Last Lecture

- Postfix Expression (e.g. **`2 14 + 23 *`**). No parentheses; easy to evaluate using a stack.
- So far we've assumed **binary** operators. What about **unary** operators, such as the negate operator: **`!`**
- A unary operator requires just one operand. So when you see a unary operator, pop **one** operand off the stack, apply the operator, and push it back onto the stack.
- Example: **`2 ! 5 ! - 1  4 ! - *`**
- You will encounter this in project 5.

This Lecture

- Thinking **Recursively.**
- What is Recursion, some basic examples.
- Three questions about a recursive algorithm.
- Stack Frames and StackOverflowException
- Tower of Hanoi.

What is Recursion?

- A method that calls itself:

```
void recursiveMethod() {
  … …
  recursiveMethod();
}
```

- Like self-referential structures (e.g. `LLNode`), the compiler is totally cool with this.
- A conceptually simple way to solve many problems (although might not be computationally efficient).

What is Recursion?

- Recursion can also take the form of two or more methods that call each other and form a cycle.

```
void first() {
  … …
  second();
}
void second() {
  … …
  first();
}
```

Basic Recursion Examples

- Let's write a method **`intsum(n)`** that computes the sum of integers from 1 to n.
- We already know two ways to write this method.

1. Write a loop to compute the sum from 1 to n.
2. Or mathematically **`intsum(n) = n(n+1)/2`**

- But we can also solve it recursively, that is, **`intsum(n)`** is equal to the sum of integers from 1 to (n-1) plus n. In other words:

```
intsum(n) = intsum(n-1) + n
```

Basic Recursion Examples

- To write down this idea in code:

```
int intsum(int n) {
  return intsum(n-1) + n;
}
```

- Think about “passing the buck” (i.e. the responsibility).
  - But the buck has to stop at some point!
  - When n=1, we know the answer is 1, so we can return without making further recursion.
  - This is called the ‘**base case**’.

Basic Recursion Examples

- To write down this idea in code:

```
int intsum(int n) {
  if(n==1) return 1;
  else return intsum(n-1) + n;
}
```

- Even with this base case, you still need to be careful, for example, what happens if I call **`intsum(0)`** or **`intsum(-1)`**? We will come back to this question in a little bit.

Basic Recursion Examples

- Recusion:

```
int intsum(int n) {
  if(n==1) return 1;
  return intsum(n-1) + n;
}
```

- While loop:

```
int total = 0;
While (n != 0) {
  total += n--;
}
```

Computing Factorials

- Given a non-negative integer n, its **factorial**, written as **`n!`** is the **product** of all integers from 1 to n.

  `n! = 1 x 2 x 3 x 4 .... x (n-1) x n`

- For example: `5! = 1 x 2 x 3 x 4 x 5 = 120`
- This is similar to `intsum(n)`, except using multiplication.
- Do you know what is `0!` equal to?
- In terms of growth speed, factorial **grows extremely fast**, exceeding even the exponential family. A well-known example is writing down all possible permutations of n elements.

Computing Factorials

- Similar to before, you can write down a loop to calculate the factorial. But you can also think about the problem recursively:

```
factorial(n) = factorial(n-1) * n;
factorial(0) = 1  // base case
```

- This translates directly to code:

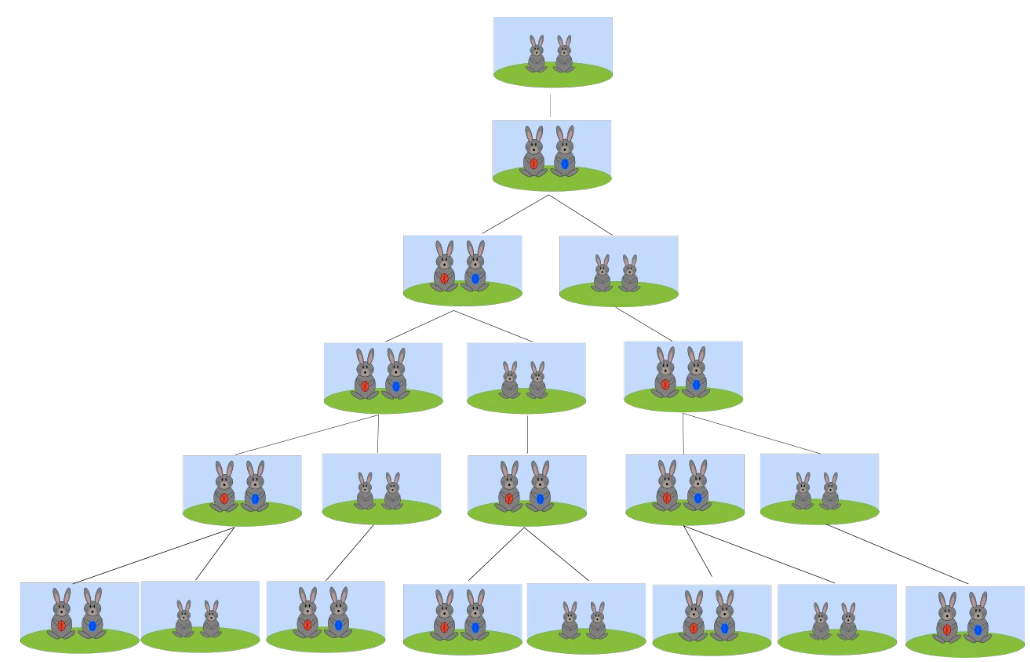
```
int factorial(int n) {
  if(n==0) return 1;
  else return (factorial(n-1)*n);
}
```

Fibonacci Numbers

- Now let’s look at a slightly more complex problem: the **Fibonacci** numbers. The story came from a simple mathematical model of reproducing rabbits.

- The Fibonacci numbers are defined recursively:

```
F(n) = F(n-1) + F(n-2)
F(0) = 1
F(1) = 1
```



- In sequence:

```
1, 1, 2, 3, 5, 8, 13, 21
```

Fibonacci Numbers

- This directly translates to the following code:

```
int Fibonacci(int n) {
   if(n==0 || n==1)
     return 1;
   else
     return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Clicker Question #1

```
public int foo (int n) {
    if (n <= 1) return 0;
    return 1 + foo(n-2);
}
```

What's the return value of this method if it is called with a positive integer **n** as parameter?

(a) n

(b) n/2+1

(c) n/2

(d) 2*n

(e) for some values of n it never ends

3 questions about Recursion

- For a recursive algorithm to work correctly on a given input value, we need to verify the following three questions:

1. Does the algorithm have a **base case**?
2. Does every recursive call make **progress towards the base case**?
3. Does the call to the algorithm get the right answer if we assume that all subsequent recursive calls get the right answer (i.e. **induction**)?

Three Questions

- A **base case** is where there is no further recursion. Our factorial algorithm has a base case of n = 0.
- We need to guarantee **progress towards the base case**. For example, in the factorial example, parameter n gets smaller at each recursion.
- We can **justify correctness.** For example, assuming `factorial(n-1)` returns the correct result, we know that `factorial(n-1)*n` gives the correct result of `factorial(n)`.

Clicker Question #2

```
public void clear(Stack<Integer> s) {
    if (!s.isEmpty()) {
        s.pop();
        clear(s);
    }
}
```

What is the **base case** of this recursive method?

(a) when `s.isEmpty()` returns true.

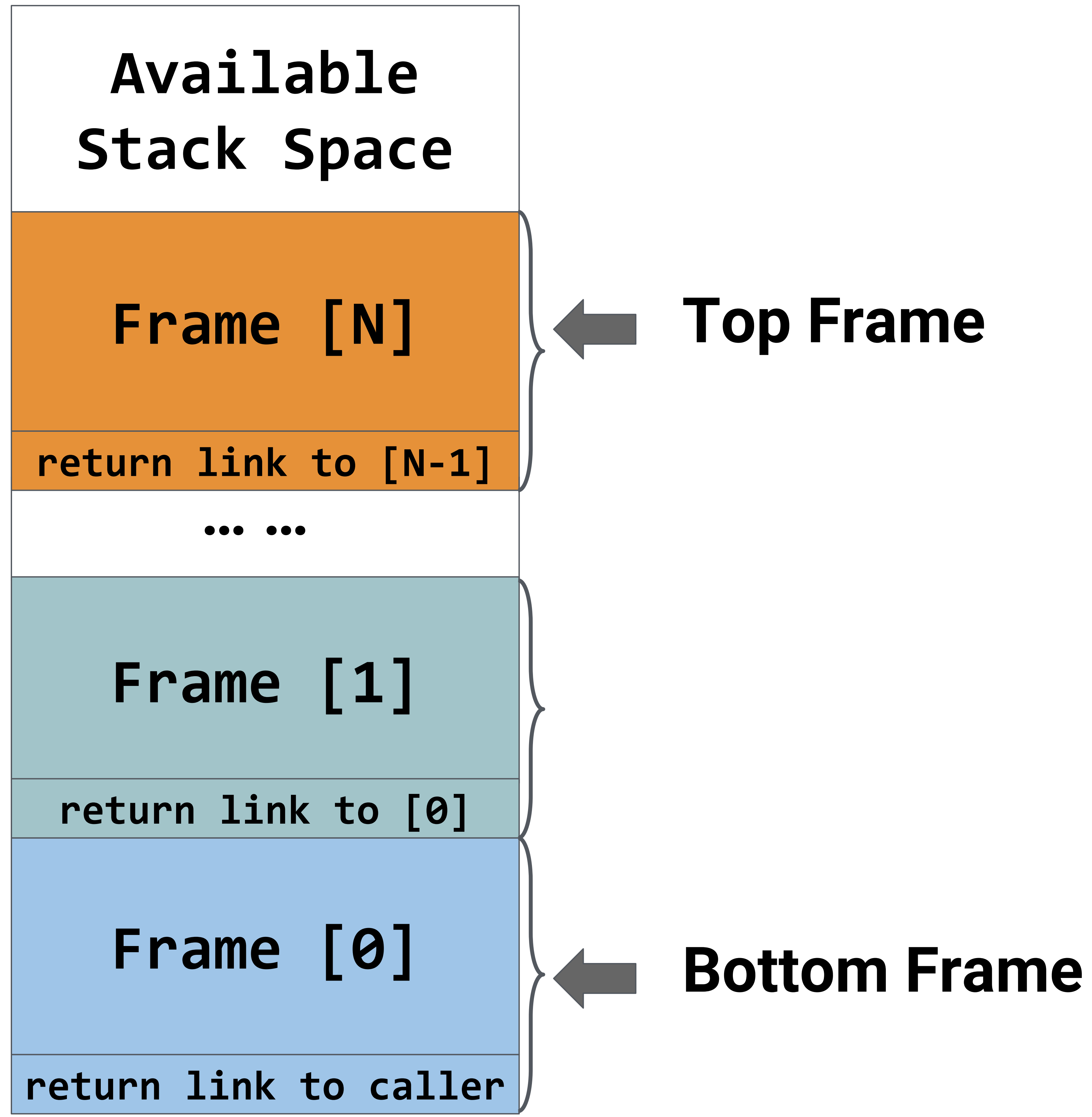
(b) when a `StackUnderflowException` is thrown.

(c) when `clear(s)` throws an exception.

(d) this method has no base case.

Program Stacks

- What's really going on with recursion? How does the computer system handle recursive calls?
- Computers use **stacks** to handle all method calls and returns. Method calls are executed in **Last-In-First-Out (LIFO)** fashion (recall the 'clean your house' example and all the additional tasks it incurred).
- A method's local variables and return link are kept in the stack memory. This is called a **stack frame**.
- The run-time system has a limited amount of stack memory. Beyond that you will get `StackOverflowException`.


Available
Stack Space
Frame [N]
return link to [N-1]
... ...
Frame [1]
return link to [0]
Frame [0]
return link to caller
Top Frame
Bottom Frame

Program Stacks

- Local variables (including the method's arguments or parameters) are allocated in program stack space.
- When **calling** a method, the local variables and return link to the current method (caller) are preserved in the stack; then the stack pointer moves up to the new frame (callee).
- Upon **returning**, the stack pointer moves back to the caller's frame, allowing computation to continue there.
- This calling mechanism is the same **for all methods calls**, regardless of whether a method calls itself or another method.

Recursion

- To the compiler and the run-time system, there is no distinction between recursive vs. non-recursive calls, they are treated the same way.
- Each stack frame keeps a separate copy of the method's local variables (remember, arguments are also local variables).
- Conceptually, it's easy to understand recursion in a top-down manner (i.e. `n! = (n-1)! * n`)
- Computationally, it's really done from bottom-up.
- Show how `factorial(5)` works.

Clicker Question #3

```
public int sum2(int n) {
    if(n==0)
      return 0;
    else
      return n+sum2(n-2);
}
```

What happens if we call `sum2(23)`?

(a) the return value is 23*(23+1)/2 = 276.

(b) the return value is 23*(23+1)/4 = 138.

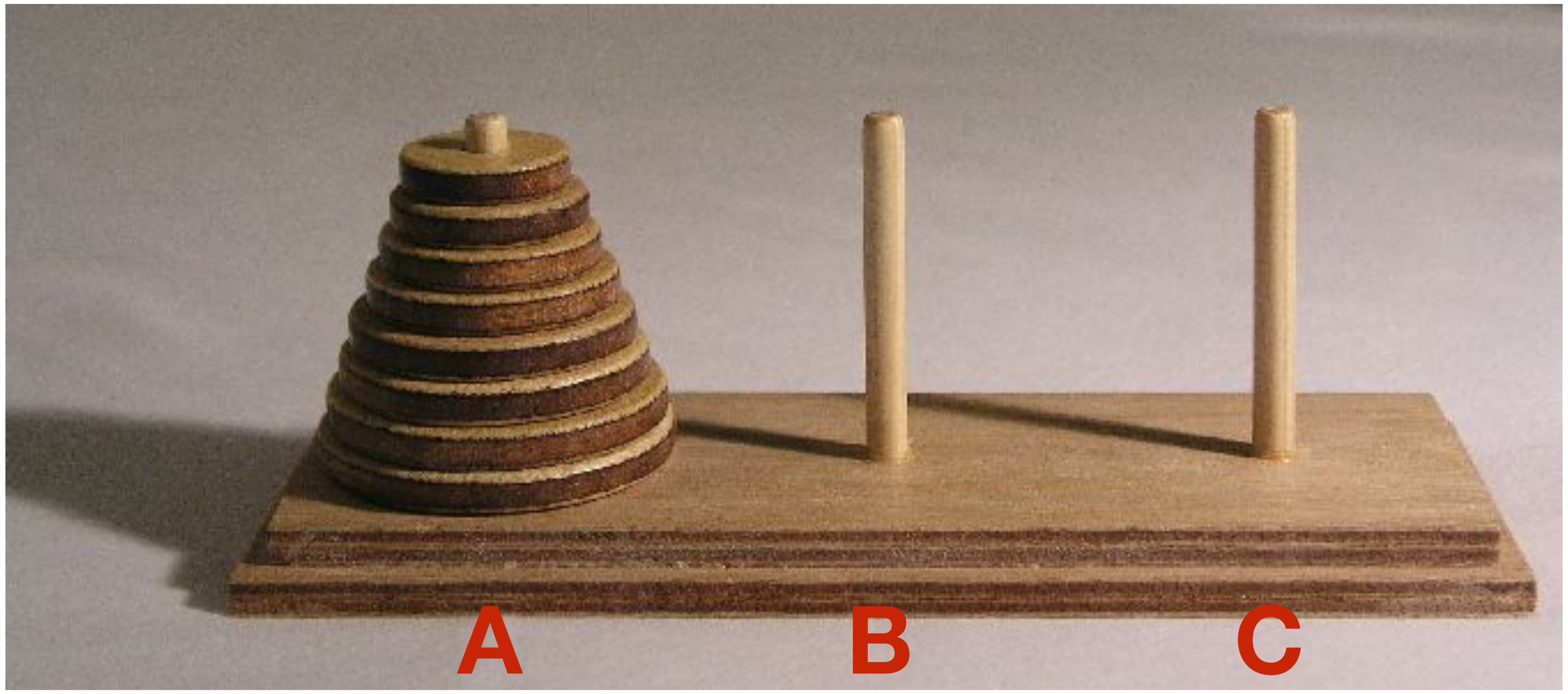
(c) the return value is 22*(22+2)/4 = 132.

(d) the return value is 22*(22+2)/2 = 264.

(e) it throws StackOverflowException.

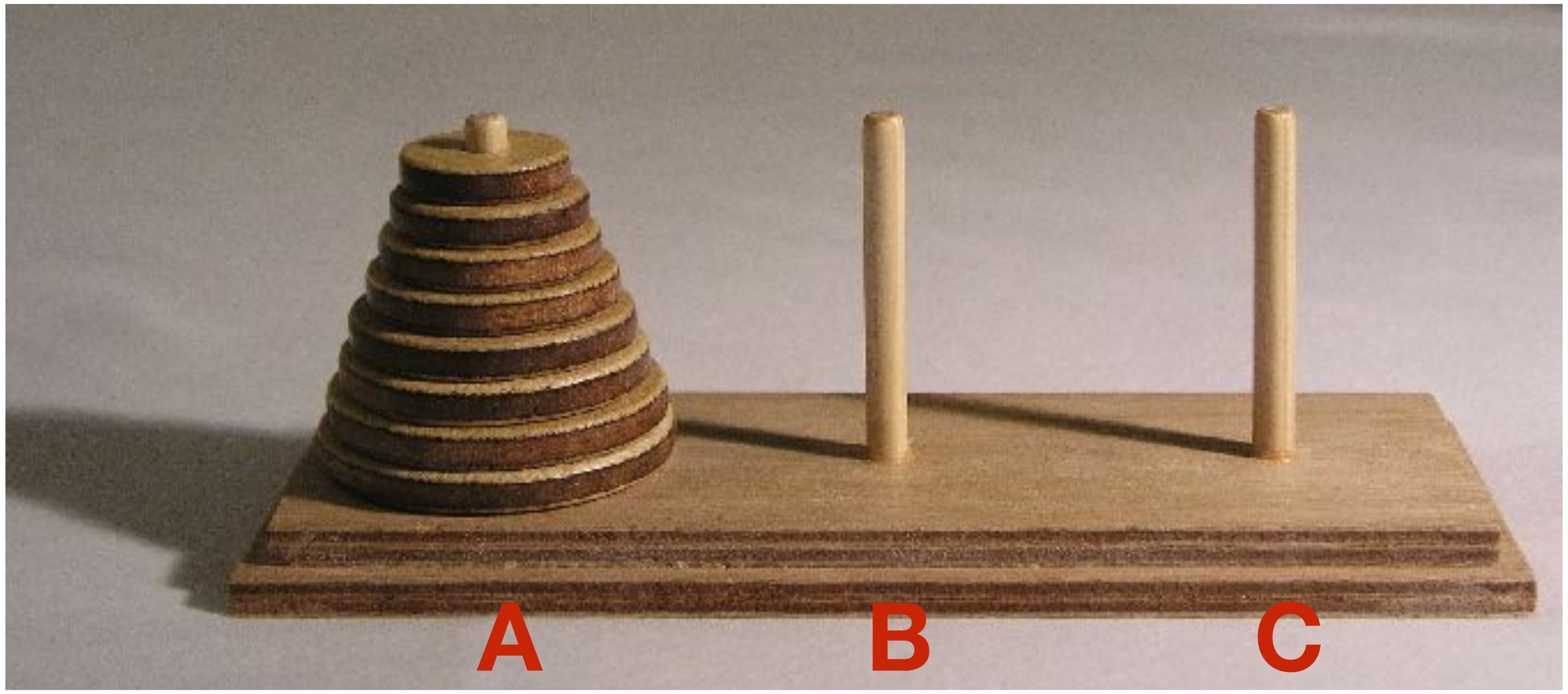
Towers of Hanoi

- A puzzle consisting of **three rods (A, B, C)**, and **n** disks, each at a different size and can slide onto any rod.



- Initially all disks are stacked onto rod A in ascending order of size (smallest at the top), forming a cone shape.

Towers of Hanoi



- **Objective**: move all disks from **A** to **C.**
- **Rules**:

Youtube demo

1. Only one disk can be moved at a time (from the **top of one stack to the top of another stack**)
2. No disk can be placed on top of a smaller disk (i.e. the disks on each stack **must be sorted at all times**)

Towers of Hanoi

- Play the game to gain some intuitions first. Also pay attention to the number of steps involved in each case.
  - n=1
  - n=2
  - n=3
  - n=4

Towers of Hanoi

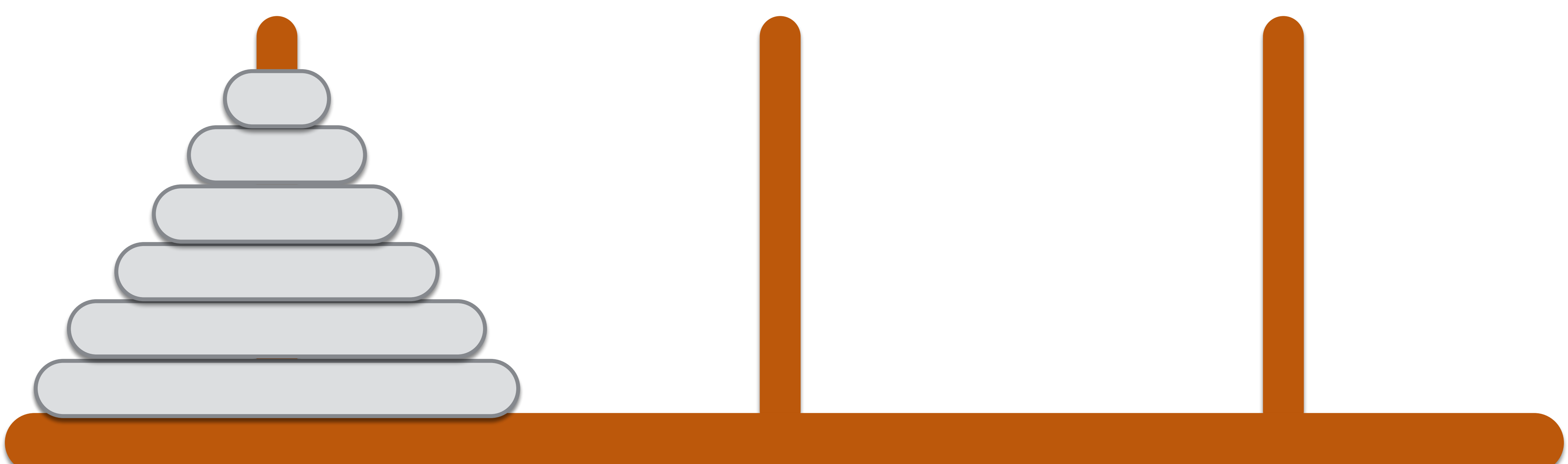
- Play the game to gain some intuitions first. Also pay attention to the number of steps involved in each case.
  - n=1: 1 step (A->C)
  - n=2: 3 steps (A->B, A->C, B->C)
  - n=3: 7 steps
  - n=4: what's your guess?

Towers of Hanoi

- Play the game to gain some intuitions first. Also pay attention to the number of steps involved in each case.
  - n=1: 1 step (A->C)
  - n=2: 3 steps (A->B, A->C, B->C)
  - n=3: 7 steps
  - n=4: 2^n-1
  - (A legend has it that the world will end as soon as some monks someplace finish the n = 64 version.)

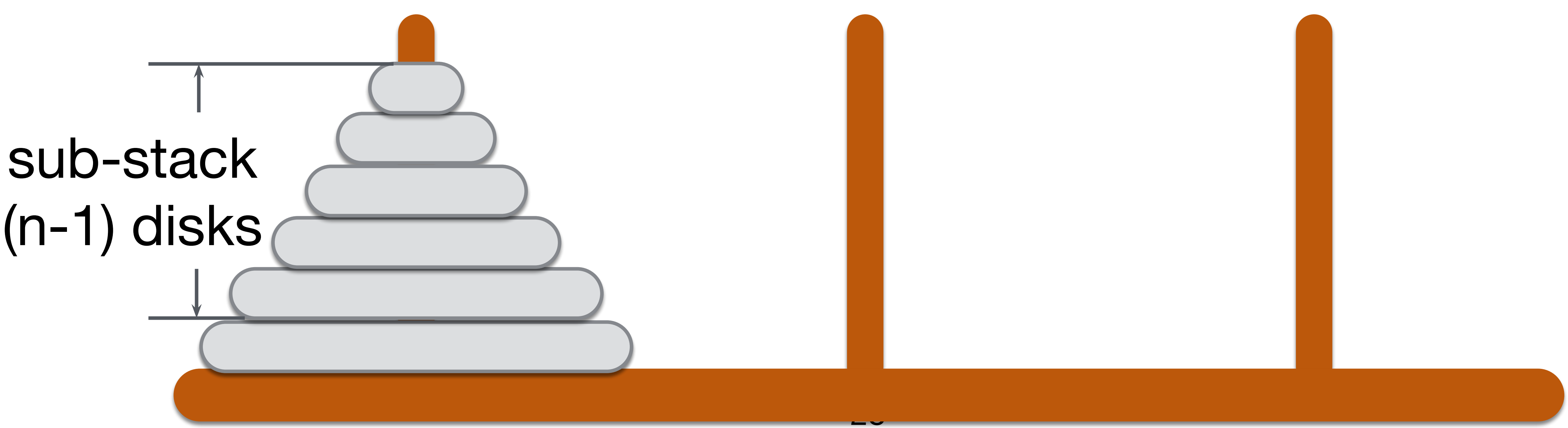
Towers of Hanoi

- Initially we have **n** disks stacked on column A. Let's call the top (**n-1**) disks a **sub-stack**.



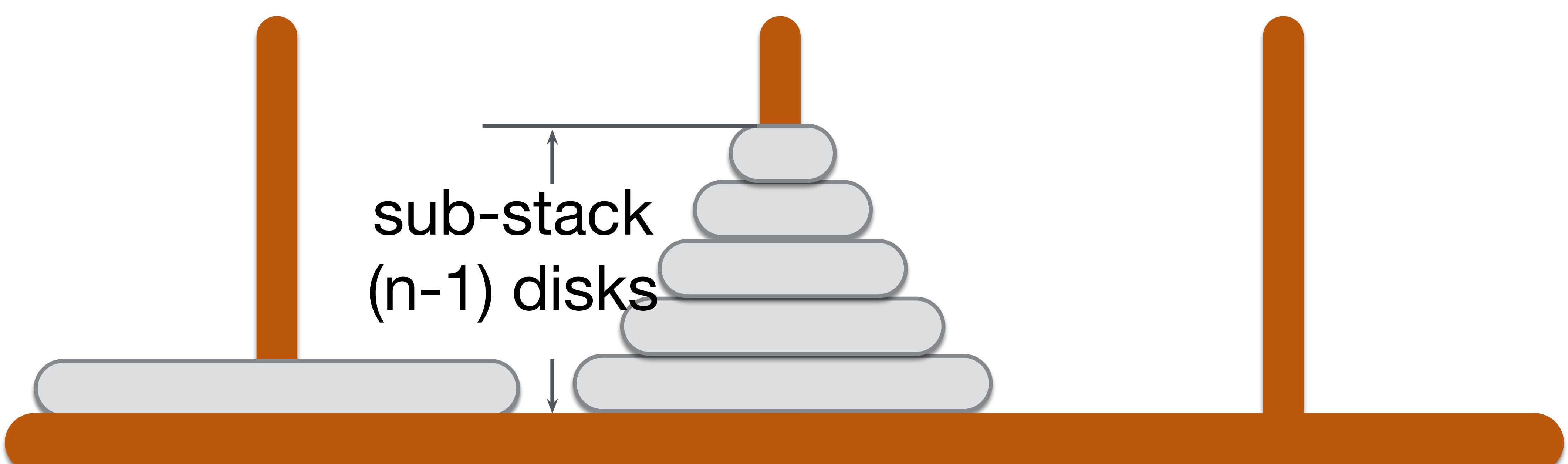
Towers of Hanoi

- Initially we have **n** disks stacked on column A. Let's call the top (**n-1**) disks a **sub-stack**.



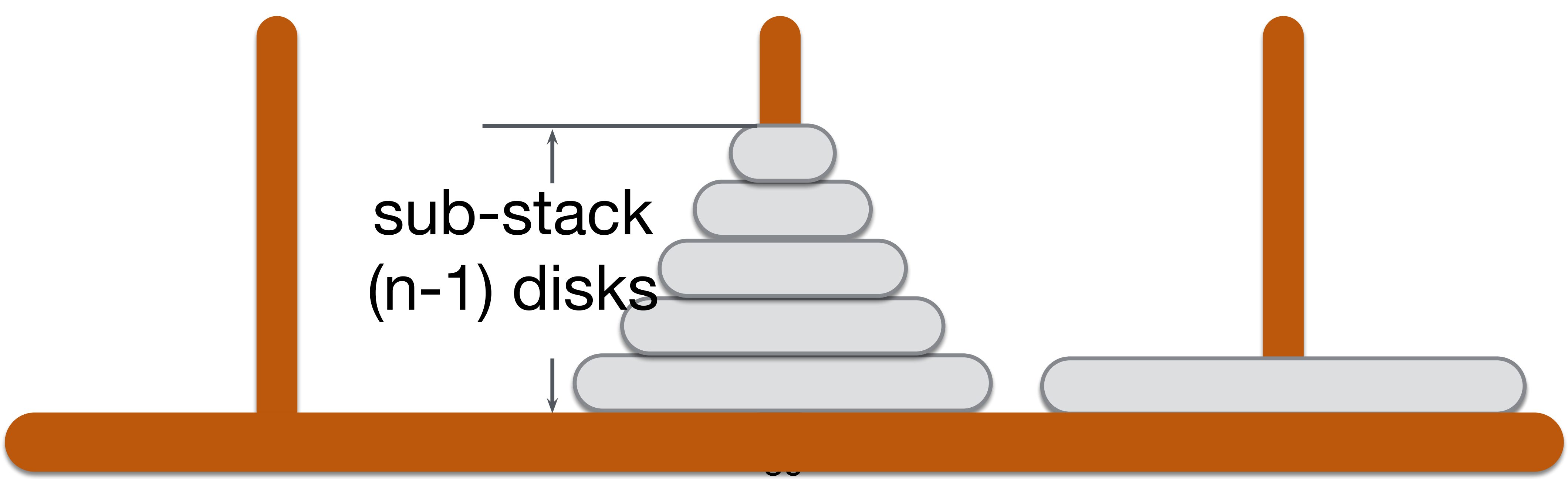
Towers of Hanoi

- Initially we have **n** disks stacked on column A. Let's call the top (**n-1**) disks a **sub-stack**.
- Assume you have some 'magical' way to move the sub-stack **from A to B via column C**.



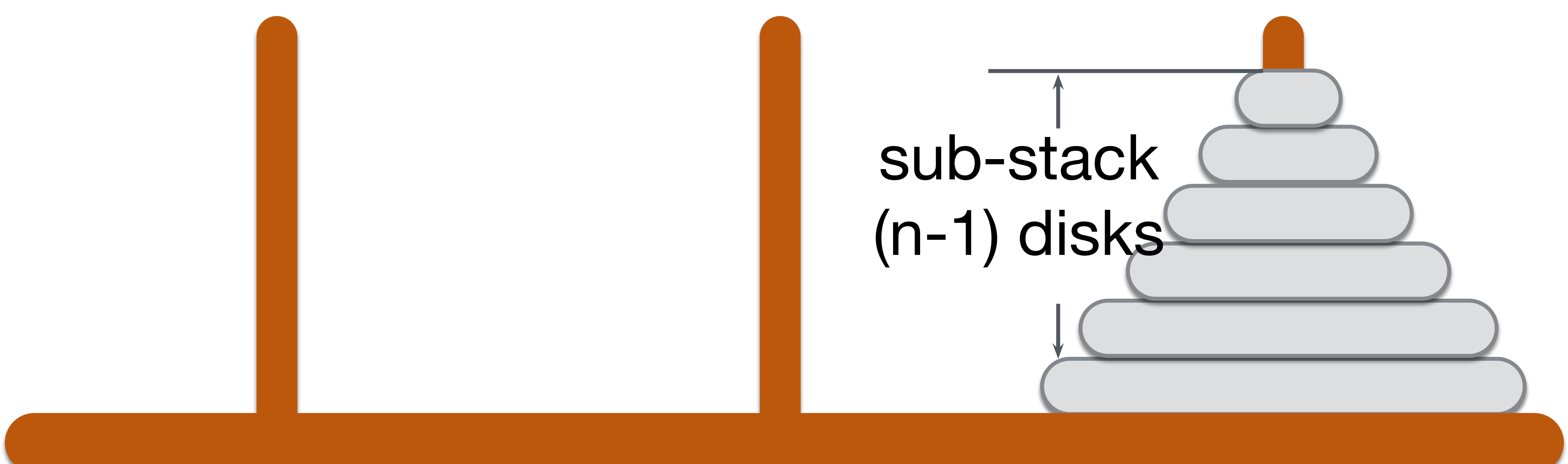
Towers of Hanoi

- Initially we have **n** disks stacked on column A. Let’s call the top (**n-1**) disks a **sub-stack**.
- Assume you have some ‘magical’ way to move the sub-stack **from A to B via column C**.
- Move the remaining one disk from A to C.



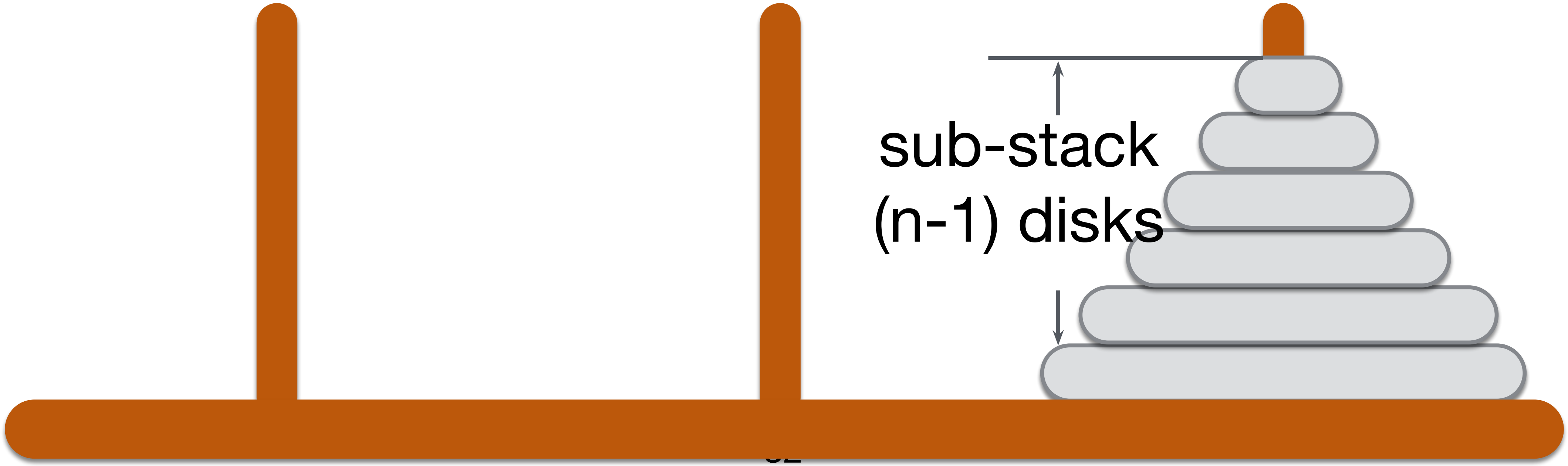
Towers of Hanoi

- Initially we have **n** disks stacked on column A. Let's call the top (**n-1**) disks a **sub-stack**.
- Assume you have some 'magical' way to move the sub-stack **from A to B via column C**.
- Move the remaining one disk from A to C.
- 'Magically' move the sub-stack **from B to C via A**.



Towers of Hanoi

- This is similar to the n=2 case, except the top disk there is now a sub-stack.
- So how do we ‘magically’ move the sub-stack? This is where recursion comes handy.
- Moving (n-1) disks from A to B via C is similar to the original problem, but with 1 less disk. We can recursively break it down to a (n-2) problem and so on.

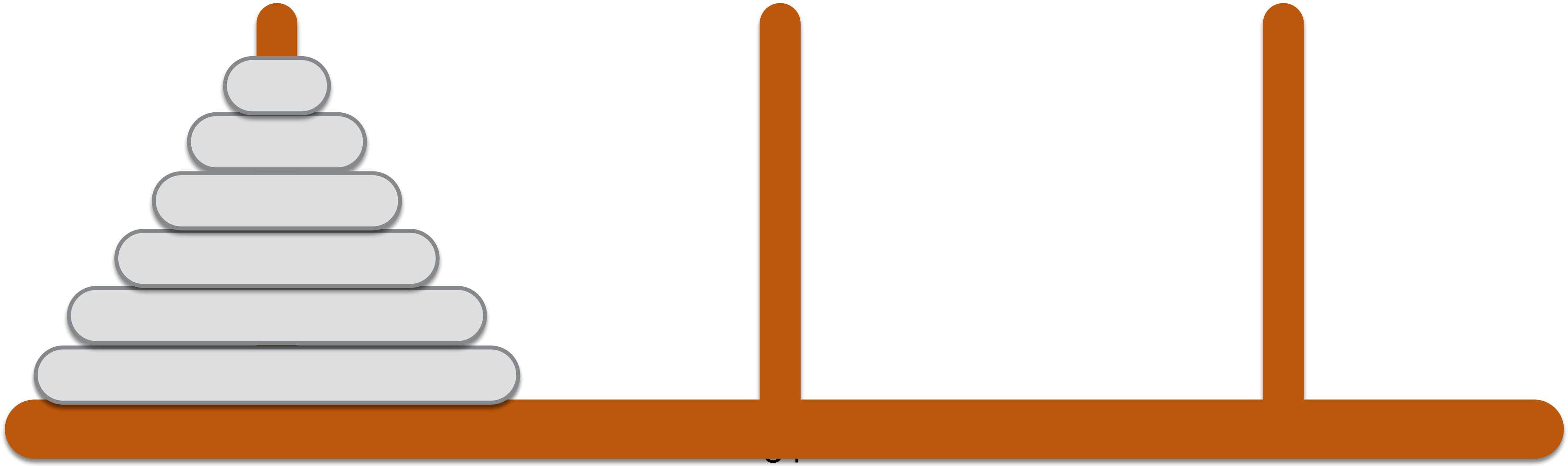


A Recursive Solution

- We can recursively solve the puzzle by defining the following procedure to move n rings from the Starting column to the Destination column, via the Via column. Conceptually:
    - move n-1 rings from Start to Via
    - move the n^{th} ring from Start to Dest
    - move n-1 rings from Via to Dest

Towers of Hanoi

- Say we want to move n disks from a **Start column S** to a **Dest column D**, using a **Via column V**.
- We use different terms than A, B, C, because during recursion, the associations of A, B, C to S, D, V will change.
- Initially:
  - **Start** is ‘A’, **Via** is ‘B’, **Dest** is ‘C’



Towers of Hanoi

To move the entire stack of **all disks** from **S** (via V) **to D**

1. Move the sub-stack (consisting of the top n-1 disks) from **S** (via D) **to V.**
2. Move the remaining (1) disk from **S to D**.
3. Move the sub-stack (consisting of the same n-1 disks as in step 1) from **V** (via S) **to D**.

Steps 1 and 3 (moving sub-stacks) involve recursion.

The base case is when the sub-stack has only 1 disk, in which case if can be trivially moved.

Recursive doTowers()

```
public void doTowers(int n, char start, char via, char dest)
{
  if (n > 0) {
    // move n-1 disks start --> via
    doTowers(n-1, start, dest, via);
    // move the nth disk start --> dest
    System.out.println("Move disk from "+start+" to "+dest);
    // move n-1 disks via --> to
    doTowers(n-1, via, start, dest);
  }
}

public static void main(String[] args) {
  doTowers(10, ‘A’, ‘B’, ‘C’);
}
```

Questions

Example of **doTowers(2,A,B,C)**

```java
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=2, start='A', via='B', dest='C')

Stacks of doTowers()

```java
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start=’A’, via=’C’, dest=’B’)
(n=2, start=’A’, via=’B’, dest=’C’) -> **call doTowers(2-1, ‘A’, ‘C’, ‘B’) //1**

Stacks of doTowers()

```java
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=0, start='A', via='B', dest='C') -> **base case**
(n=1, start='A', via='C', dest='B') -> **call doTowers(1-1, 'A', 'B', 'C') //1**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'A', 'C', 'B') //1**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='A', via='C', dest='B') -> call doTowers(1-1, 'A', 'B', 'C') //1
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'A', 'C', 'B') //1**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='A', via='C', dest='B') -> **print "Move disk from A to B" //2**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'A', 'C', 'B') //1**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='A', via='C', dest='B') -> **call doTowers(1-1, 'C', 'A', 'B') //3**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'A', 'C', 'B') //1**

Stacks of doTowers()

```java
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=0, start='C', via='A', dest='B') -> **base case**
(n=1, start='A', via='C', dest='B') -> **call doTowers(1-1, 'C', 'A', 'B') //3**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'A', 'C', 'B') //1**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='A', via='C', dest='B') -> call doTowers(1-1, 'C', 'A', 'B') //3
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'A', 'C', 'B') //1**

Stacks of doTowers()

```java
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=2, start='A', via='B', dest='C') -> call doTowers(2-1, 'A', 'C', 'B') //1

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=2, start='A', via='B', dest='C') -> **print "Move disk from A to C" //2**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'B', 'A', 'C') //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start=’B’, via=’A’, dest=’C’)
(n=2, start=’A’, via=’B’, dest=’C’) -> **call doTowers(2-1, ‘B’, ‘A’, ‘C’) //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=0, start='B', via='C', dest='A') -> **base case**
(n=1, start='B', via='A', dest='C') -> **call doTowers(1-1, 'B', 'C', 'A') //1**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'B', 'A', 'C') //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='B', via='A', dest='C') -> call doTowers(1-1, 'B', 'C', 'A') //1

(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'B', 'A', 'C') //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start=’B’, via=’A’, dest=’C’) -> **print “Move disk from B to C” //2**
(n=2, start=’A’, via=’B’, dest=’C’) -> **call doTowers(2-1, ‘B’, ‘A’, ‘C’) //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='B', via='A', dest='C') -> **call doTowers(1-1, 'A', 'B', 'C') //3**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'B', 'A', 'C') //3**

Stacks of doTowers()

```java
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=0, start='A', via='B', dest='C') -> **base case**
(n=1, start='B', via='A', dest='C') -> **call doTowers(1-1, 'A', 'B', 'C') //3**
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'B', 'A', 'C') //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=1, start='B', via='A', dest='C') -> call doTowers(1-1, 'A', 'B', 'C') //3
(n=2, start='A', via='B', dest='C') -> **call doTowers(2-1, 'B', 'A', 'C') //3**

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

(n=2, start='A', via='B', dest='C') -> call doTowers(2-1, 'B', 'A', 'C') //3

Stacks of doTowers()

```
public void doTowers(int n, char start, char via, char dest) {
  if (n > 0) {
    doTowers(n-1, start, dest, via);//1
    System.out.println("Move disk from "+start+" to "+dest);//2
    doTowers(n-1, via, start, dest);//3
  }
}
```

Done. In sum, printouts are:

Move disk from A to B
Move disk from A to C
Move disk from B to C

Clicker Question #4

```
public void applepen(int x, int y) {
   if(x == 0 || y == 0)
     System.out.print("Pen ");
   Else {
     applepen(x, y-1);
     if ((x - y)%2==1) System.out.print("Apple ");
     else System.out.print("Pineapple ");
     applepen(x-1, y);}
}
```

What happens if we call `applepen(3, 1)`?

(a) Pen Pineapple Apple Pineapple Pen

(b) Pen Pineapple Pen Apple Pen Pineapple Pen

(c) Pen Pineapple Apple Pen

(d) Pen Apple Pen

(e) it throws StackOverflowException.