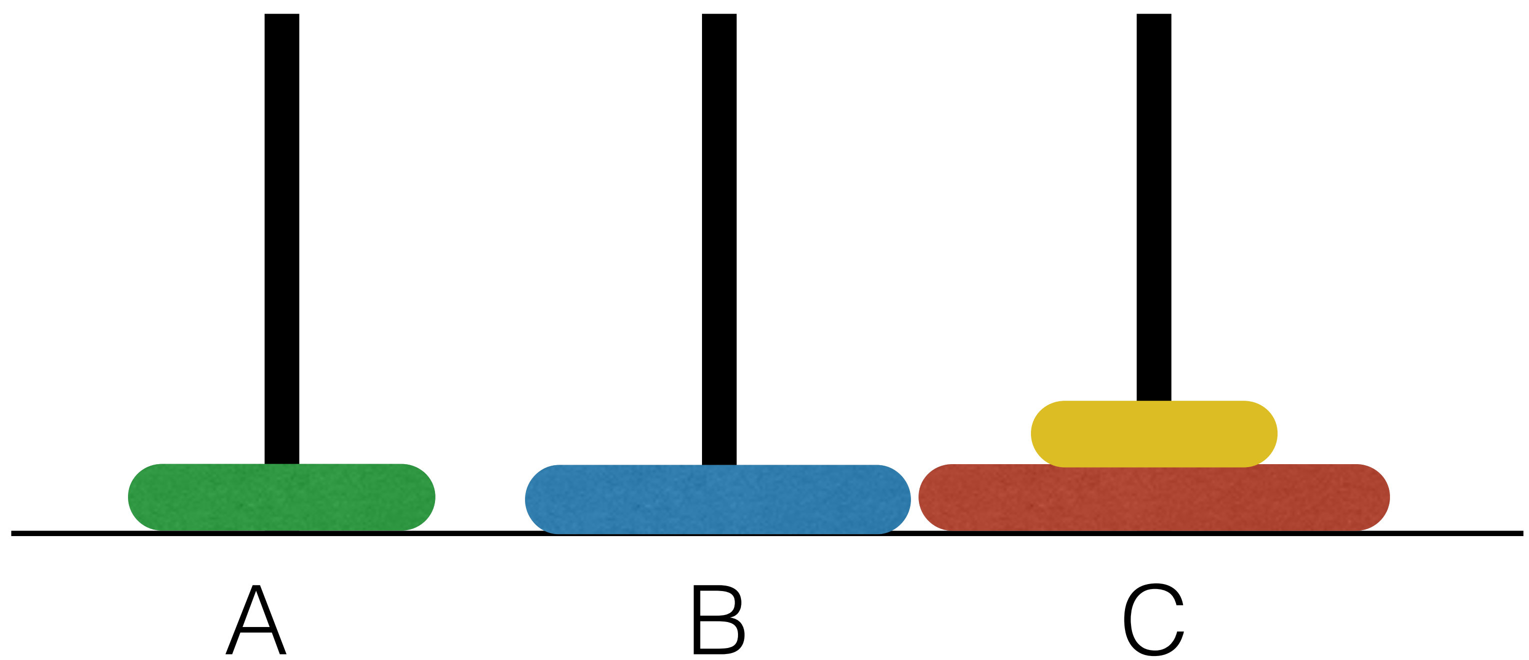
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

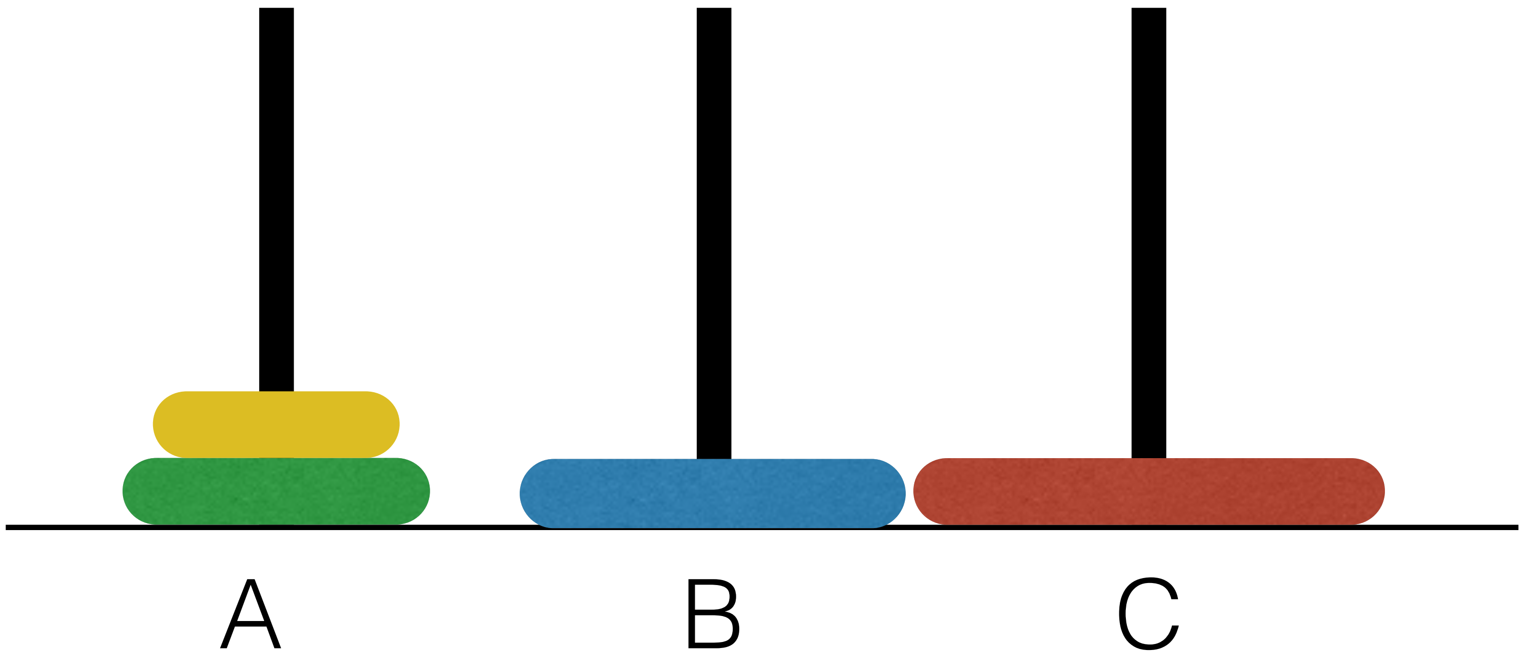
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

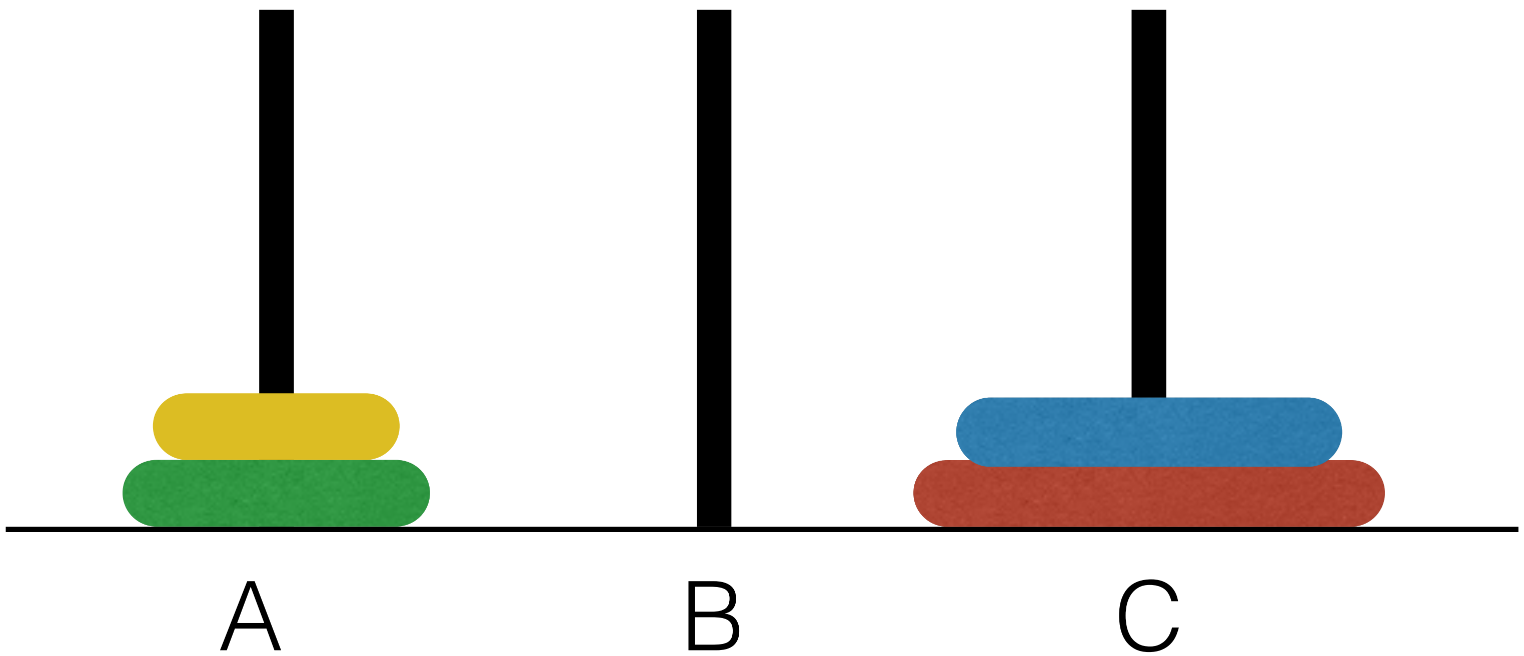
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

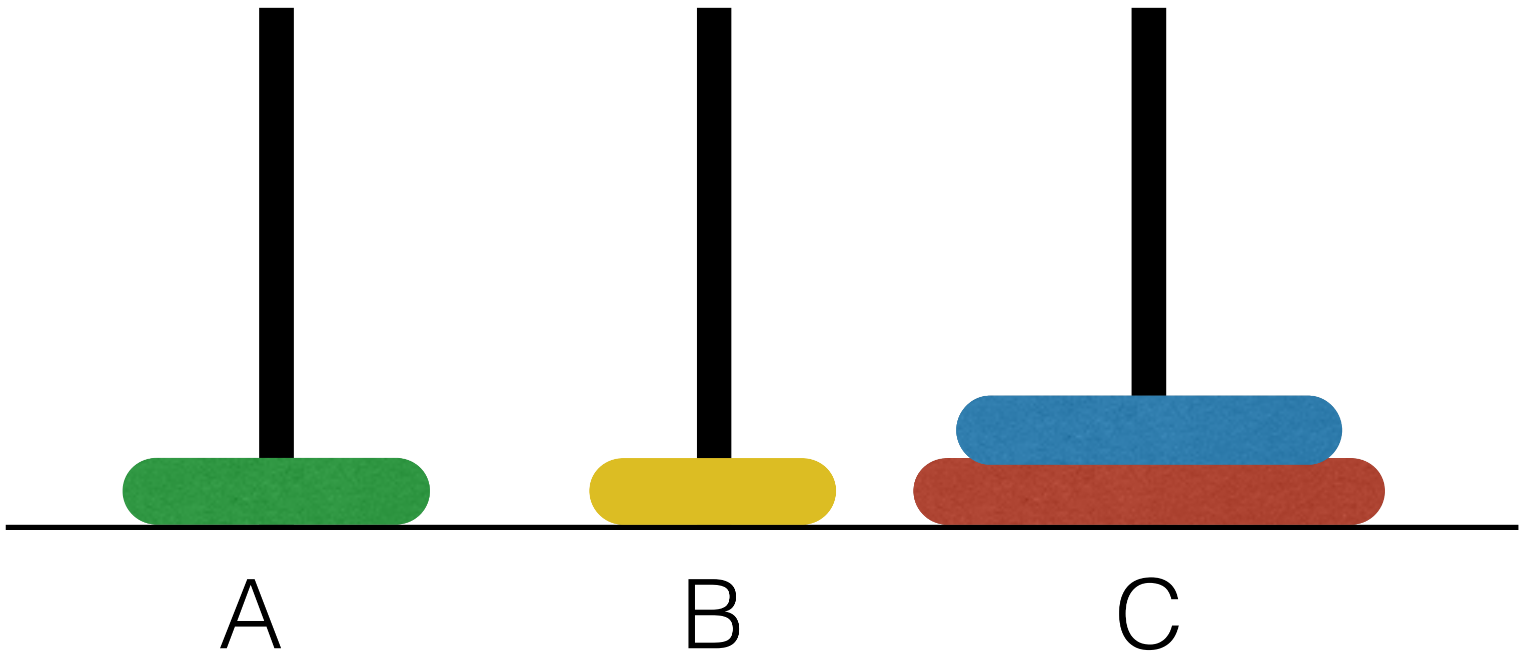
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

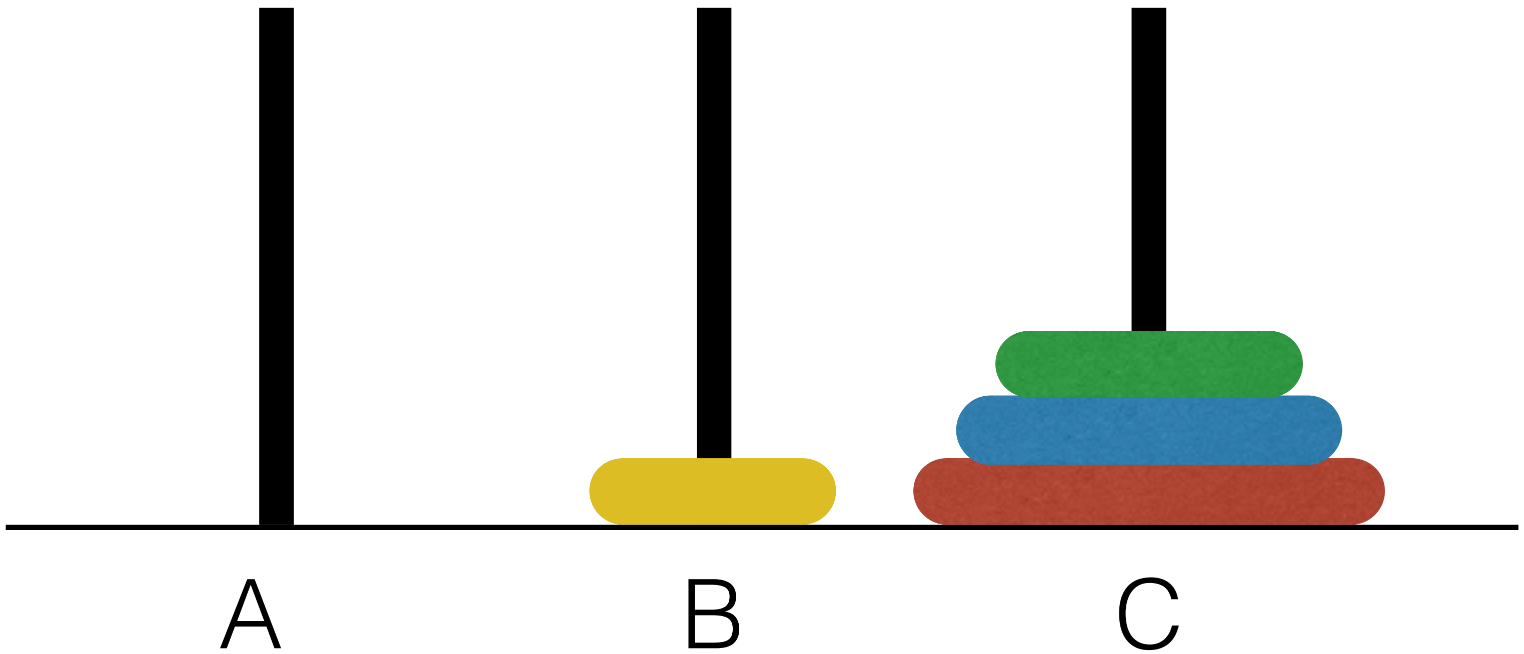
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

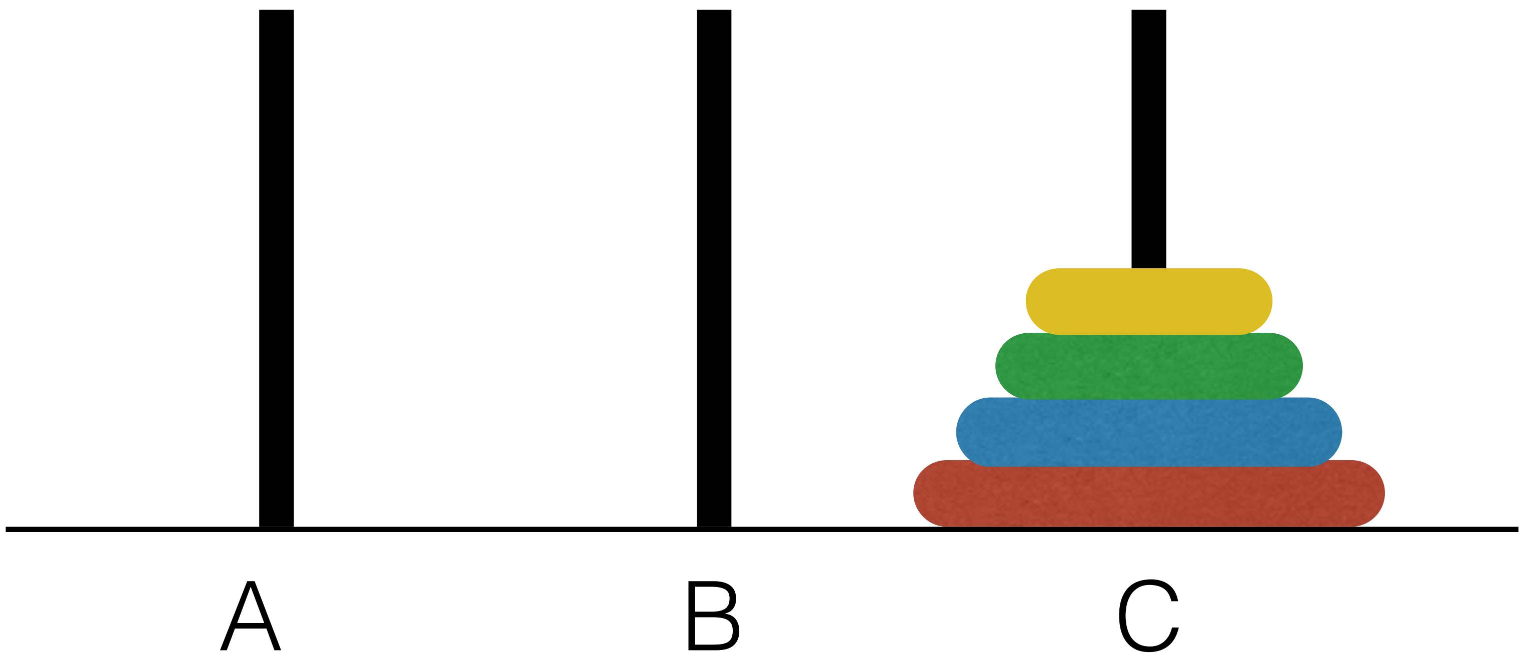
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

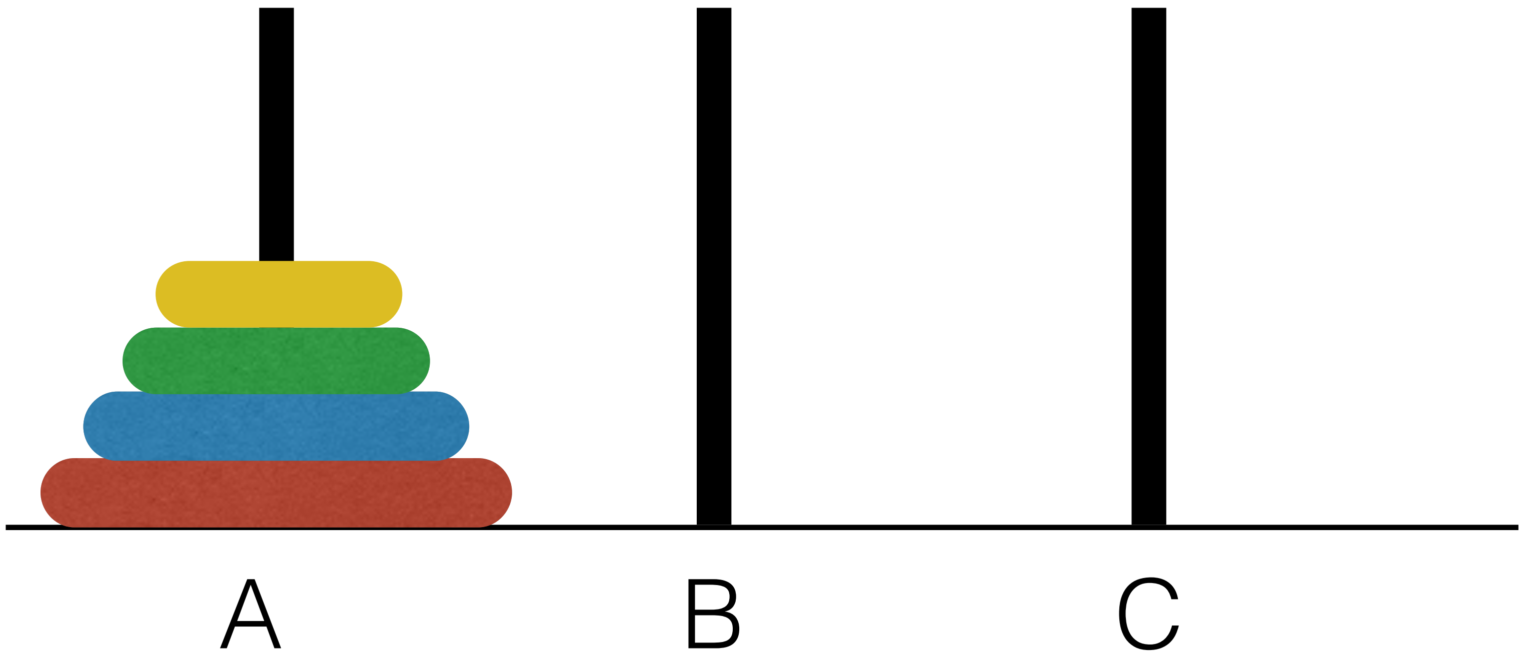
The Towers of Hanoi



Goal: Move all disks to the right peg

Moves: Take any disk on top of a stack and move it to the top of another stack. No disk may be placed on a smaller disk.

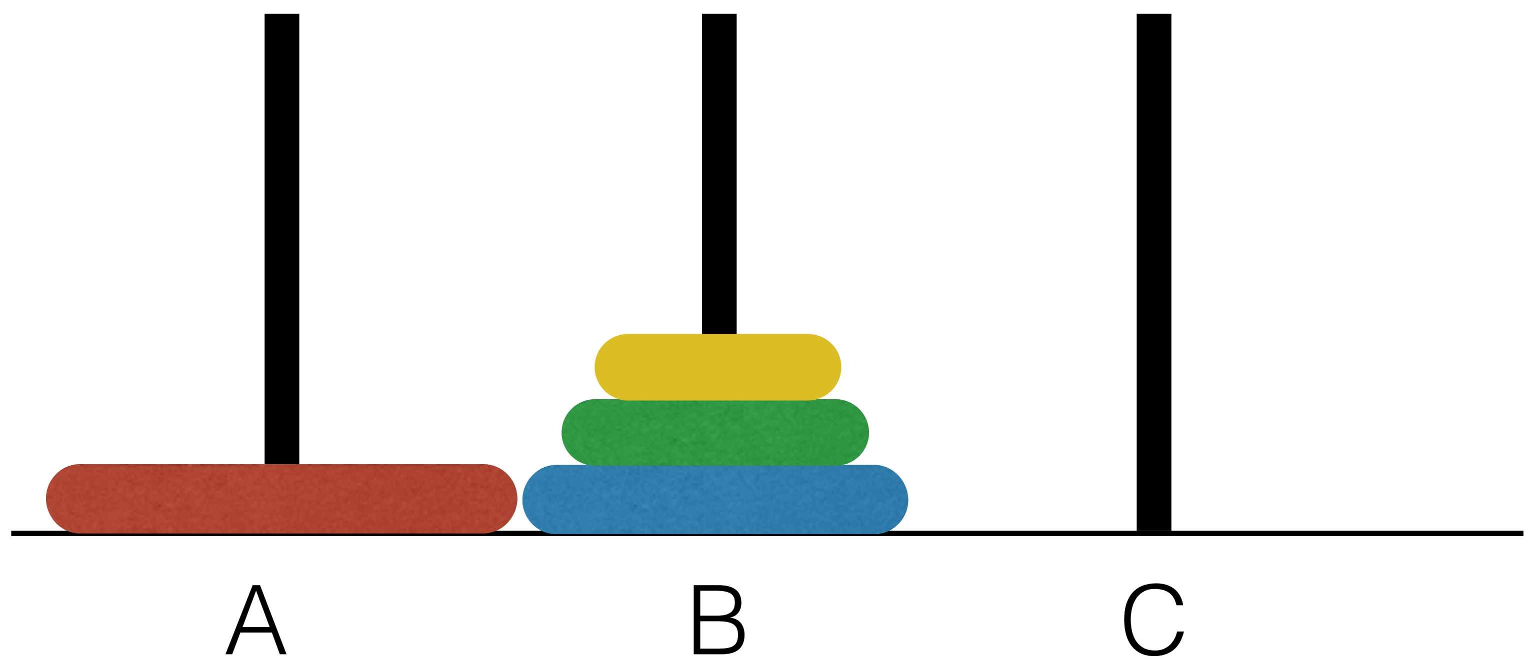
The Towers of Hanoi



Insight: To move 4 disks from A to C

1. move top three disks from A to B
2. move fourth disk to C
3. move top three disks from B to C

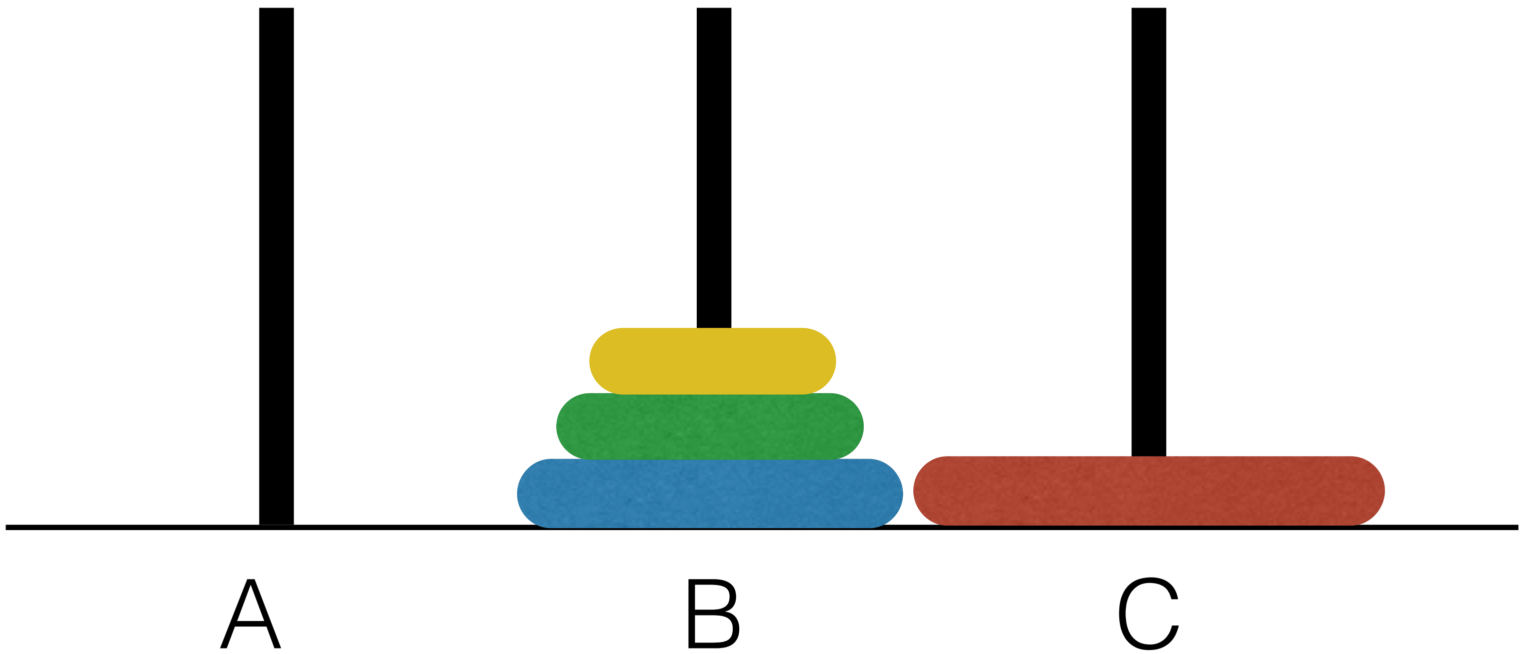
The Towers of Hanoi



Insight: To move 4 disks from A to C

1. move top three disks from A to B
2. move fourth disk to C
3. move top three disks from B to C

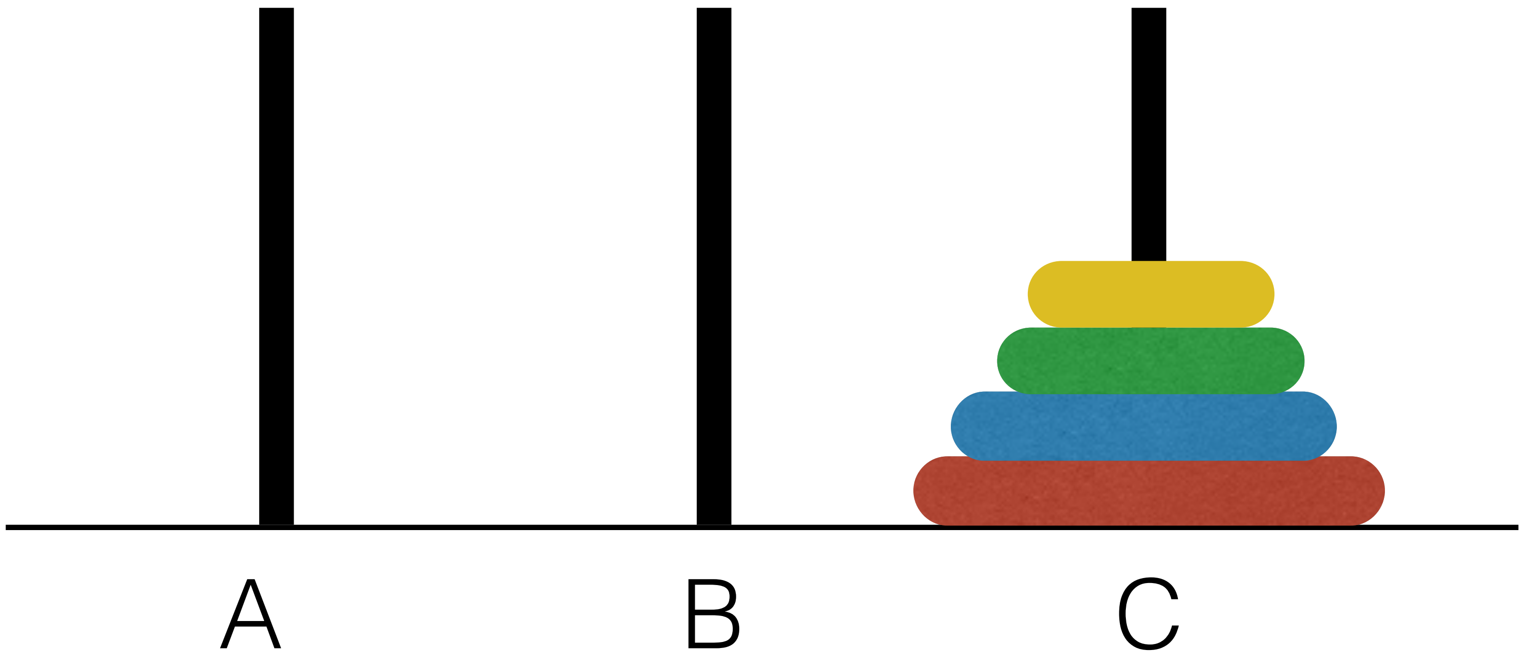
The Towers of Hanoi



Insight: To move 4 disks from A to C

1. move top three disks from A to B
2. move fourth disk to C
3. move top three disks from B to C

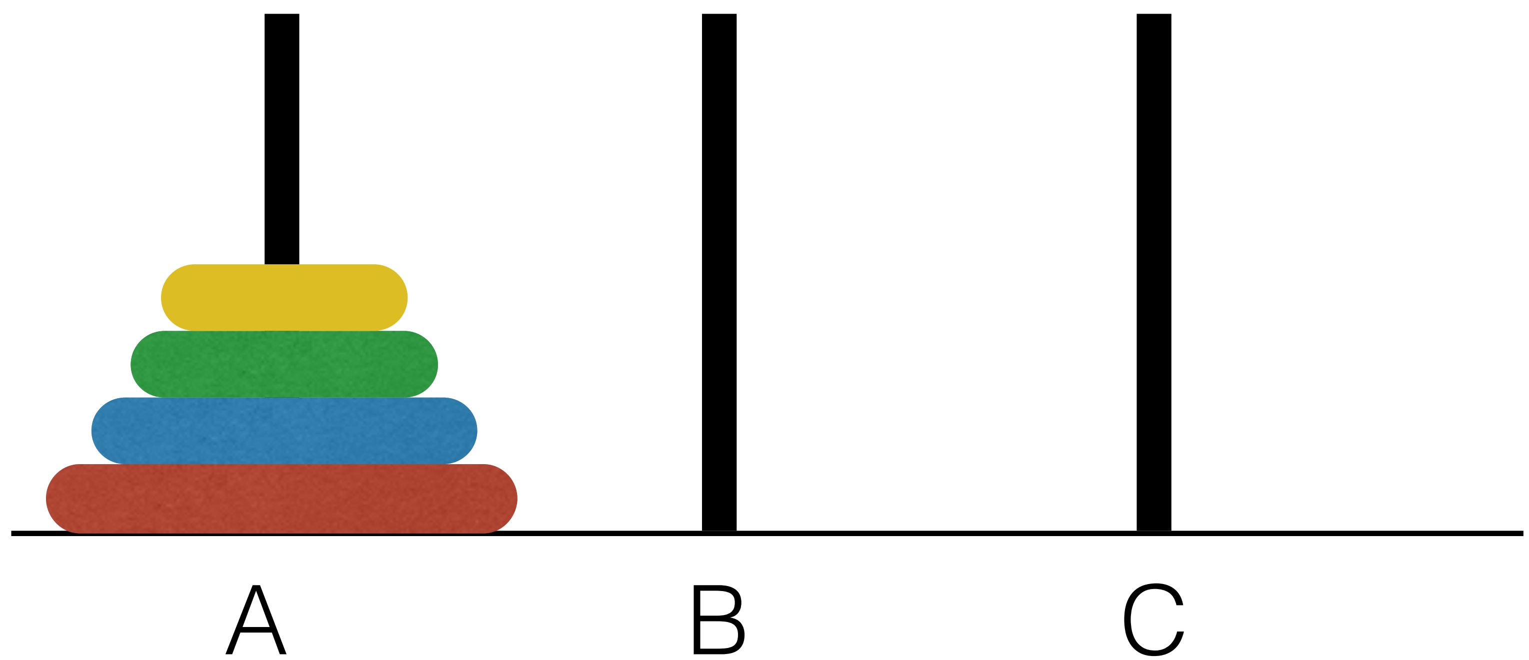
The Towers of Hanoi



Insight: To move 4 disks from A to C

1. move top three disks from A to B
2. move fourth disk to C
3. move top three disks from B to C

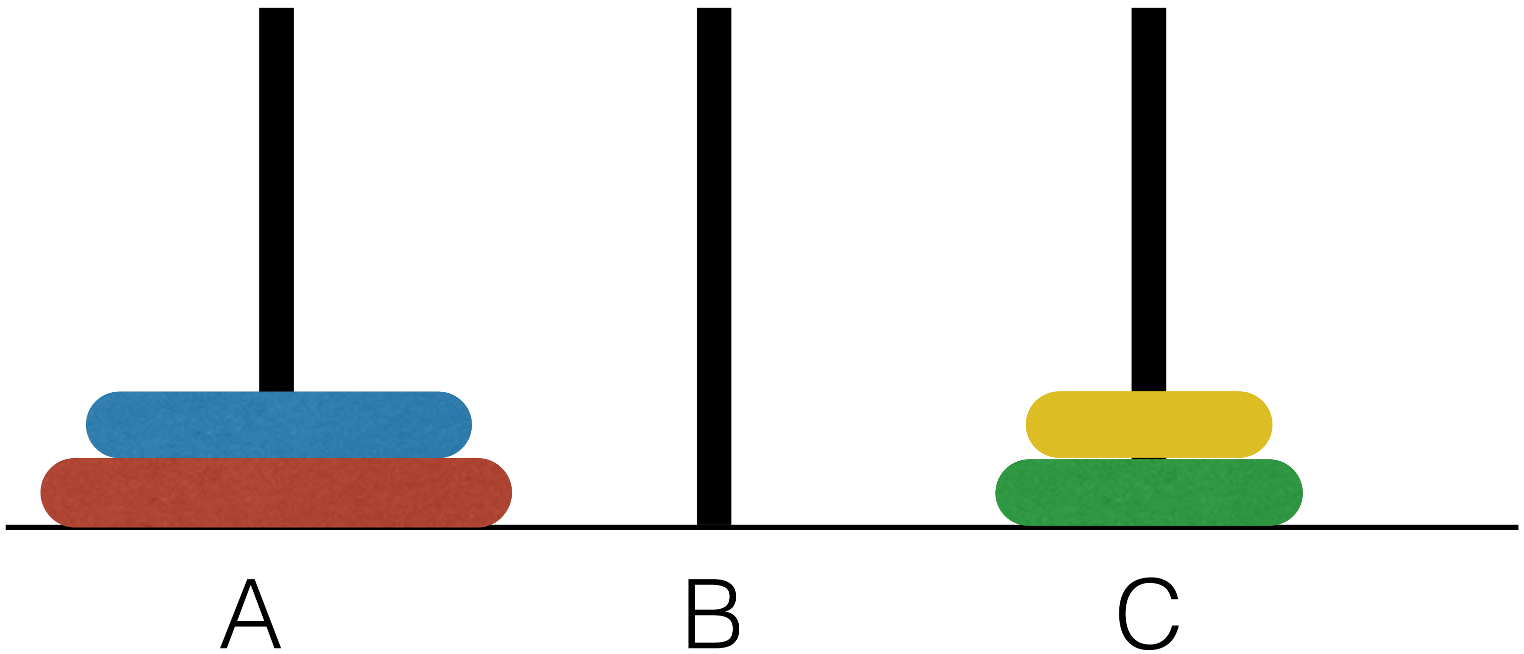
The Towers of Hanoi



Insight: To move 3 disks from A to B

1. move top two disks from A to C
2. move third disk to B
3. move top two disks from C to B

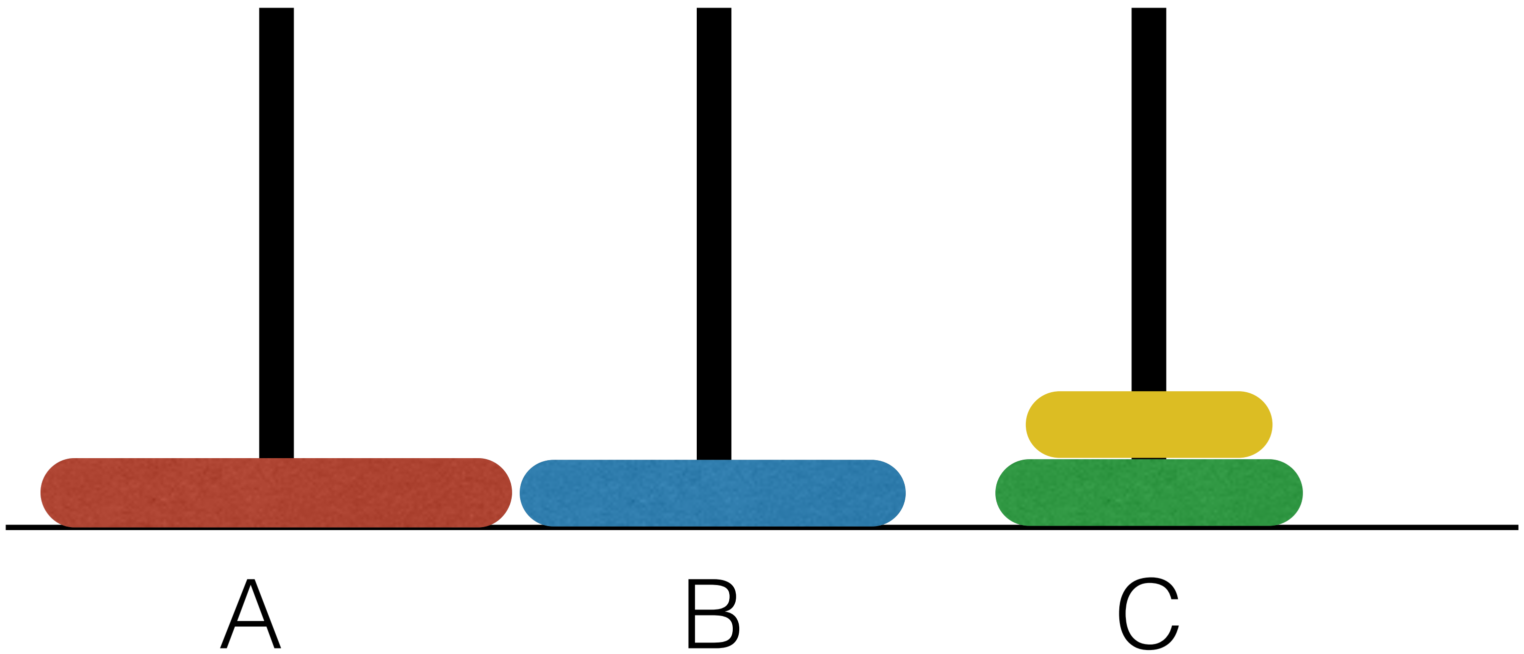
The Towers of Hanoi



Insight: To move 3 disks from A to B

1. move top two disks from A to C
2. move third disk to B
3. move top two disks from C to B

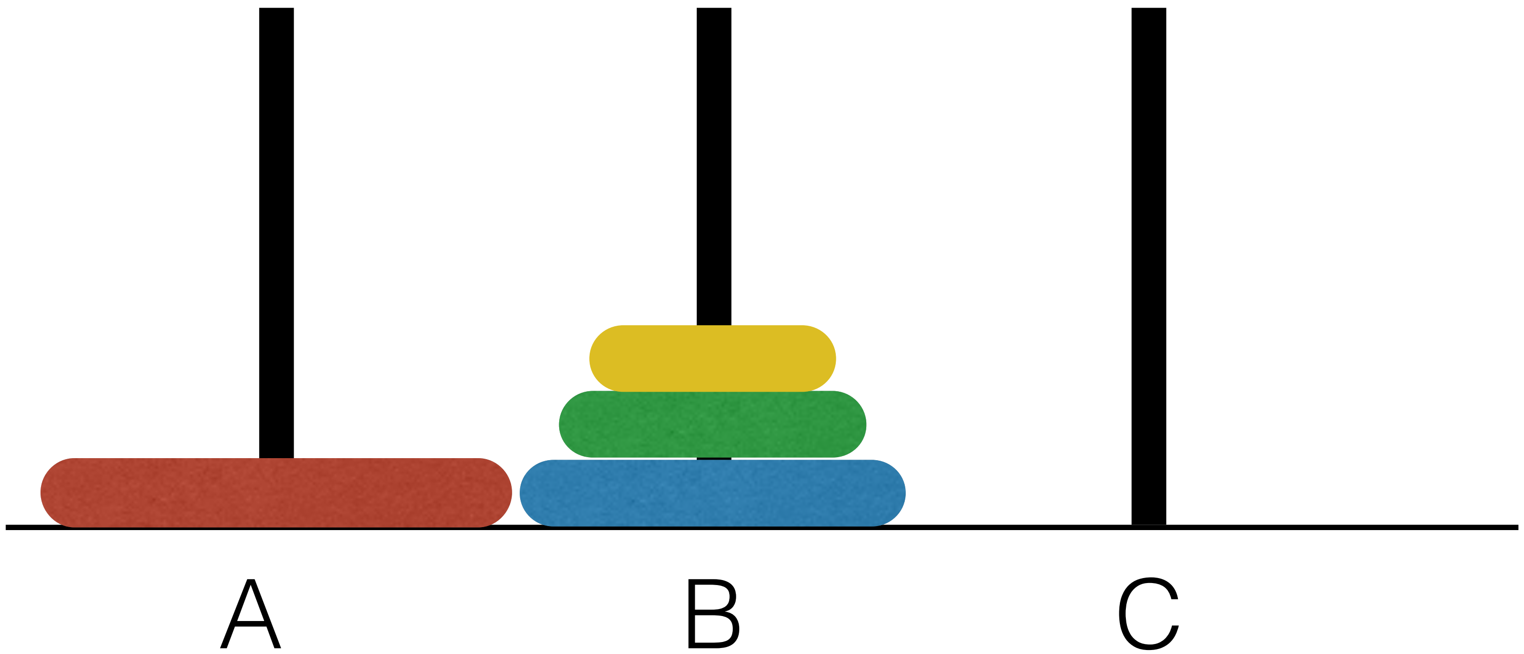
The Towers of Hanoi



Insight: To move 3 disks from A to B

1. move top two disks from A to C
2. move third disk to B
3. move top two disks from C to B

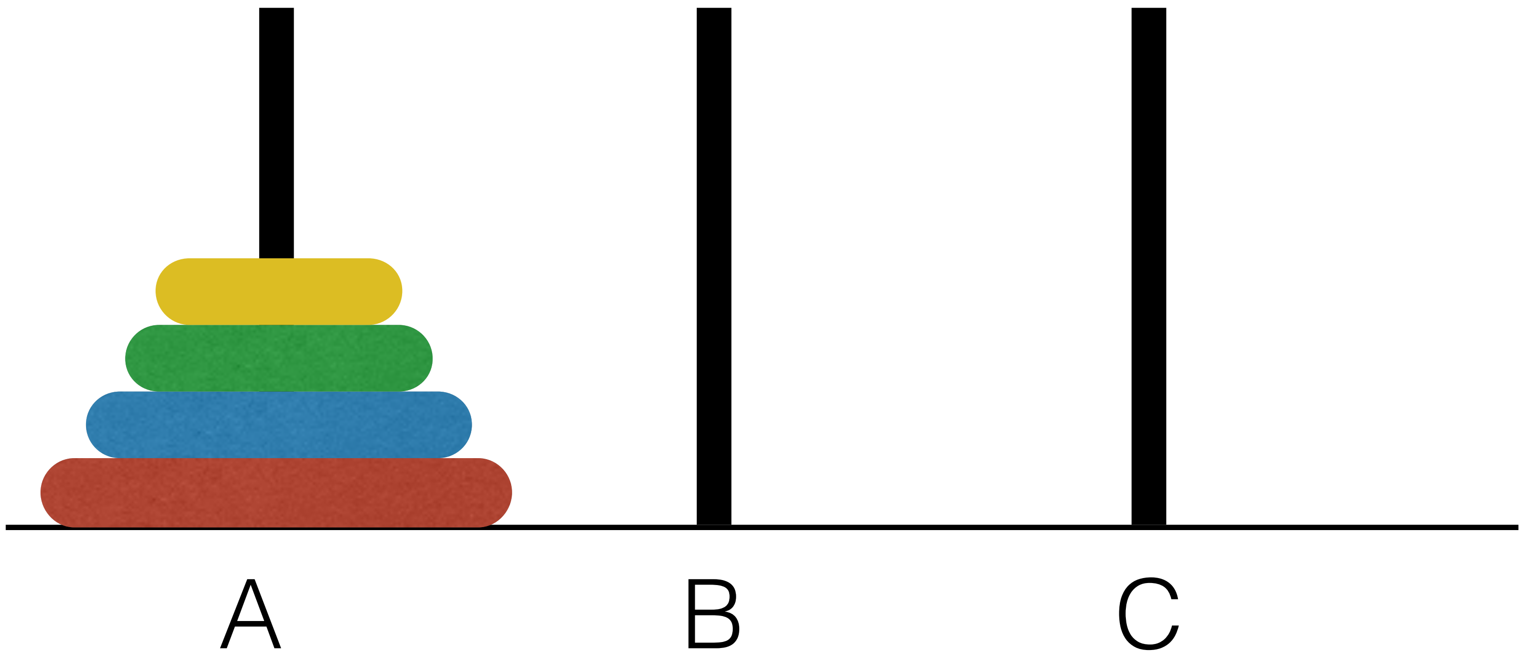
The Towers of Hanoi



Insight: To move 3 disks from A to B

1. move top two disks from A to C
2. move third disk to B
3. move top two disks from C to B

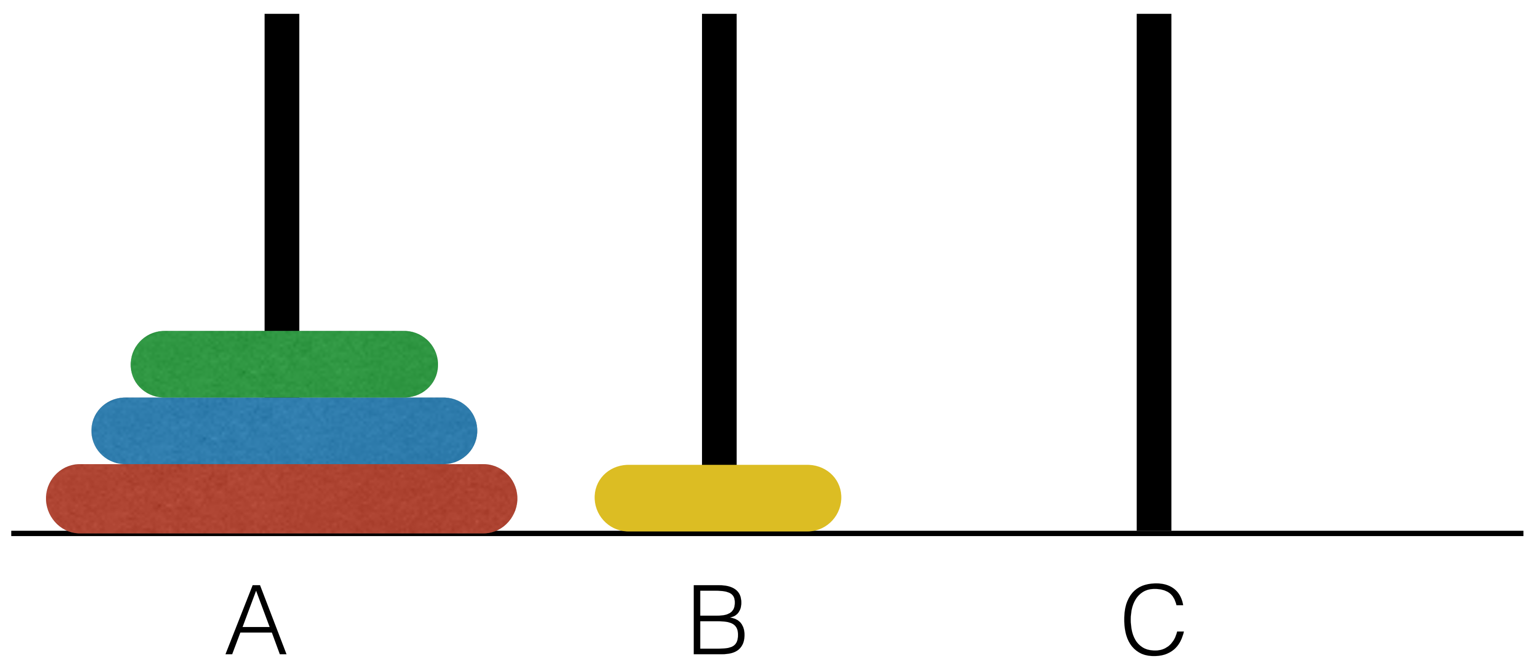
The Towers of Hanoi



Insight: To move 2 disks from A to C

1. move top one disks from A to B
2. move third disk to C
3. move top one disks from B to C

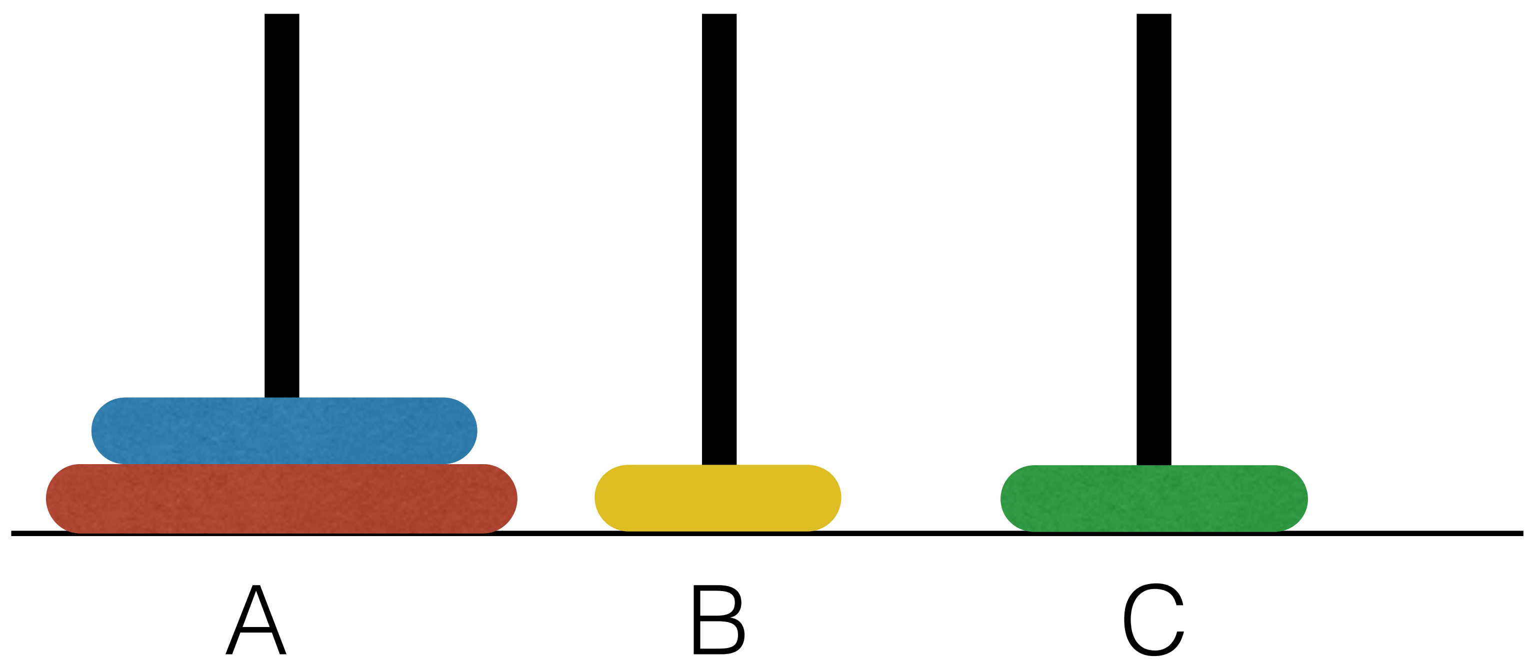
The Towers of Hanoi



Insight: To move 2 disks from A to C

1. move top one disks from A to B
2. move third disk to C
3. move top one disks from B to C

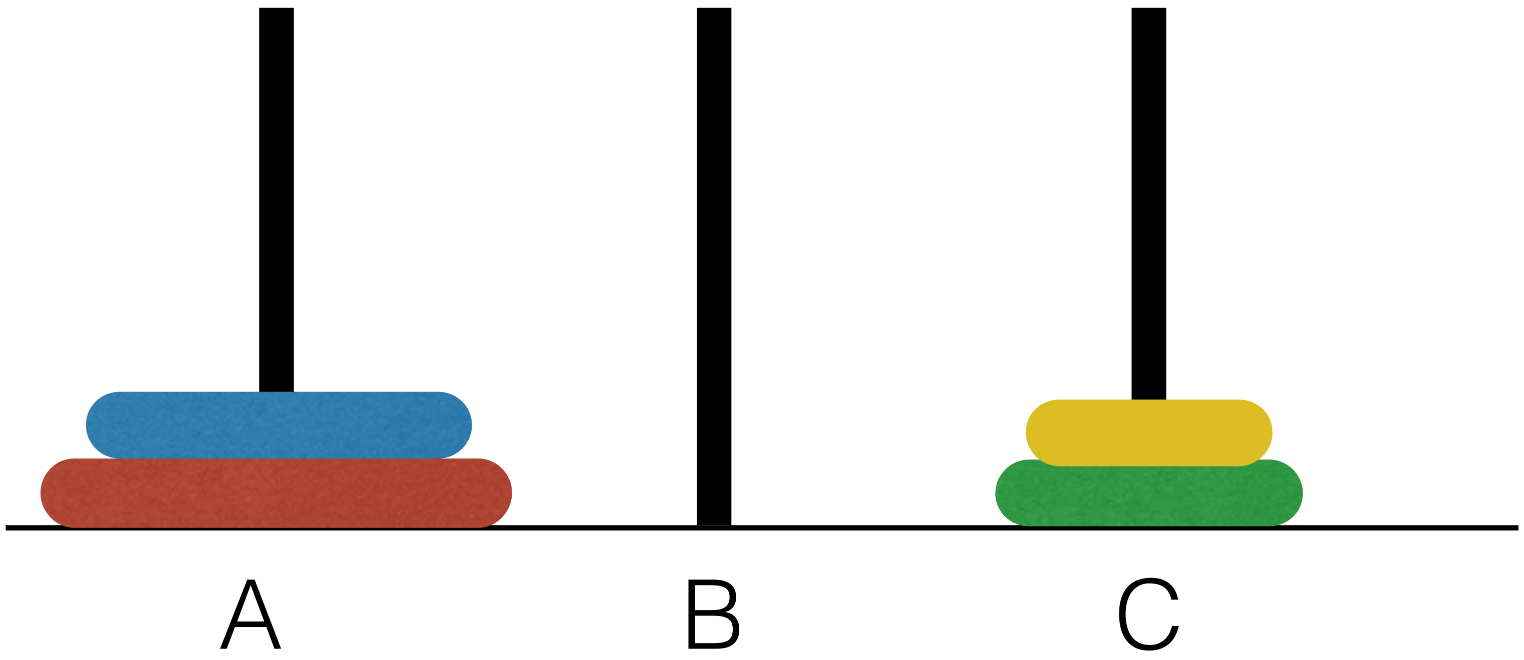
The Towers of Hanoi



Insight: To move 2 disks from A to C

1. move top one disks from A to B
2. move third disk to C
3. move top one disks from B to C

The Towers of Hanoi



Insight: To move 2 disks from A to C

1. move top one disks from A to B
2. move third disk to C
3. move top one disks from B to C

The Towers of Hanoi

Algorithm (sketch)

To move *n* disks from A to C

1. move top *n-1* disks from A to B
2. move *n*-th to C
3. move top *n-1* disks from B to C

A = source peg
C = target peg
B = “help” peg (to temporarily store disks)

Peg labels change in each recursive call.

The Towers of Hanoi

To move *n* disks from A to C

1. move top *n-1* disks from A to B
2. move *n*-th to C
3. move top *n-1* disks from B to C

$$T(N) = 2 \cdot T(N-1) + 1$$

$$T(1) = 1$$

Need to solve this recurrence relation!