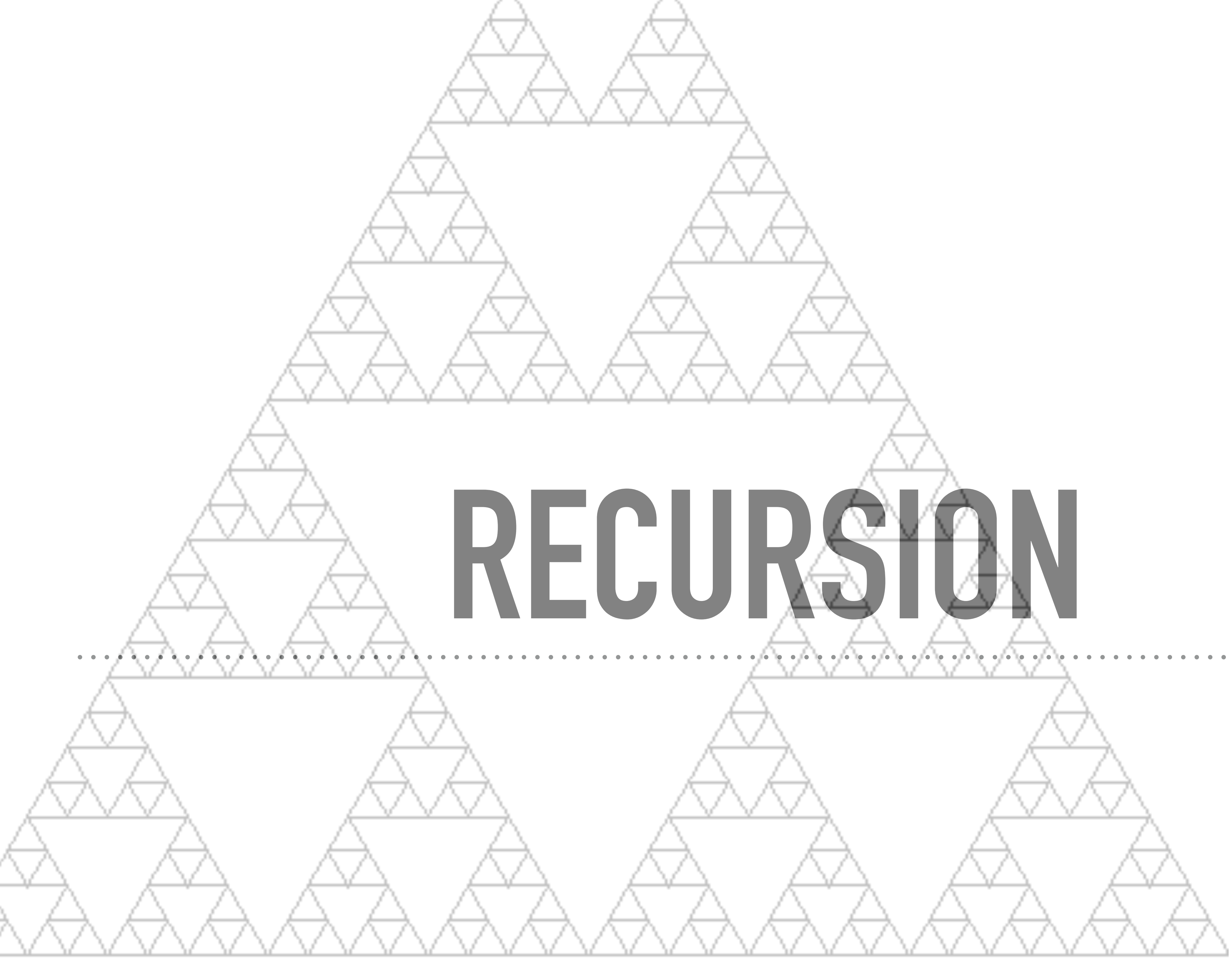
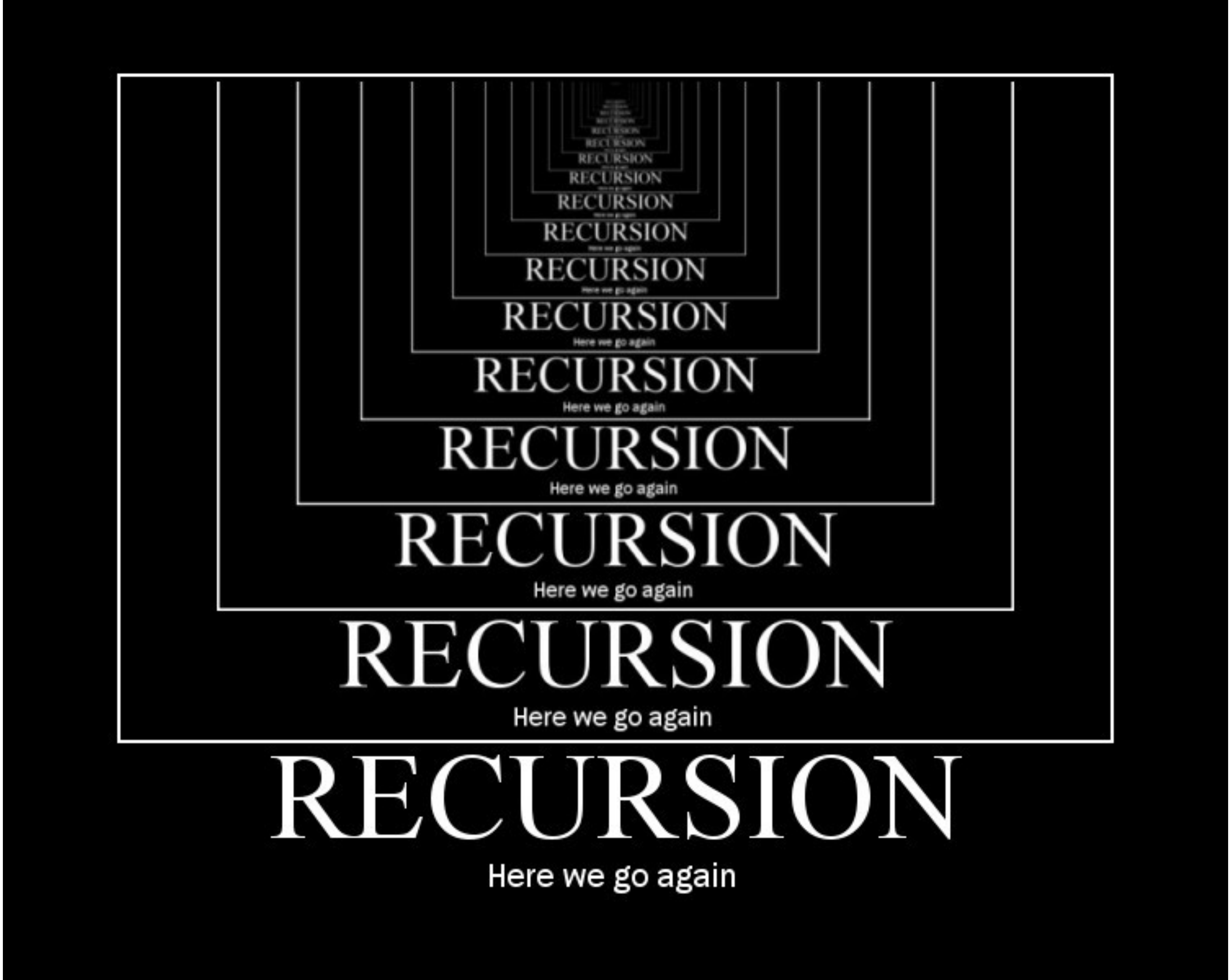
RECURSION



DEFINITION

- re·cur·sion (rəˈkərZHən) - the repeated application of a recursive procedure or definition. See *recursion*.

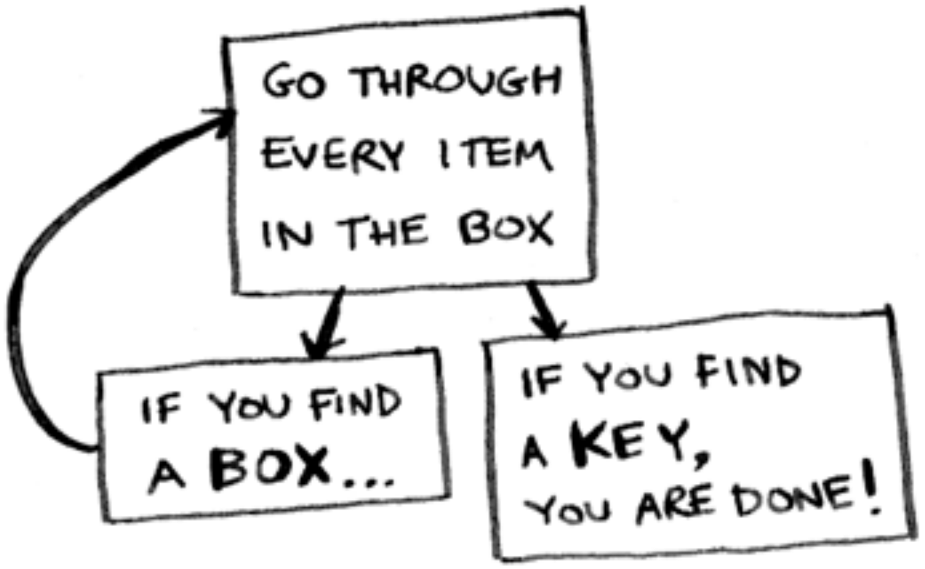
“In order to understand recursion, one must first understand recursion.”


RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION

RECURSION IS A FUNCTION THAT CALLS ITSELF

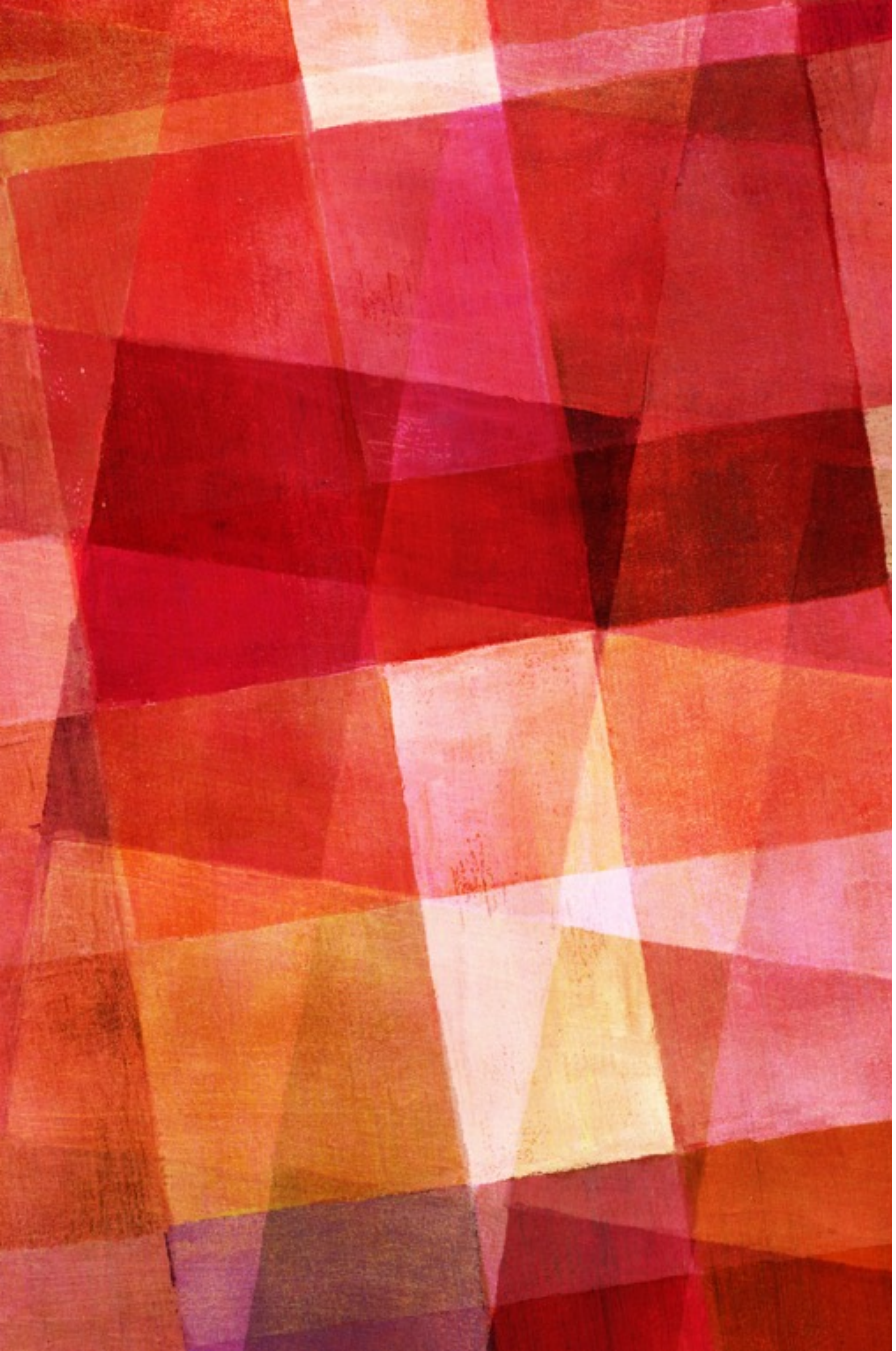
```
function look_for_key(box) {
  for (item in box) {
    if (item.is_a_box()) {
      look_for_key(item);
    } else if (item.is_a_key()) {
      console.log("found the key!")
    }
  }
}
```

Recursive Approach


GO THROUGH EVERY ITEM IN THE BOX
IF YOU FIND A BOX...
IF YOU FIND A KEY, YOU ARE DONE!

/jrs/code/teach/recursion/ demo01-countdown.js

```
function countdown(i) {
  console.log(i)
  if (i > 0) {
    countdown(i - 1)
  } else if (i === 0) {
    countdown('blastoff')
    return
  }
}
countdown(5) // This is the initial call to the function.
```



BENEFITS OF RECURSION

- Recursion can make the code very simple for some problems
- Ability to maintain state at different levels of recursion.
- Leads to less code.

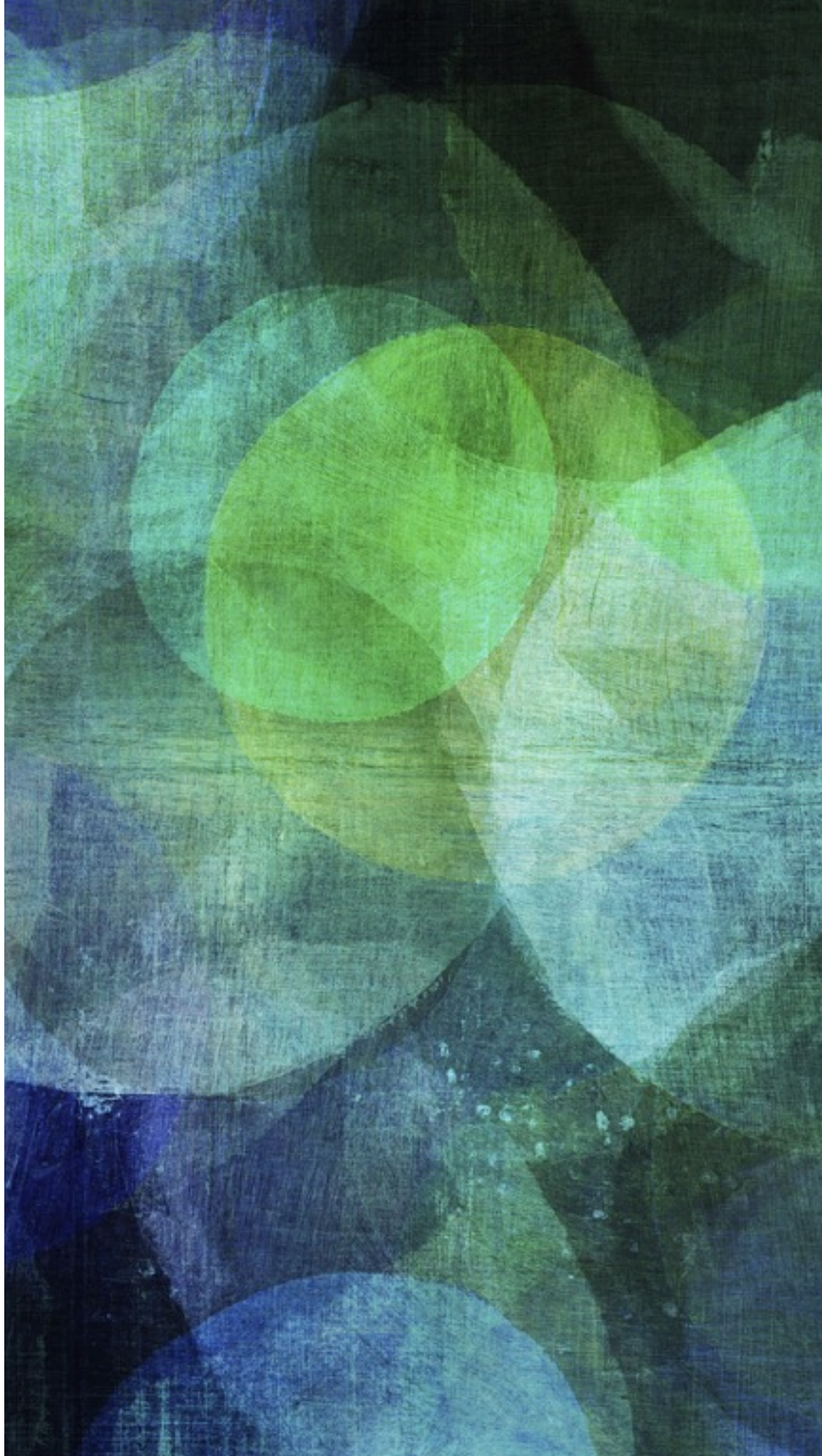
THE CALL STACK

- Recursive functions use something called “the call stack.”
- In computer science, a ***call stack*** is a stack data structure that stores information about the active subroutines of a computer program.
- This kind of stack is also known as an execution stack, program stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack".
- When a program calls a function, that function goes on top of the call stack. This similar to a stack of books.
- You add things one at a time.
- Ability to maintain state at different levels of recursion.
- In other words, you can’t access a value within an item in the stack from another item in the stack.
- Then, when you are ready to take something off (when the function call is completed), you always take off the top item in the stack.

DISADVANTAGES OF RECURSION

- No performance benefit over iterating through a loop.
- A recursive function calls itself.
- Every time the function is called the computer adds a call onto the stack.
- Too many calls and you will blow the stack. This is bad. This is known as “stack overflow”.
- Memory usage. A function that calls itself 200 times has to keep track of each call in memory.
  - FATAL ERROR: CALL_AND_RETRY_LAST Allocation failed - JavaScript heap out of memory

3 KEY FEATURES OF RECURSION



3 KEY FEATURES OF A RECURSIVE FUNCTION

1. Termination Case
2. Base Case
3. Recursion

#1 TERMINATION CONDITION EXAMPLE

- Compute the exponent of a number.
- A base number of 8 to the power of 3 would result in `8*8*8 = 512.`
- Here 8 is the base and 3 is the exponent.
  `power(8, 3)   // => 512`

```
const power = function(base, exp) {
  // A Termination Condition
  // this function will only work with non-negative integer
exponents
  // if exp is negative, return undefined
  if (exp < 0) {
    return
  }
  ...
```

#2 A BASE CASE

- *stops recursion just like the termination condition.*
- *base case represents our goal of the recursion.*
- *For example, if you were looking through a tree of folders and files on your hard drive looking for a particular file, you would recurse all the directories and files until you found the file. Your goal in this case would be finding the file.*
- *Once found, you would return some sort of result, such as the directory location of the file, and cease recursion.*

#2 BASE CASE EXAMPLE

- Recurse until you’ve reached the goal
- Once you have found the goal, stop recursing.
- Return a result to the previous caller in the stack
- ***Causes a chain reaction and begins to unwind “the stack”.***

```
const power = function(base, exp) {
  if (exp < 0) {
    return
  } else if (exp === 0) {
    return 1
  }
```

#3: THE RECURSION

- If you haven't reached the goal, ***recurse*** by having the function call itself.
- Each time place a call on the call stack including the base and exponent argument values at that point in time.
- Below the recursion is happening in the following statement `return base * power(base, exp - 1)`.
- Each time we recursively call the function, the call is placed onto the stack.
- The recursion repeats until we reach the base case
- Once we return from the base case, we begin to unwind back through the call stack with return values until we hit the original call on the stack and we have the final answer.

```
const power = function(base, exp) {

  if (exp < 0) {

    return

  } else if (exp === 0) {

    return 1

  } else {

    return base * power(base, exp - 1)

  }

}
```

EXERCISE 1

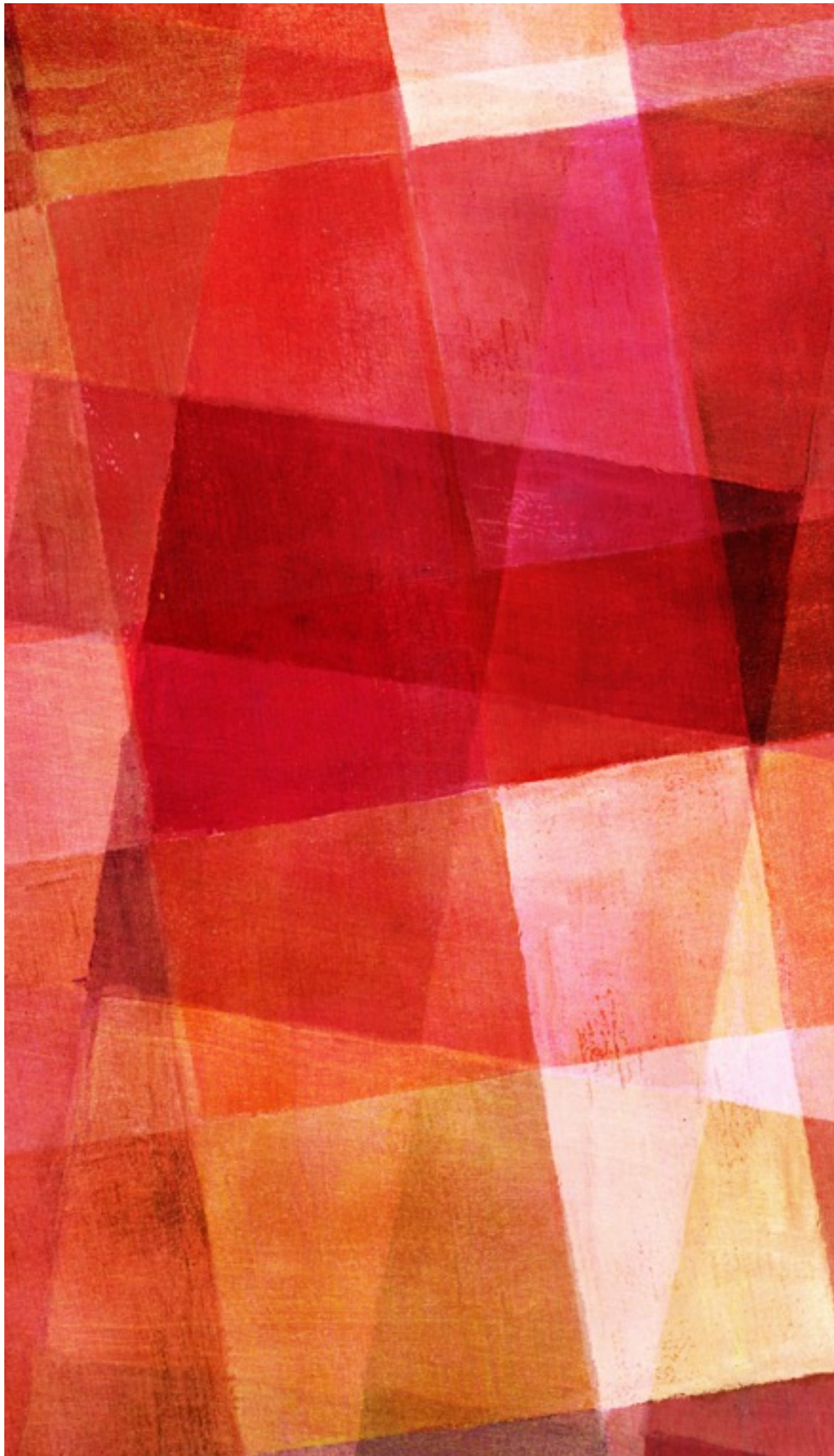
Create a recursive function named "lister" that takes two parameters:

1) max - the maximum number of items in the list.

2) current - the current number in the list.

Calling lister(6,1) with a maximum of 6 and the current item of 1 should log each item to the console like this:

```
1.
2.
3.
4.
5.
6.
```



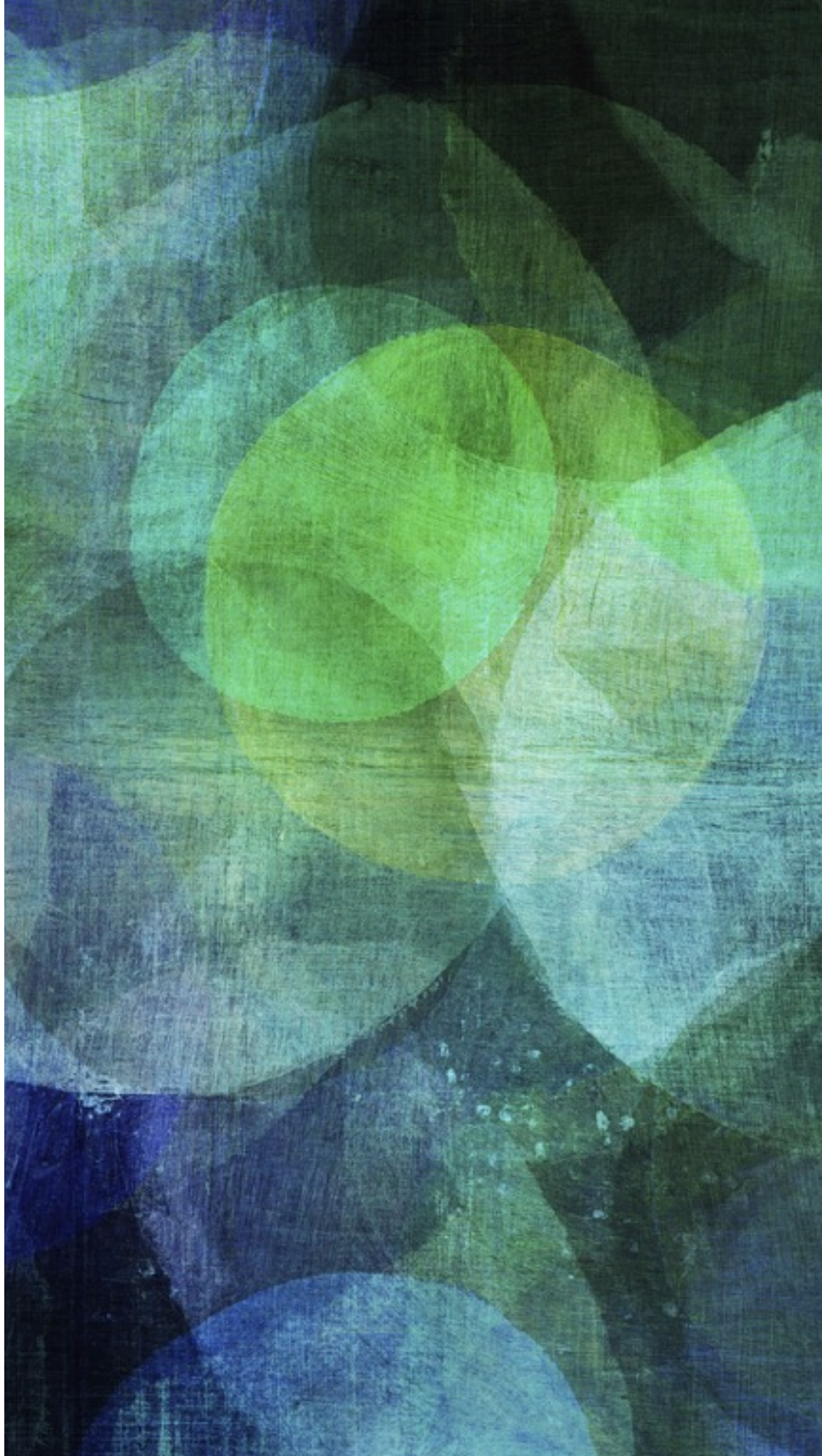
EXERCISE 2

write a recursive function to calculate the factorial of a number

Name the function "factorial"

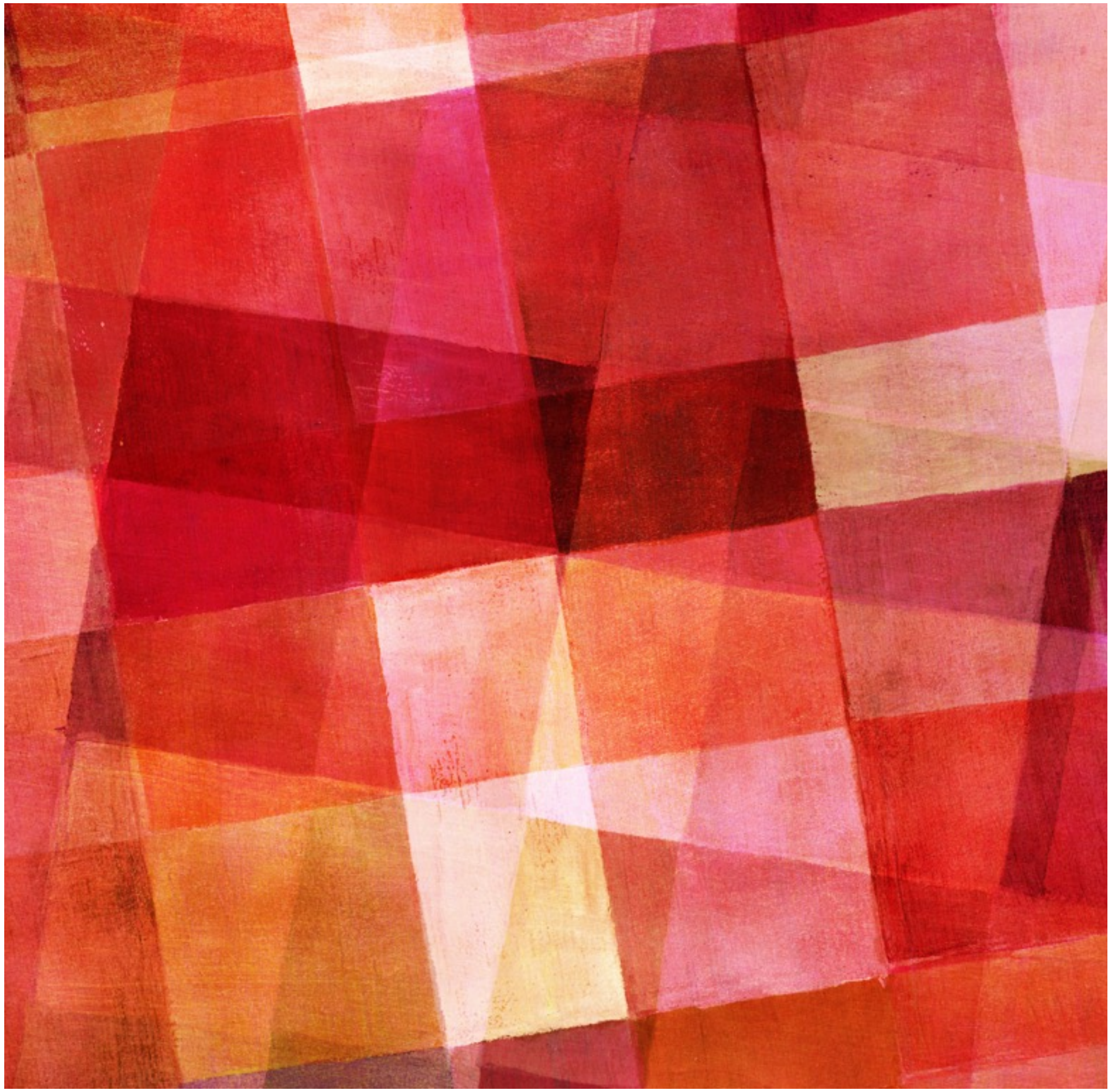
*5! = 5 * 4 * 3 * 2 * 1.*

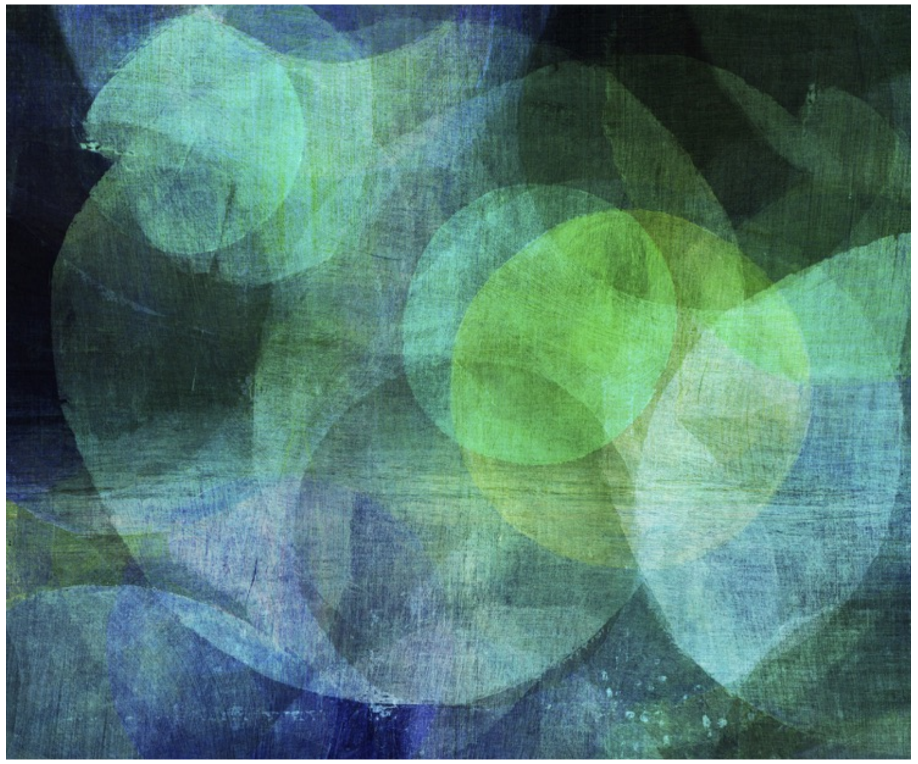
factorial(5) // 120



EXERCISE 3

Find the sum of an array of numbers recursively.





https://github.com/tripott/recursion.git