CMPUT 175
Introduction to Foundations of Computing

Recursion

Objectives

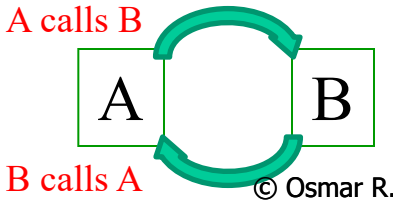
- Introduce the concept of recursion
- Understand how recursion works
- Learn how recursion can be used instead of repetition
- See some examples that use recursion

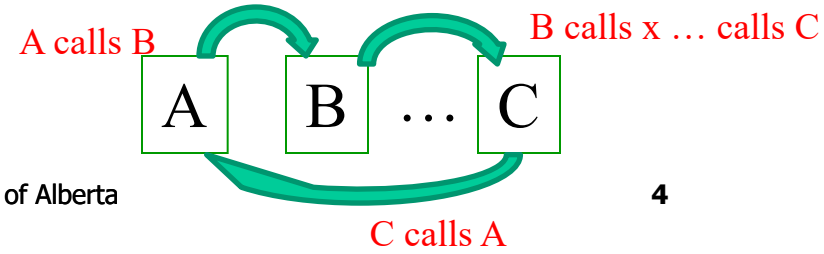
Outline of Lecture

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- Towers of Hanoi

Recursion

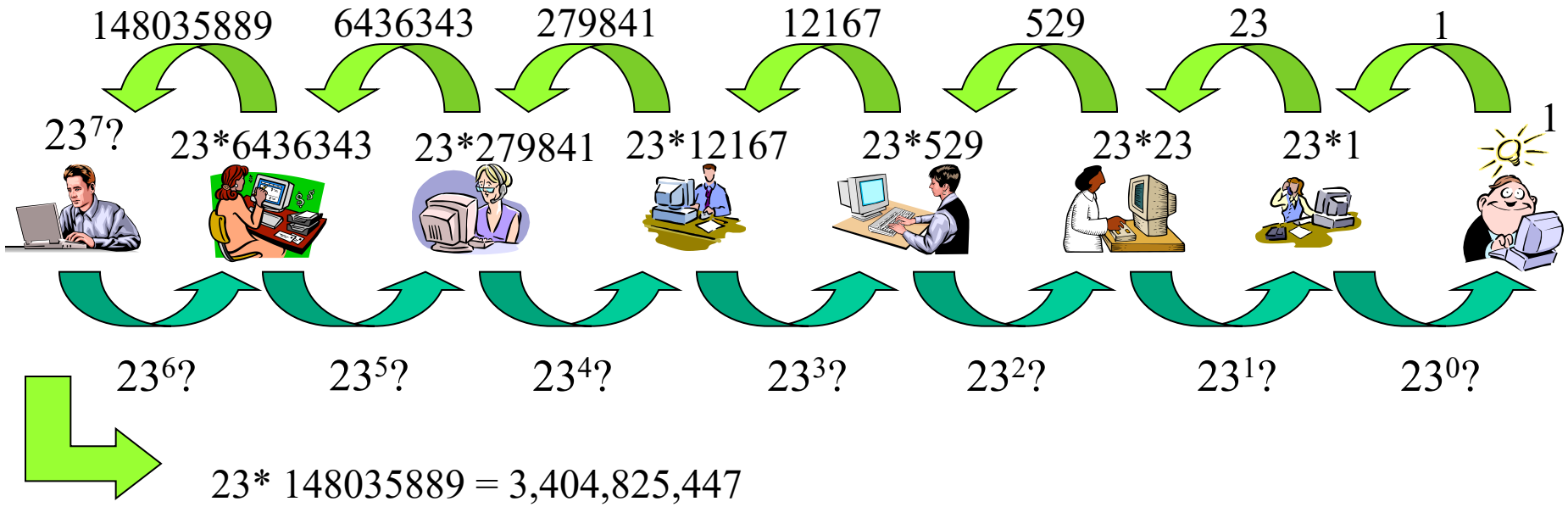
- **Recursion** occurs when a method calls itself, either directly or indirectly.
- If a problem can be resolved by solving a simple part of it a resolving the rest of the big problem the same way, we can write a method that solves the simple part of the problem then calls itself to resolve the rest of the problem.
- This is called a **recursive method**.





Recursive Method Example

- Suppose we want to calculate 23^7. We know that 23^7 is $23*23^6$. If we know the solution for 23^6 we would know the solution for 23^7.



$$
\begin{aligned}
23^{7} = \quad & 23 * 23^{6} = \\
& 23 * (23 * 23^{5}) = \\
& 23 * (23 * (23 * 23^{4})) = \\
& 23 * (23 * (23 * (23 * 23^{3}))) = \\
& 23 * (23 * (23 * (23 * (23 * 23^{2})))) = \\
& 23 * (23 * (23 * (23 * (23 * (23 * 23^{1}))))) = \\
& 23 * (23 * (23 * (23 * (23 * (23 * (23 * 23^{0})))))) = \\
& 23 * (23 * (23 * (23 * (23 * (23 * (23 * 1)))))) = \\
& 23 * (23 * (23 * (23 * (23 * (23 * (23)))))) = \\
& 23 * (23 * (23 * (23 * (23 * (529))))) = \\
& 23 * (23 * (23 * (23 * (12{,}167)))) = \\
& 23 * (23 * (23 * (279{,}841))) = \\
& 23 * (23 * (6{,}436{,}343)) = \\
& 23 * (148{,}035{,}889) = \\
& 3{,}404{,}825{,}447
\end{aligned}
$$

Outline of Lecture

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- Towers of Hanoi

Recursive Methods

- For recursion to **terminate**, two conditions must be met:
  - there must be one or more simple cases that do not make recursive calls. (base case)
  - the recursive call must somehow be simpler than the original call. (change the state to move towards the base case)

Factorial

- For example, we would like to write a recursive method that computes the factorial of an Integer:

  0! = 1

  1! = 1

  2! = 2*1 = 2 ➔ 2! = 2*1!

  3! = 3*2*1 = 6 ➔ 3! = 3*2!

  n! = n*(n-1) * … *3*2* 1 ➔ n! = n*(n-1)!

- The last observation, together with the simple cases is the basis for a recursive method.

Factorial - Non-recursive version

```
def factorial(anInt):
#factorial function using iteration instead of recursion
        productSoFar = 1
        for index in range(anInt,0,-1):
                productSoFar = index * productSoFar
        return productSoFar
```

Recursive Factorial Function –Example 1

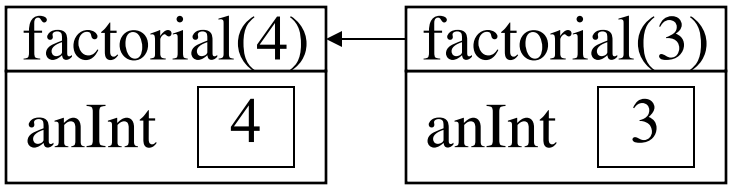
```
def factorial(anInt) :
# Return the factorial of the given int.

        if (anInt <= 1):
                return 1
        else:
                return anInt * factorial(anInt – 1)
```

Calling factorial(4)

factorial(4)	
anInt	4

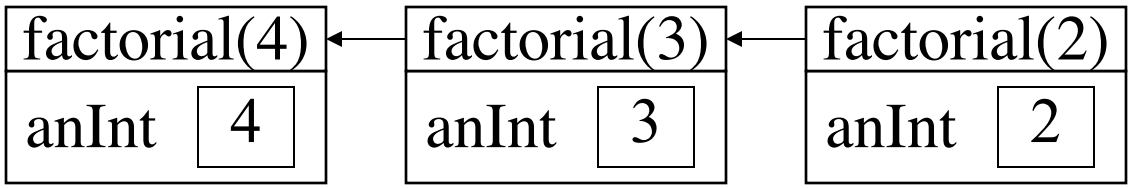
```
if (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Calling factorial(3)



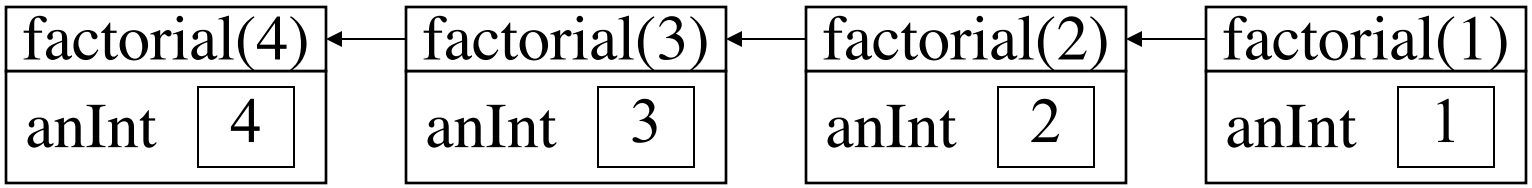

```
if (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Calling factorial(2)



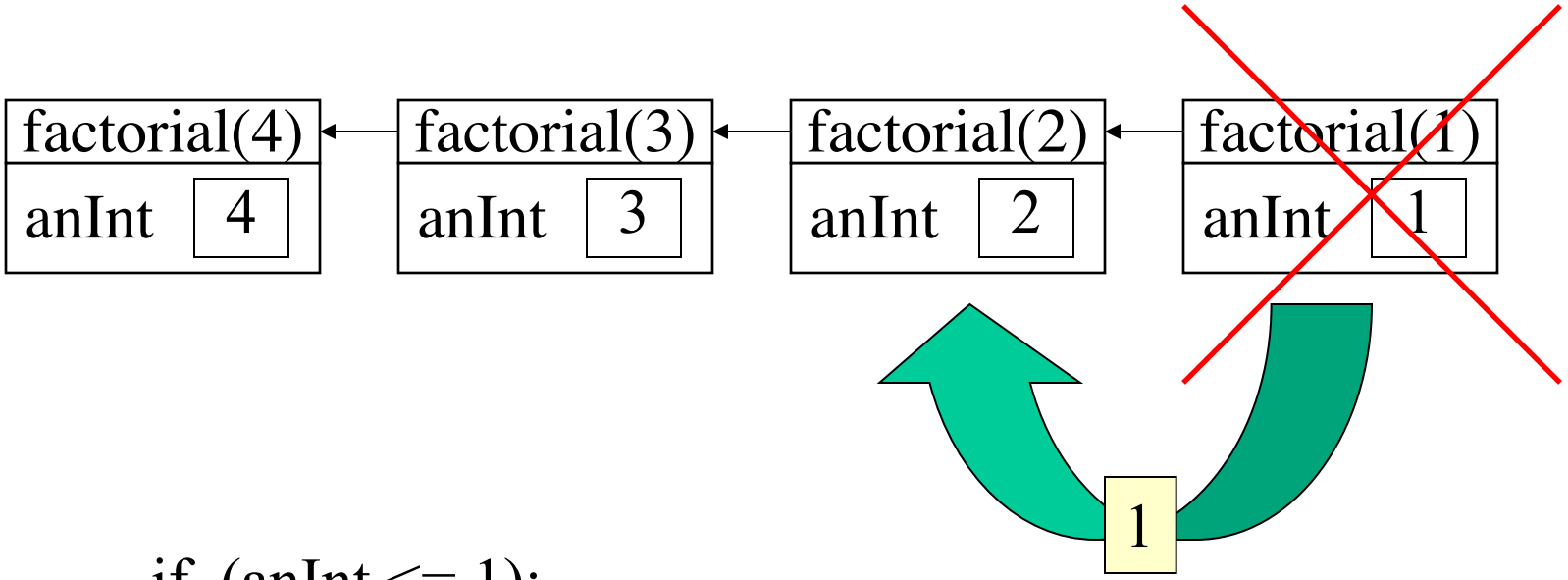

```
if (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Calling factorial(1)



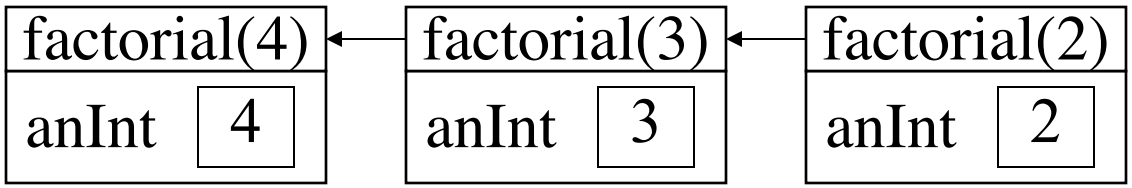

```
if (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Exiting factorial(1)




```
if  (anInt <= 1):
        return 1
else:
        return anInt * factorial (anInt – 1)
```

Exiting factorial(2)

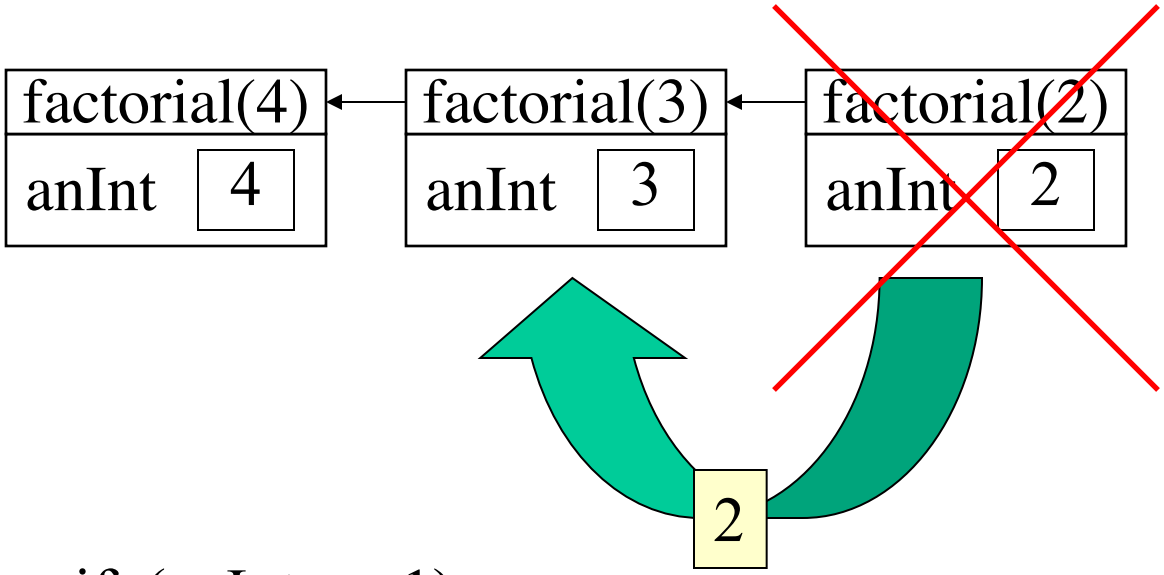



```
if  (anInt <= 1):
        return 1
else:
        return anInt *  factorial(anInt – 1)
```

Now evaluates to 1

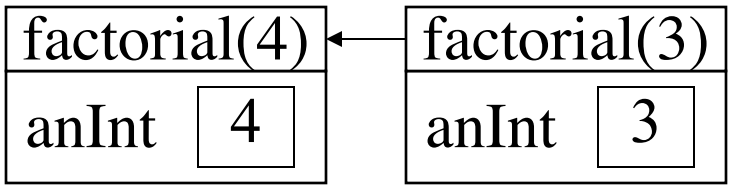
Now evaluates to 2 * 1

Exiting factorial(2)




```
if  (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Exiting factorial(3)

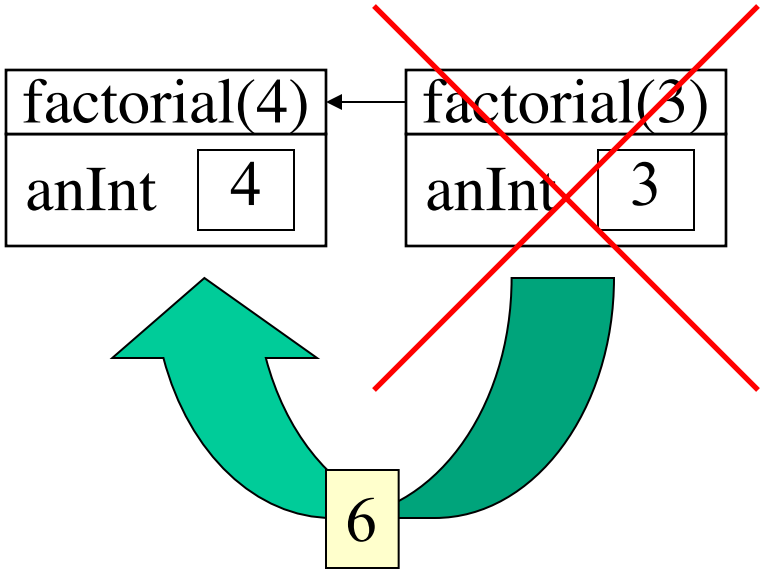



```
if  (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Now evaluates to 2

Now evaluates to 3 * 2

Exiting factorial(3)




```
if  (anInt <= 1):
        return 1
else:
        return anInt *   factorial(anInt – 1)
```

Exiting factorial(4)

factorial(4)	
anInt	4

So 24 is returned to the code which made the original call:

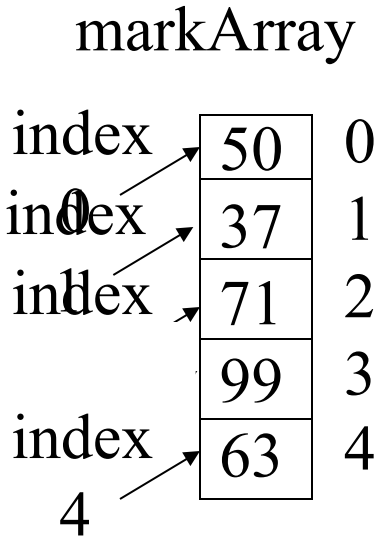
```
if  (anInt <= 1):
        return 1
else
        return anInt *   factorial(anInt – 1);
```

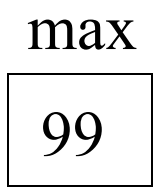
Now evaluates to 6

Now evaluates to 4 * 6

Find the Largest – Non-Recursive

```
// Find the largest element in an array of ints

markList = [50, 37, 71, 99, 63]
max = markList[0]
for index in range(1, len(markList)):
        if (markList[index] > max):
                max = markList[index]
print(max)
```

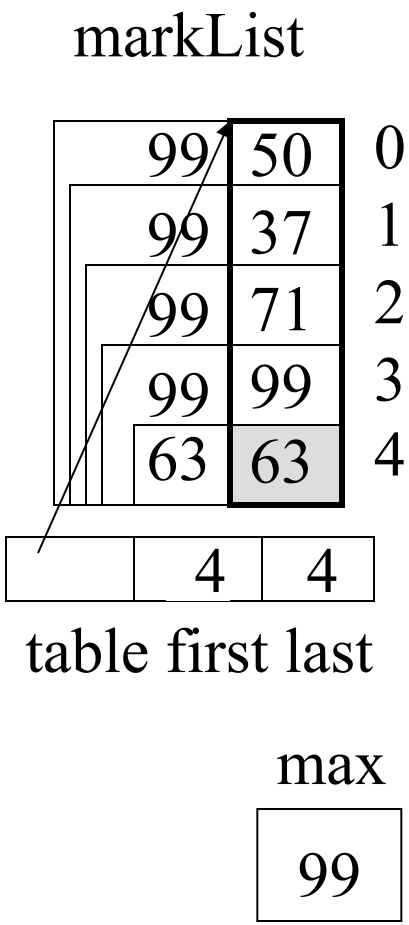




Find the Largest - Recursion

```
# Find the largest element in an array of ints
markList=[50, 37, 71, 99, 63]
max=largest(markList,0,len(markList)-1)
print(max)
```

```
def largest(table, first, last):
    if (first >= last):
        return table[last]
    else:
        myMax=largest(table,first+1,last)
        if (myMax > table[first]):
            return myMax
        else:
            return table[first]
```



Outline of Lecture

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- Towers of Hanoi

Direct References in Methods

- When a method is executing it can access some objects and some values.
- The receiver object can be referenced directly using the pseudo-variable **self**.
- Other objects and values can be referenced directly using method parameters and local variables.
- Still other objects and values can only be accessed indirectly by sending messages that return references to them.

Method Activations and Frames

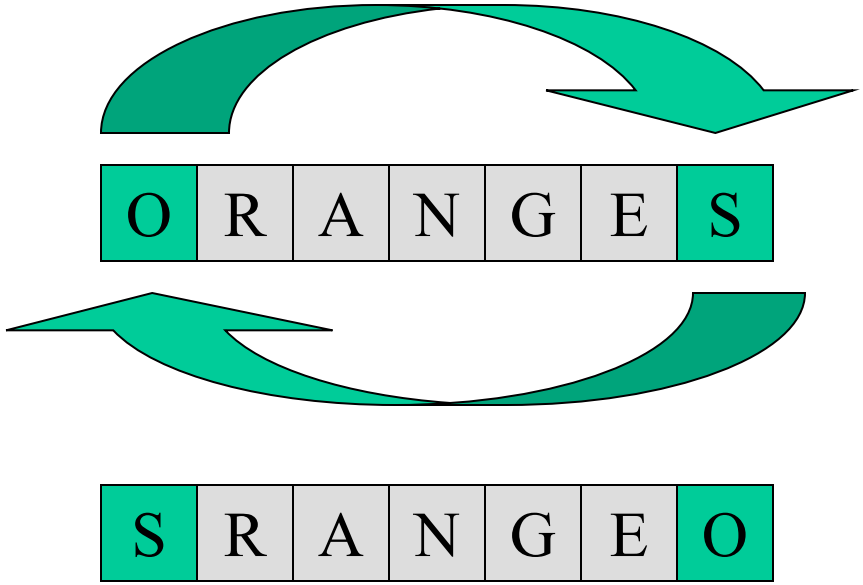
- A method can only access objects while it is executing or **active**.
- The collection of all direct references in a method is called the **frame** or **stack frame** of a method.
- The frame is created when the method is invoked, and destroyed when the method finishes.
- If a method is invoked again, a new frame is created for it with all its local variables.

Multiple Activations of a Method

- When we invoke a recursive method, the method becomes active.
- Before it is finished, it makes a recursive call to the same method.
- This means that when recursion is used, there is more than one copy of the same method active at once.
- Therefore, each active method has its own frame which contains independent copies of its direct references.
- These frames are stores in a stack: **stack frame**

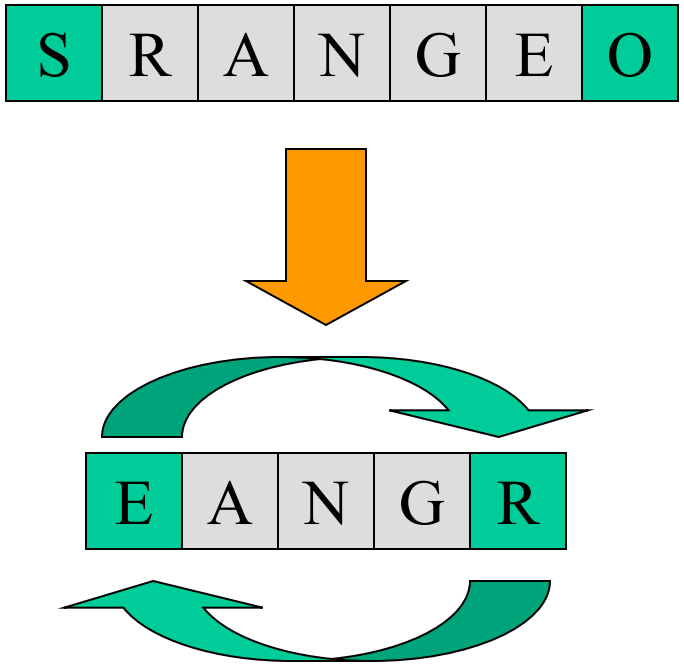
Reversing A String

- Want to reverse the characters of a string
  - Our basic operation is swap the two characters at the beginning and end of the string



Reversing A String

- Now we just need to do that with inner string, and attach two outer characters



- Stop when we get to 1 or 0 characters

Reversing A String

reverse("ORANGES")

S + reverse("RANGE") + O

E + reverse("ANG") + R

G + reverse("N") + A

Base Case! N

Reversing A String

reverse(“ORANGES”)

S + reverse(“RANGE”) + O

E + reverse(“ANG”) + R

G + N + A

Reversing A String

reverse(“ORANGES”)

S + reverse(“RANGE”) + O

E + reverse(“ANG”) + R

G N A

Reversing A String

reverse("ORANGES")

S + reverse("RANGE") + O

E + G N A + R

Reversing A String

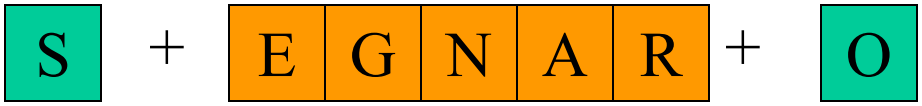
reverse(“ORANGES”)

S + reverse(“RANGE”) + O

E	G	N	A	R

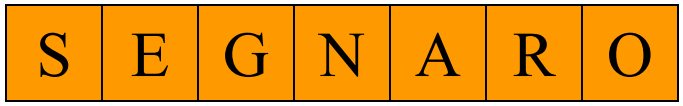
Reversing A String

reverse("ORANGES")



Reversing A String

reverse("ORANGES")

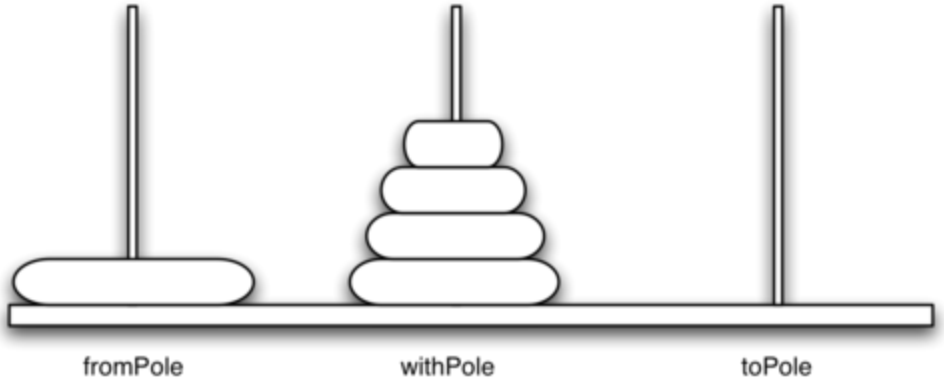


Complete the Python Program

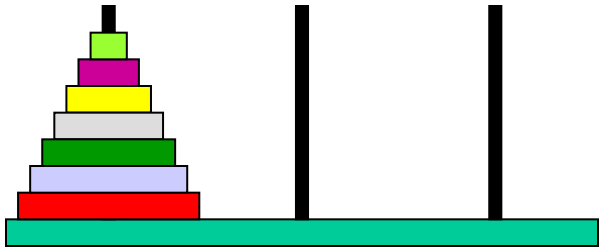
```
# Reverse a String
def reverse(word):
  if len(word) <2:
      return word
  else:
      return word[-1] + reverse(word[1:-1]) + word[0]

def main():
     word=input(" Enter a string :")
     print(reverse(word))
```

Outline of Lecture

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- Towers of Hanoi



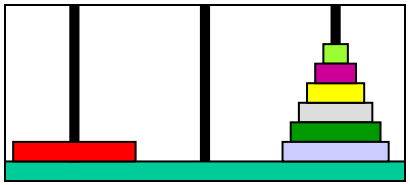
Towers of Hanoi



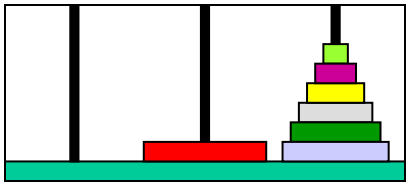
- No disk can be on top of a smaller disk;
- Only one disk is moved at a time;
- A disk must be placed on a tower;
- Only the top most disk can be moved.

To move n disks from tower 1 to 2:

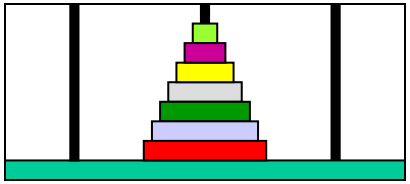
- Move n-1 disks from tower 1 to 3;



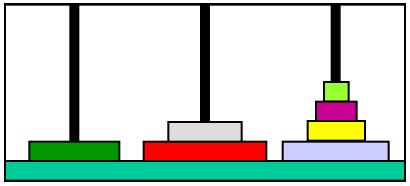
- Move 1 disk from tower 1 to 2;

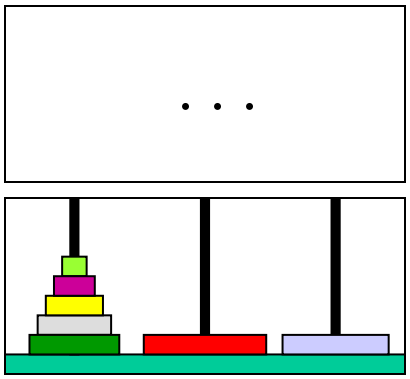


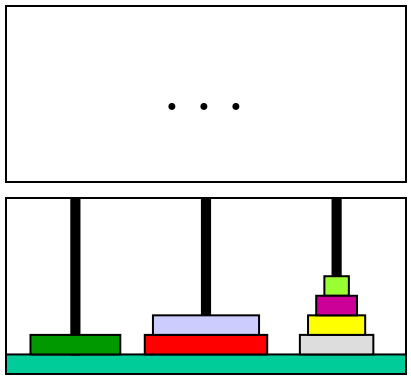
- Move n-1 disks from tower 3 to 2.

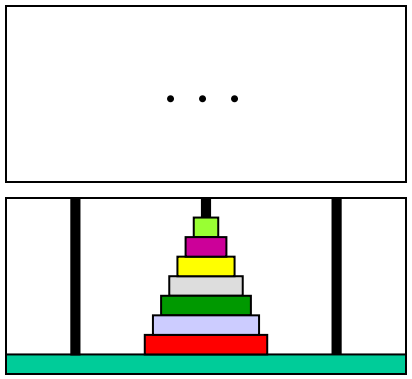


Towers of Hanoi 1

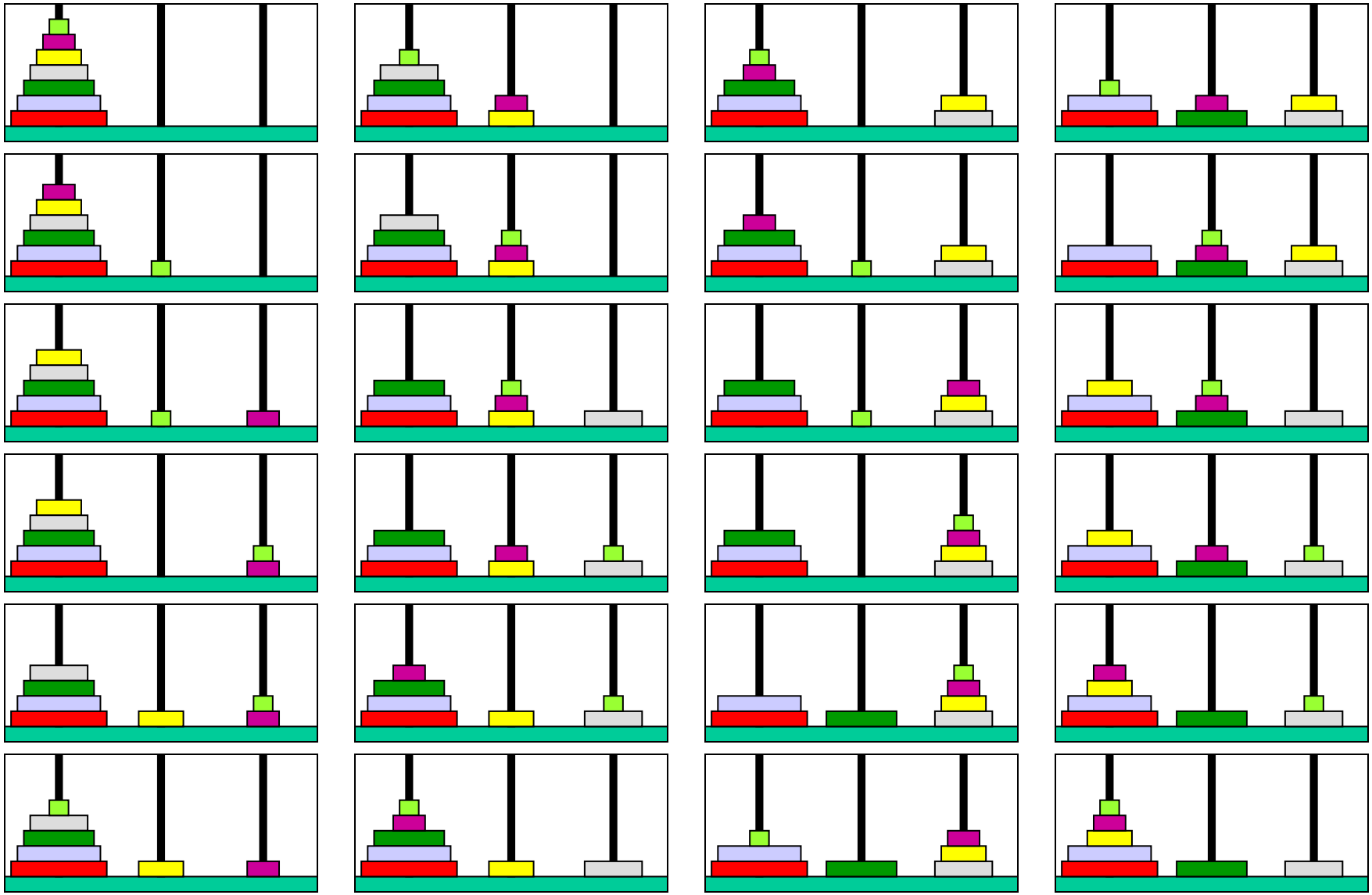




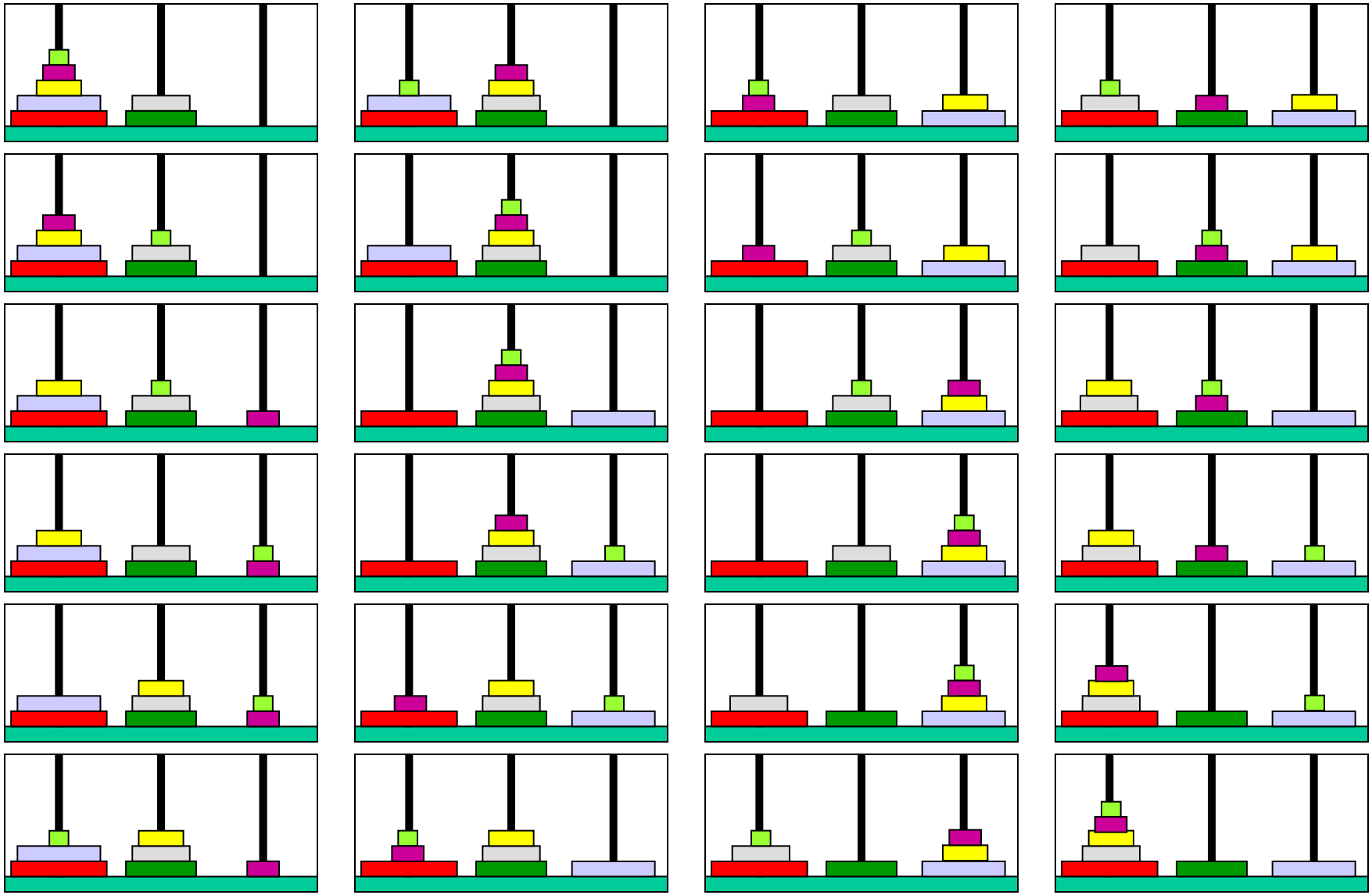




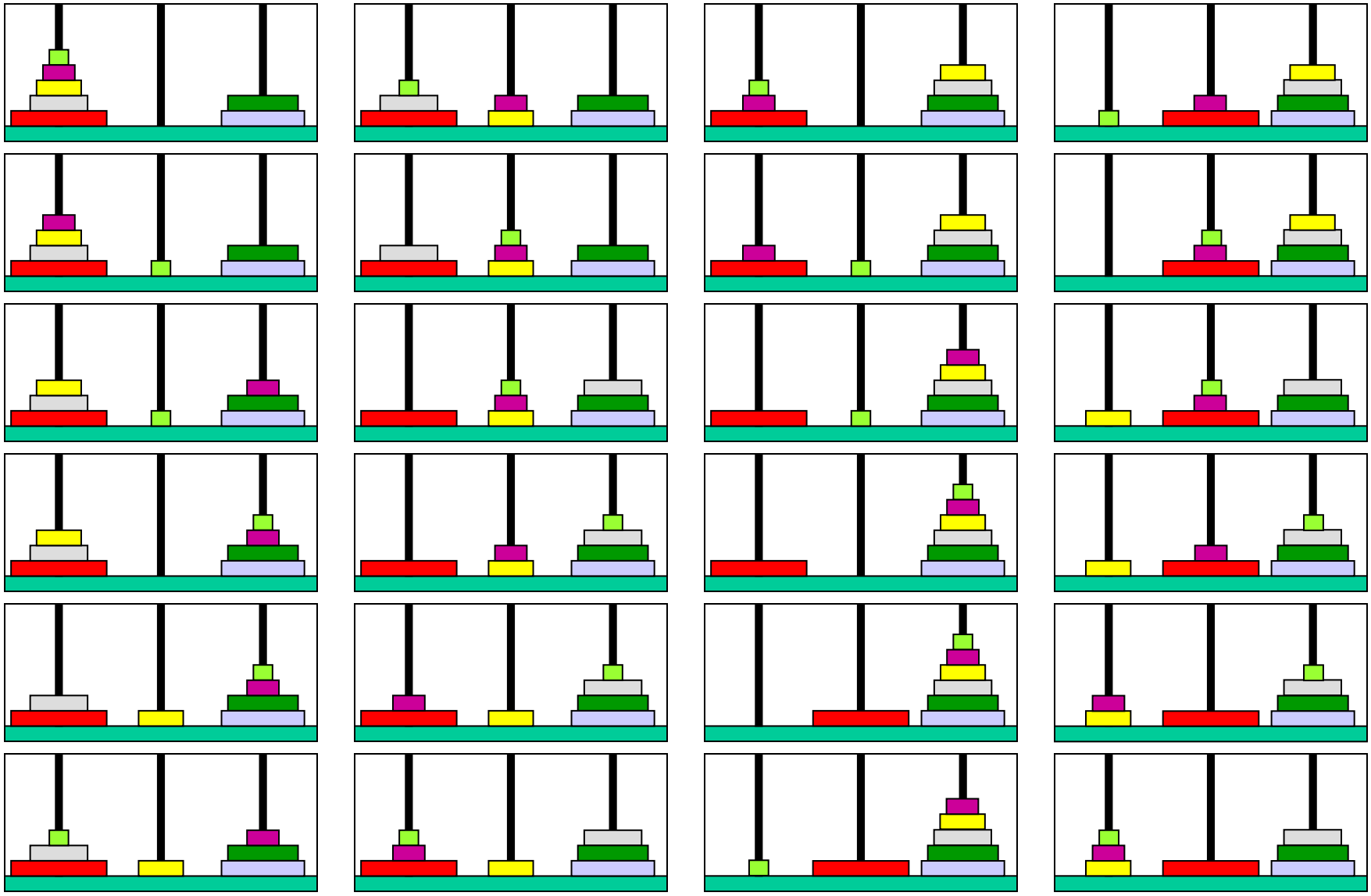
Towers of Hanoi 1



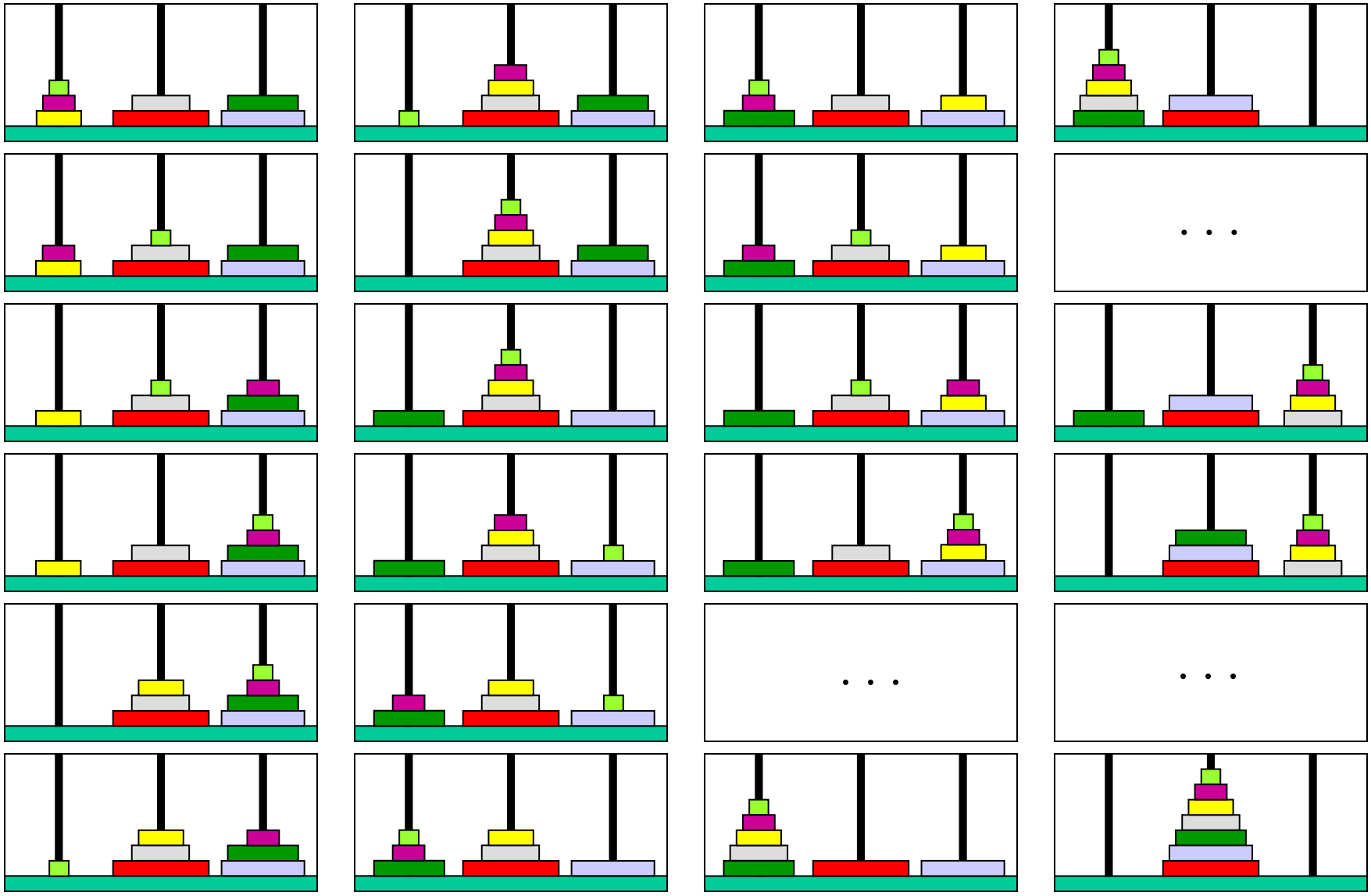
Towers of Hanoi 2



Towers of Hanoi 3



Towers of Hanoi 4



Example

```
def hanoi(n,fromPole,toPole,withPole):
    if n == 1:
       print('Move disk '+str(n)+' from '+fromPole+' to '+toPole)
    else:
       hanoi(n-1,fromPole,withPole,toPole)
       print('Move disk '+str(n)+' from '+fromPole+' to '+toPole)
       hanoi(n-1,withPole,toPole,fromPole)
def main():
    hanoi(3,'A','B','C')
main()
```

Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B

Example implementation in Python

```
A=[5,4,3,2,1]
B=[]
C=[]
def hanoi(height,fromPole, toPole, withPole):
    if height>=1:
        hanoi(height-1,fromPole,withPole,toPole)
        toPole.append(fromPole.pop())
        print (A,B,C)
        hanoi(height-1,withPole,toPole,fromPole)
hanoi(5,A,B,C)
```

```
[5, 4, 3, 2] [1] []
[5, 4, 3] [1] [2]
[5, 4, 3] [] [2, 1]
[5, 4] [3] [2, 1]
[5, 4, 1] [3] [2]
[5, 4, 1] [3, 2] []
[5, 4] [3, 2, 1] []
[5] [3, 2, 1] [4]
[5] [3, 2] [4, 1]
[5, 2] [3] [4, 1]
```

```
[5, 2, 1] [3] [4]
[5, 2, 1] [] [4, 3]
[5, 2] [1] [4, 3]
[5] [1] [4, 3, 2]
[5] [] [4, 3, 2, 1]
[] [5] [4, 3, 2, 1]
[1] [5] [4, 3, 2]
[1] [5, 2] [4, 3]
[] [5, 2, 1] [4, 3]
[3] [5, 2, 1] [4]
```

```
[3] [5, 2] [4, 1]
[3, 2] [5] [4, 1]
[3, 2, 1] [5] [4]
[3, 2, 1] [5, 4] []
[3, 2] [5, 4, 1] []
[3] [5, 4, 1] [2]
[3] [5, 4] [2, 1]
[] [5, 4, 3] [2, 1]
[1] [5, 4, 3] [2]
[1] [5, 4, 3, 2] []
[] [5, 4, 3, 2, 1] []
```