CMPUT 175 Introduction to Foundations of Computing

Recursion

Objectives

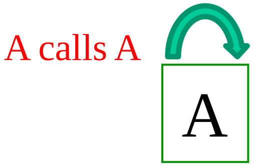
- Introduce the concept of recursion
- Understand how recursion works
- Learn how recursion can be used instead of repetition
- See some examples that use recursion

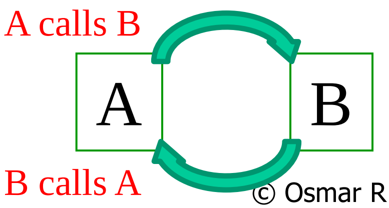
Outline of Lecture

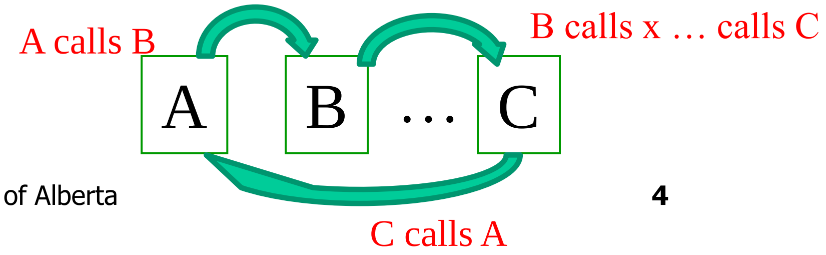
- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- Towers of Hanoi

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 and 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**.

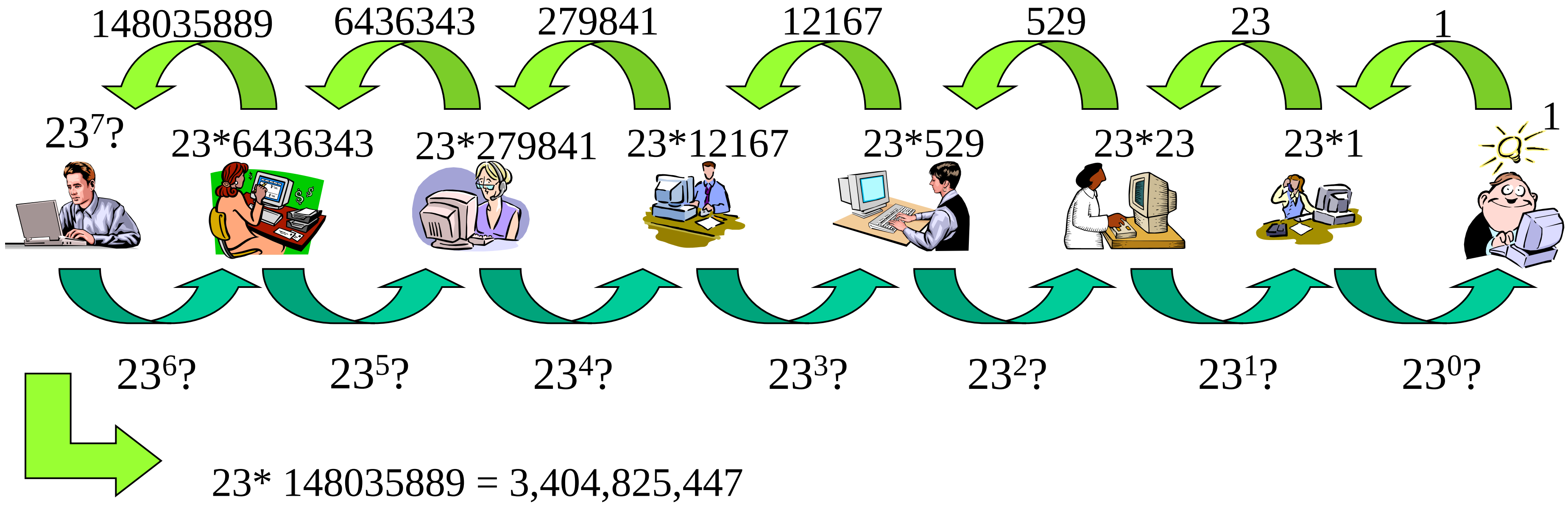






Recursive Method Example

- 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**.



148035889 6436343 279841 12167 529 23 1

23^7? 23*6436343 23*279841 23*12167 23*529 23*23 23*1 1

23^6? 23^5? 23^4? 23^3? 23^2? 23^1? 23^0?

23* 148035889 = 3,404,825,447

$23^{7} = \quad 23 * 23^{6} =$

$23 * (23 * 23^{5}) =$

$23 * (23 * (23 * 23^{4})) =$

$23 * (23*(23*(23 * 23^{3}))) =$

$23 * (23*(23*(23*(23*23^{2}))))=$

$23 * (23*(23*(23*(23*(23*23^{1})))))=$

$23 * (23 *(23*(23*(23*(23*(23*23^{0}))))))=$

$23 * (23 *(23*(23*(23*(23*(23*1))))))=$

$23 * (23 *(23*(23*(23*(23*(23))))))=$

$23 * (23 *(23*(23*(23*(529)))))=$

$23 * (23 *(23*(23*(12{,}167))))=$

$23 * (23 *(23*(279{,}841)))=$

$23 * (23 *(6{,}436{,}343))=$

$23 * (148{,}035{,}889)=$

$3{,}404{,}825{,}447$

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)

Outline of Lecture

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

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.

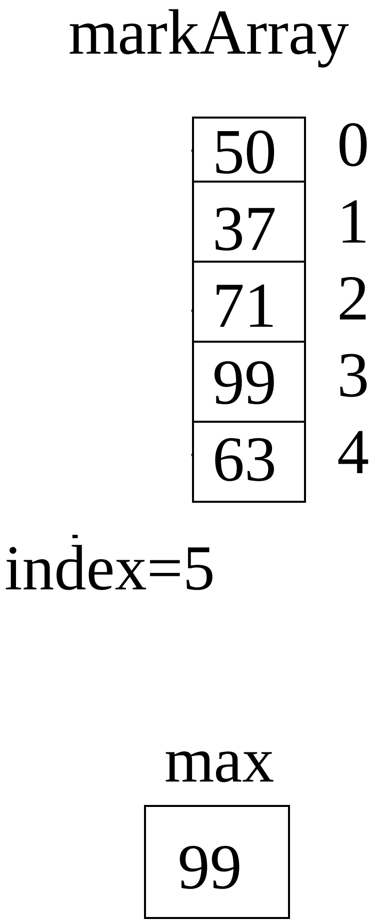
Factorial Method

$n! = n*(n-1)!$

```python
def factorial(number):
    # Return the factorial of number.

    if (number == 0 or number== 1):    #base case
        answer = 1
    else:
        answer = number * factorial(number-1)

    return answer
```

Loop Example

```
// 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)
```



markArray

50	0
37	1
71	2
99	3
63	4

index=5

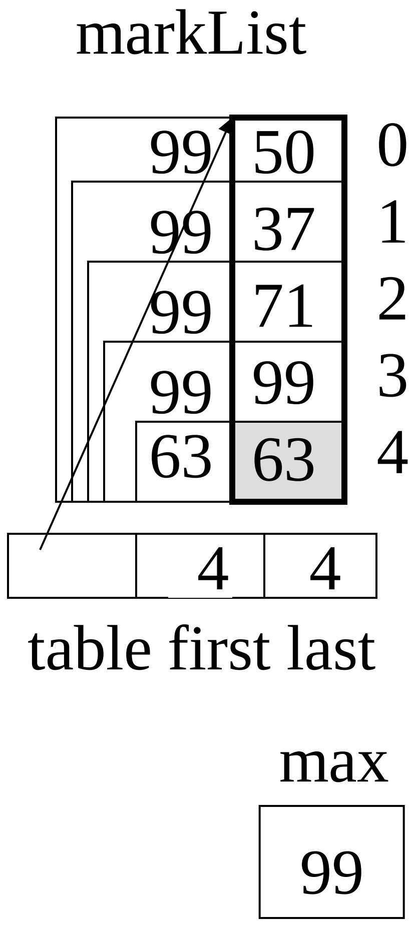
max

99

Recursion Example

```
# 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

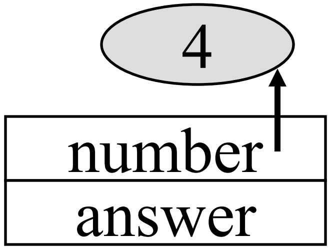
- When we invoke a recursive method, the method becomes active.
- Before it is finished, it makes a recursive call to the same method.
- This means that when recursion is used, there is more than one copy of the same method active at once.
- Therefore, each active method has its own frame which contains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

Factorial Method

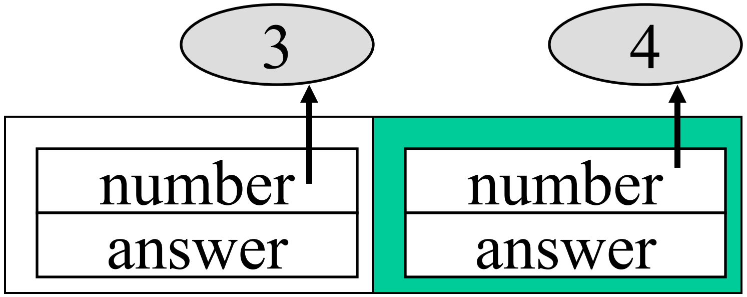
```
def factorial(number):
    # Return the factorial of number.

    if (number == 0 or number== 1):    #base case
        answer = 1
    else:
        answer = number * factorial(number-1)

    return answer
```

Calling factorial(4)

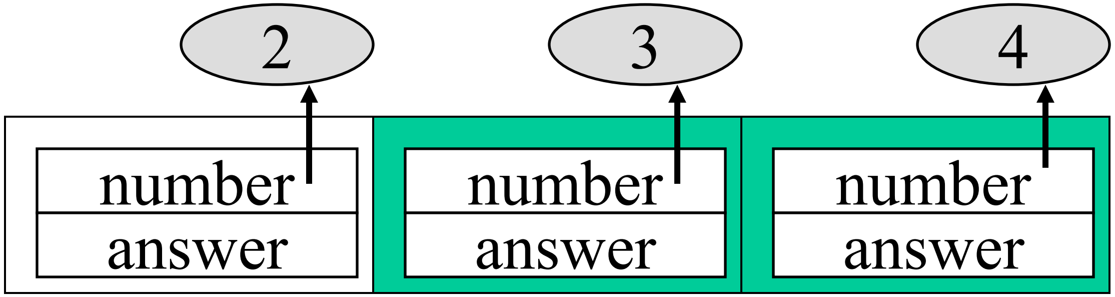
```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```



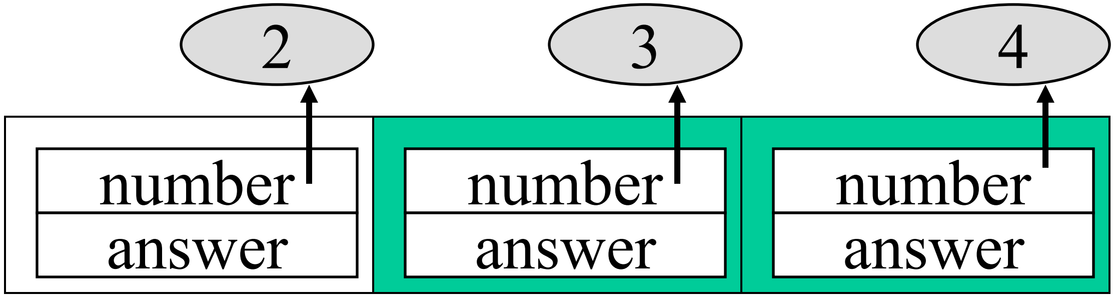
```
answer = number * factorial(number-1) # answer=4*factorial(3)
```



```
answer = number * factorial(number-1)  # answer=3*factorial(2)
```

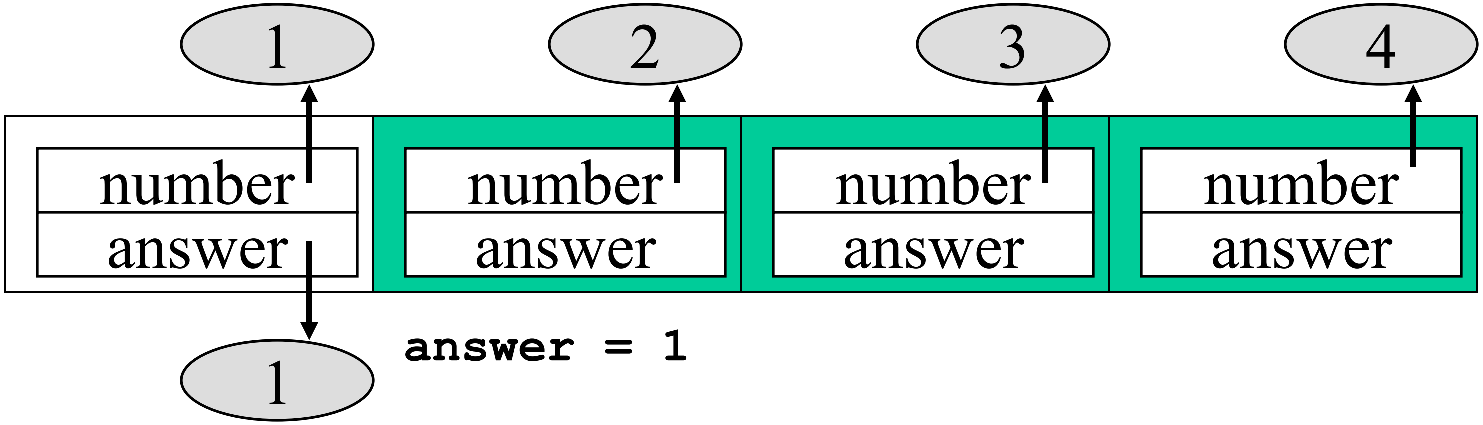


Calling factorial(4)



```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```

```
answer = number * factorial(number-1)  # answer=2*factorial(1)
```

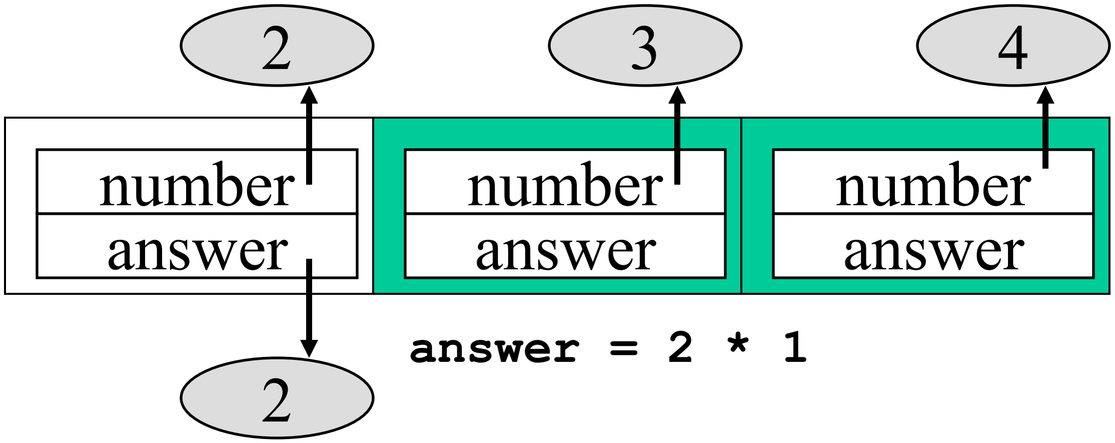


```
answer = 1
```

```
factorial(1) finishes and returns 1
```

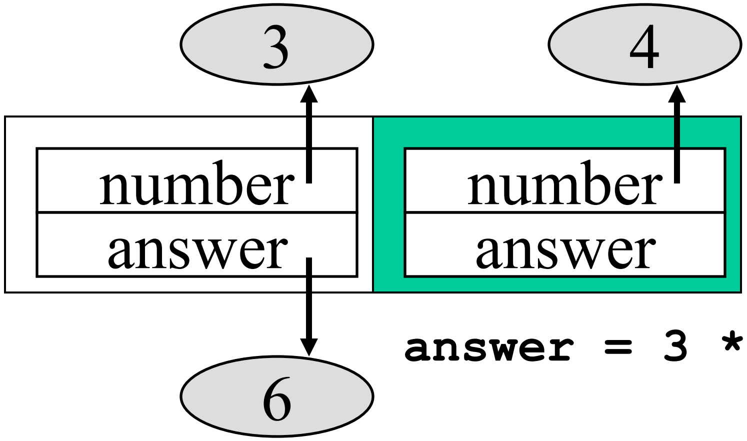
Calling factorial(4)

```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```



answer = 2 * 1

factorial(2) finishes and returns 2

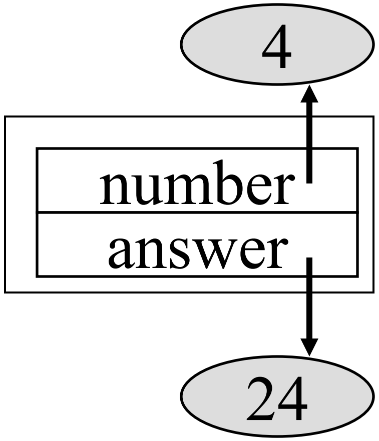


answer = 3 * 2

factorial(3) finishes and returns 6

Calling factorial(4)

```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```

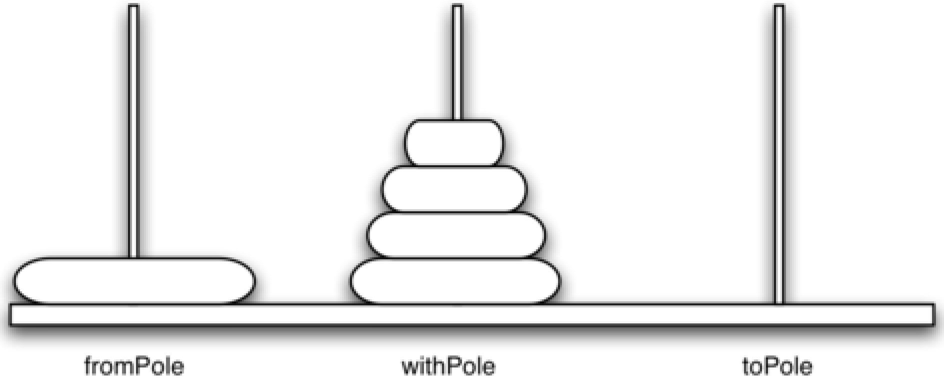


answer = 4 * 6

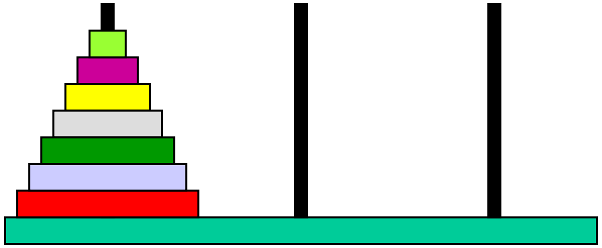
factorial(4) finishes and returns 24

Outline of Lecture

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



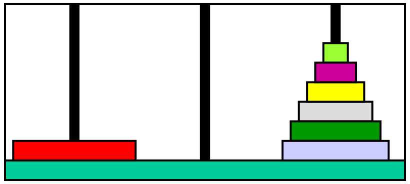
Towers of Hanoi



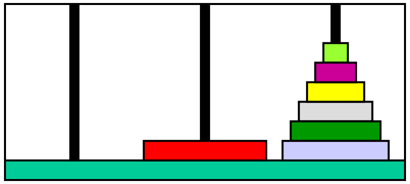
- No disk can be on top of a smaller disk;
- Only one disk is moved at a time;
- A disk must be placed on a tower;
- Only the top most disk can be moved.

To move n disks from tower 1 to 2:

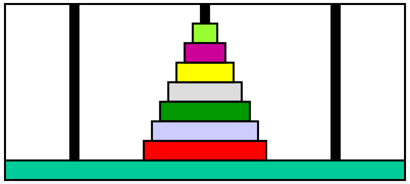
- Move n-1 disks from tower 1 to 3;



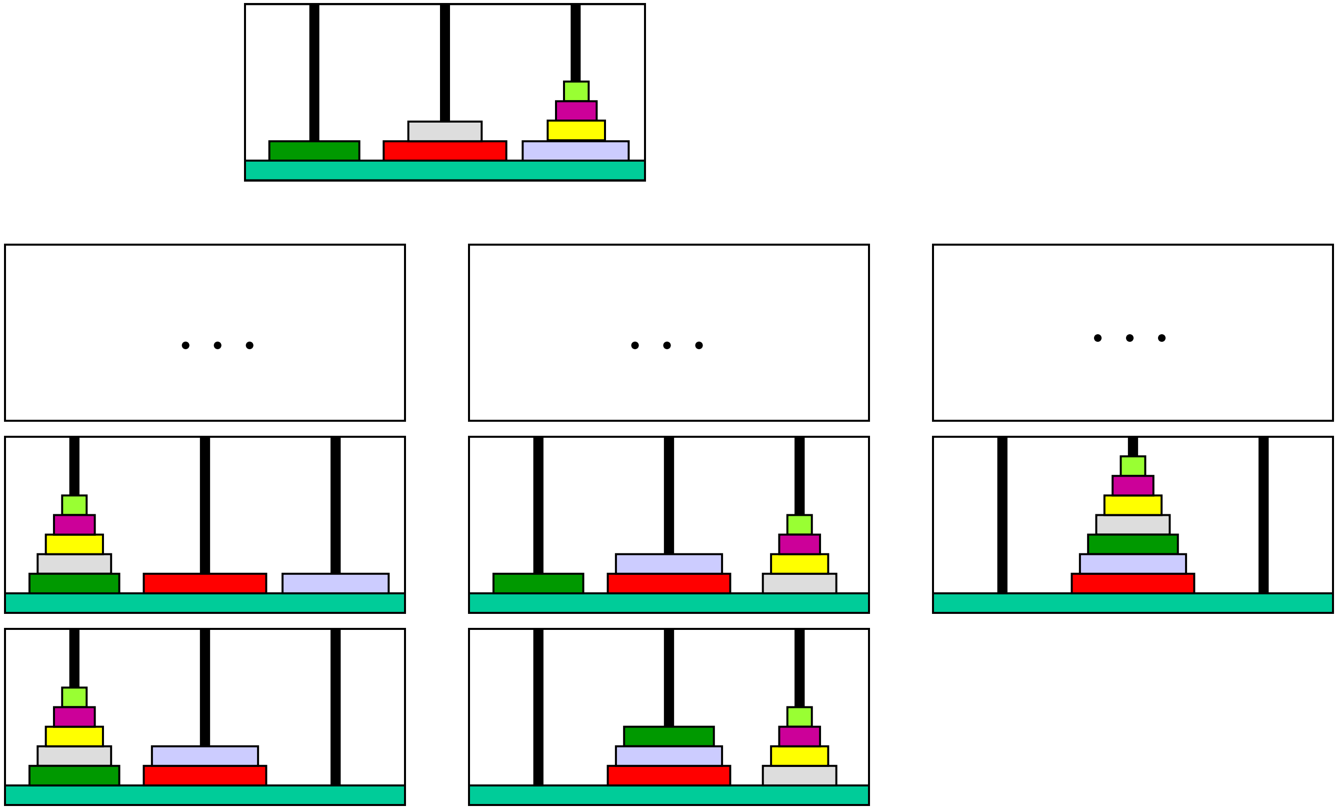
- Move 1 disk from tower 1 to 2;



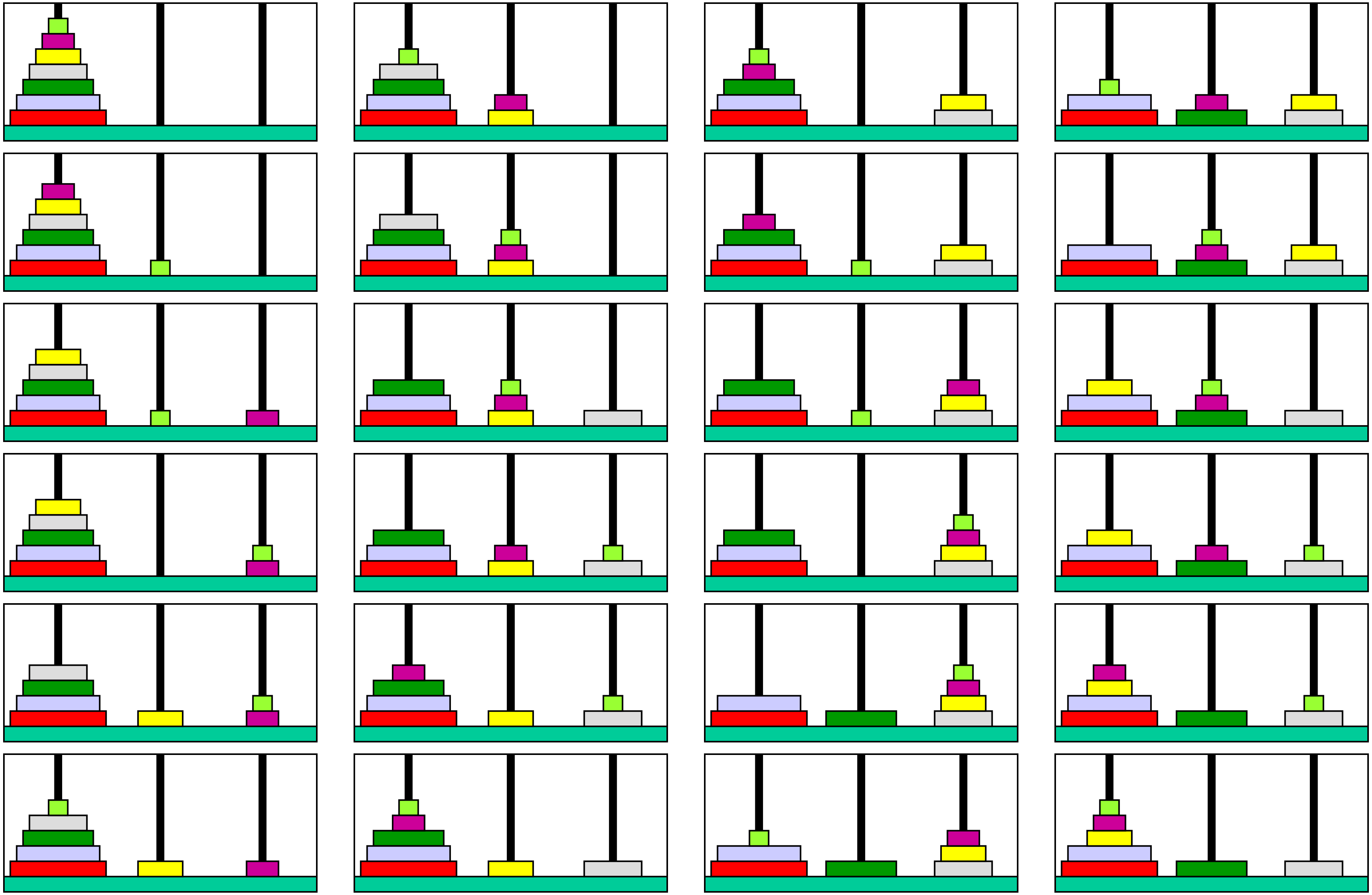
- Move n-1 disks from tower 3 to 2.



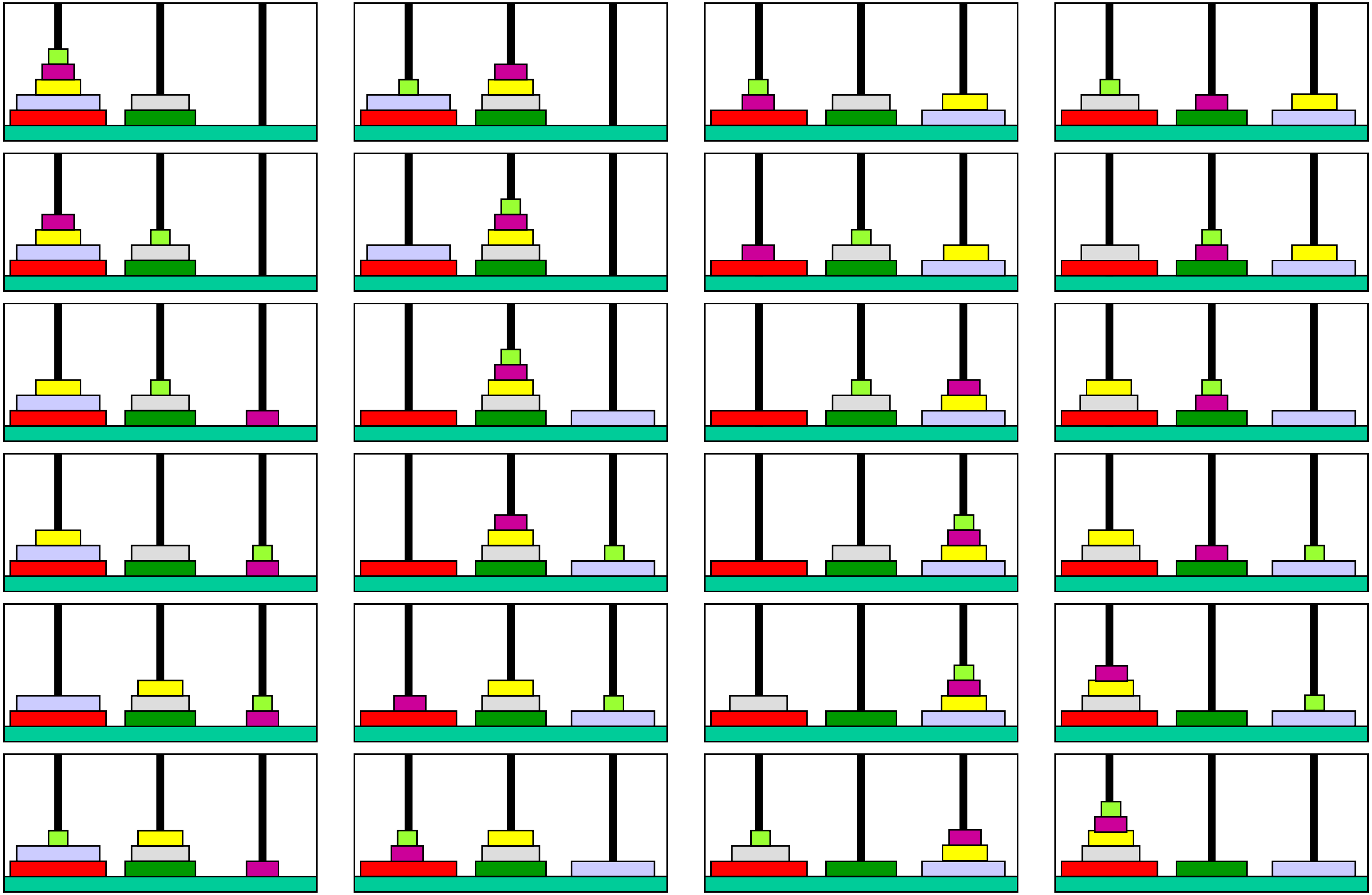
Towers of Hanoi 1



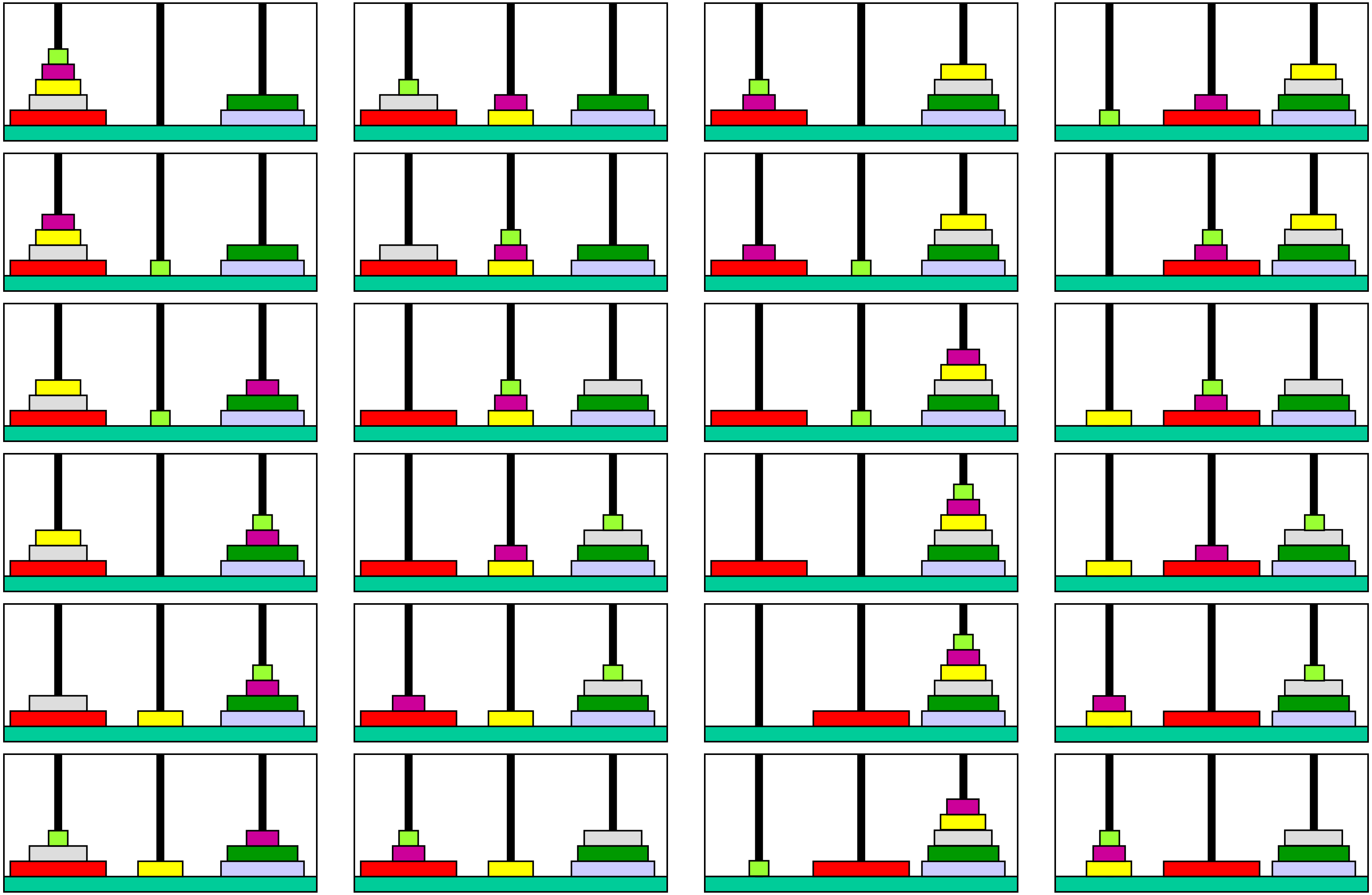
Towers of Hanoi 1



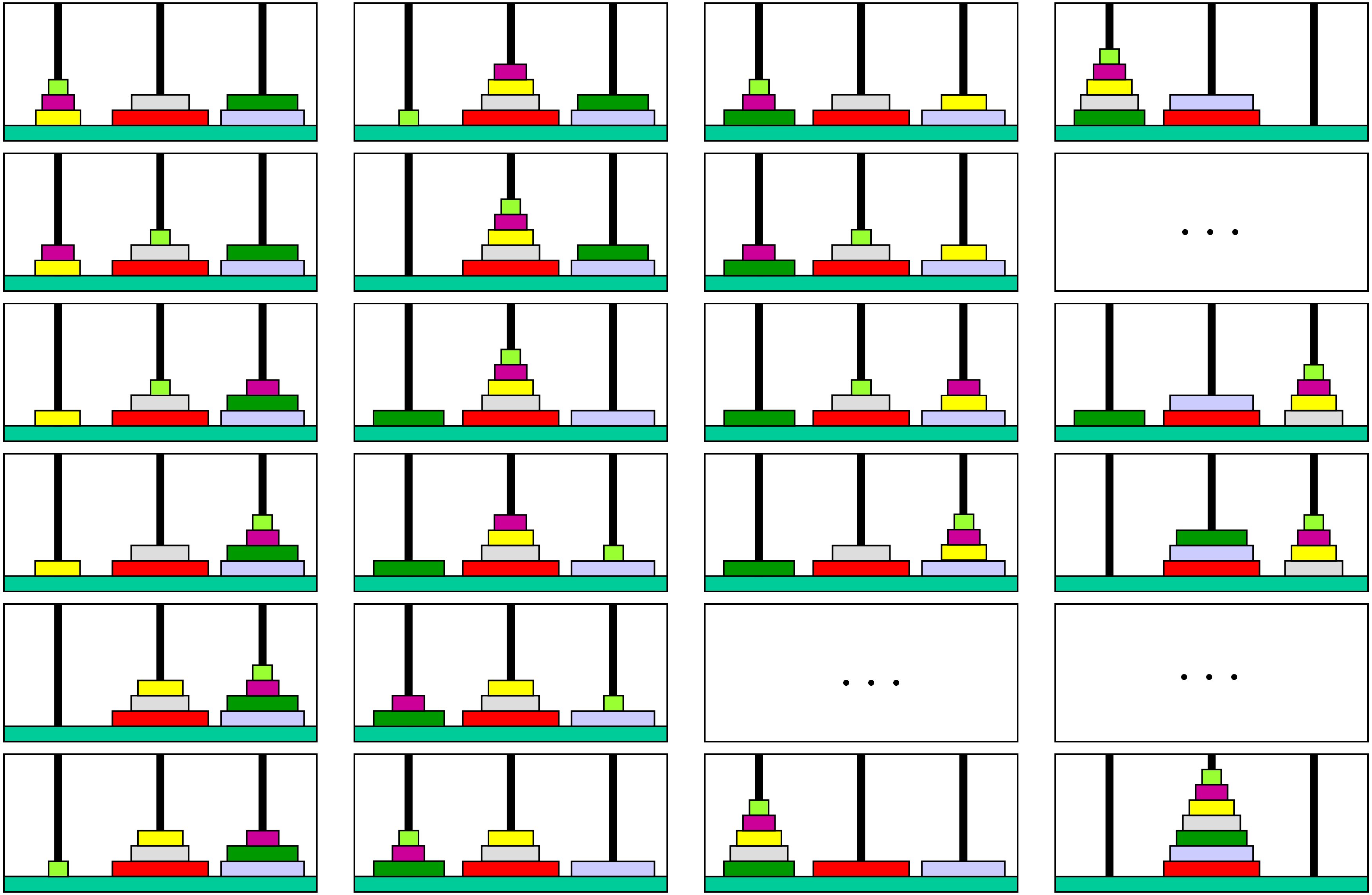
Towers of Hanoi 2



Towers of Hanoi 3



Towers of Hanoi 4



Example 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] []
```