

# Recursion

## Base Case of a Recursive Function

A recursive function should have a base case with a condition that stops the function from recursing indefinitely. In the example, the base case is a condition evaluating a negative or zero value to be true.

```
function countdown(value)
  if value is negative or zero
    print "done"
  otherwise if value is greater than zero
    print value
    call countdown with (value-1)
```

## Recursive Step in Recursive Function

A recursive function should have a **recursive step** which calls the recursive function with some input that brings it closer to its base case. In the example, the recursive step is the call to `countdown()` with a decremented value.

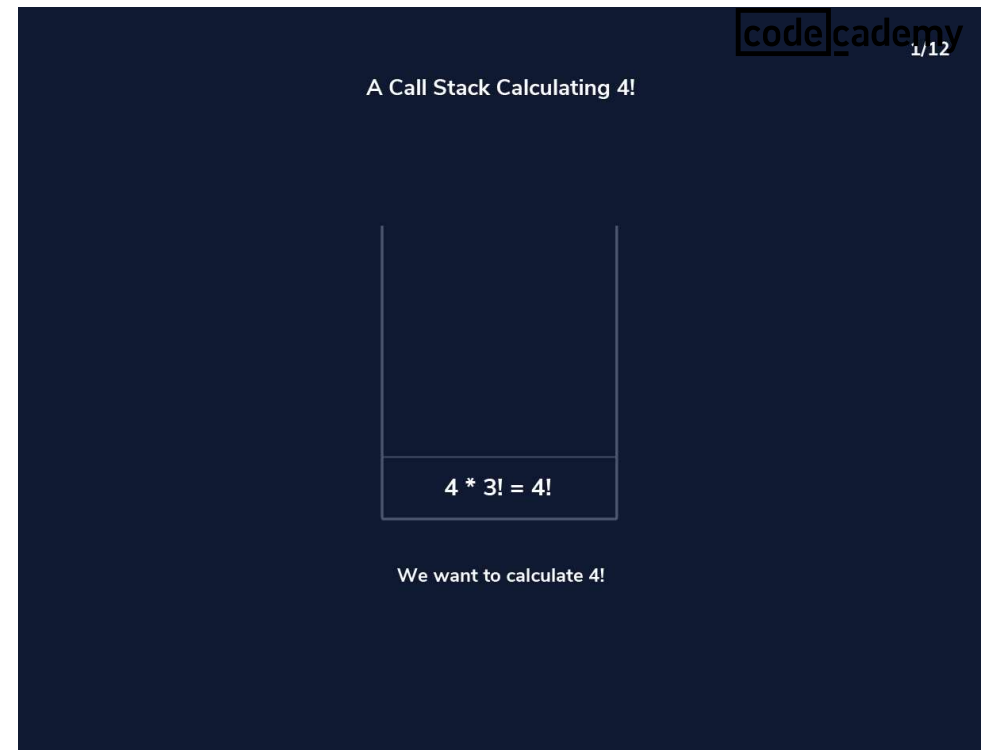
```
def countdown(value):
  if value <= 0:
    print("done")
  else:
    print(value)
    countdown(value-1)  #recursive step
```

## What is Recursion

Recursion is a strategy for solving problems by defining the problem in terms of itself. A recursive function consists of two basic parts: the base case and the recursive step.

## Call Stack in Recursive Function

Programming languages use a facility called a **call stack** to manage the invocation of recursive functions. Like a stack, a call stack for a recursive function calls the last function in its stack when the **base case** is met.



## Big-O Runtime for Recursive Functions

The big-O runtime for a recursive function is equivalent to the number of recursive function calls. This value varies depending on the complexity of the algorithm of the recursive function. For example, a recursive function of input N that is called N times will have a runtime of  $O(N)$ . On the other hand, a recursive function of input N that calls itself twice per function may have a runtime of  $O(2^N)$ .

## Weak Base Case in Recursive Function

A recursive function with a weak base case will not have a condition that will stop the function from recursing, causing the function to run indefinitely. When this happens, the call stack will overflow and the program will generate a *stack overflow* error.

## Execution Context of a Recursive Function

An execution context of a recursive function is the set of arguments to the recursive function call. Programming languages use execution contexts to manage recursive functions.

## Stack Overflow Error in Recursive Function

A recursive function that is called with an input that requires too many iterations will cause the call stack to get too large, resulting in a stack overflow error. In these cases, it is more appropriate to use an iterative solution. A recursive solution is only suited for a problem that does not exceed a certain number of recursive calls.

For example, `myfunction()` below throws a stack overflow error when an input of 1000 is used.

```
def myfunction(n):  
    if n == 0:  
        return n  
    else:  
        return myfunction(n-1)
```

```
myfunction(1000) #results in stack overflow error
```

## Fibonacci Sequence

A Fibonacci sequence is a mathematical series of numbers such that each number is the sum of the two preceding numbers, starting from 0 and 1.

```
Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

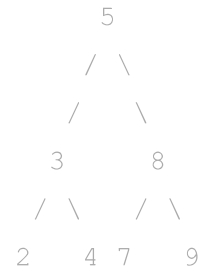
## Call Stack Construction in While Loop

A call stack with execution contexts can be constructed using a `while` loop, a `list` to represent the call stack and a `dictionary` to represent the execution contexts. This is useful to mimic the role of a call stack inside a recursive function.

## Binary Search Tree

In Python, a binary search tree is a recursive data structure that makes sorted lists easier to search. Binary search trees:

- Reference two children at most per tree node.
- The “left” child of the tree must contain a value lesser than its parent.
- The “right” child of the tree must contain a value greater than it’s parent.



## Recursion and Nested Lists

A nested list can be traversed and flattened using a recursive function. The base case evaluates an element in the list. If it is not another list, the single element is appended to a flat list. The recursive step calls the recursive function with the nested list element as input.

```
def flatten(mylist):
    flatlist = []
    for element in mylist:
        if type(element) == list:
            flatlist += flatten(element)
        else:
            flatlist += element
    return flatlist

print(flatten(['a', ['b', ['c', ['d']], 'e'], 'f']))
# returns ['a', 'b', 'c', 'd', 'e', 'f']
```

## Fibonacci Recursion

Computing the value of a Fibonacci number can be implemented using recursion. Given an input of index N, the recursive function has two base cases – when the index is zero or 1. The recursive function returns the sum of the index minus 1 and the index minus 2.

The Big-O runtime of the Fibonacci function is  $O(2^N)$ .

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

## Modeling Recursion as Call Stack

One can model recursion as a call stack with execution contexts using a while loop and a Python list. When the base case is reached, print out the call stack list in a LIFO (last in first out) manner until the call stack is empty. Using another while loop, iterate through the call stack list. Pop the last item off the list and add it to a variable to store the accumulative result. Print the result.

```
def countdown(value):
    call_stack = []
    while value > 0:
        call_stack.append({"input": value})
        print("Call Stack:", call_stack)
        value -= 1
    print("Base Case Reached")
    while len(call_stack) != 0:
        print("Popping {} from call
stack".format(call_stack.pop()))
        print("Call Stack:", call_stack)
    countdown(4)
'''
Call Stack: [{'input': 4}]
Call Stack: [{'input': 4}, {'input': 3}]
Call Stack: [{'input': 4}, {'input': 3}, {'input': 2}]
Call Stack: [{'input': 4}, {'input': 3}, {'input': 2},
{'input': 1}]
Base Case Reached
Popping {'input': 1} from call stack
Call Stack: [{'input': 4}, {'input': 3}, {'input': 2}]
Popping {'input': 2} from call stack
Call Stack: [{'input': 4}, {'input': 3}]
Popping {'input': 3} from call stack
Call Stack: [{'input': 4}]
Popping {'input': 4} from call stack
Call Stack: []
'''
```

In Python, a recursive function accepts an argument and includes a condition to check whether it matches the base case. A recursive function has:

- Base Case – a condition that evaluates the current input to stop the recursion from continuing.
- Recursive Step – one or more calls to the recursive function to bring the input closer to the base case.

## Build a Binary Search Tree

To build a binary search tree as a recursive algorithm do the following:

BASE CASE:

If the list is empty, return "No Child" to show that there is no

RECURSIVE STEP:

1. Find the middle index of the list.
2. Create a tree node with the value of the middle index.
3. Assign the tree node's left child to a recursive call with the
4. Assign the tree node's right child to a recursive call with the
5. Return the tree node.

```
def countdown(value):
    if value <= 0:    #base case
        print("done")
    else:
        print(value)
        countdown(value-1)    #recursive case
```

```
def build_bst(my_list):
    if len(my_list) == 0:
        return "No Child"

    middle_index = len(my_list) // 2
    middle_value = my_list[middle_index]

    print("Middle index: {}".format(middle_index))
    print("Middle value: {}".format(middle_value))

    tree_node = {"data": middle_value}
    tree_node["left_child"] = build_bst(my_list[:
middle_index])
    tree_node["right_child"] =
build_bst(my_list[middle_index + 1 : ])

    return tree_node

sorted_list = [12, 13, 14, 15, 16]
binary_search_tree = build_bst(sorted_list)
print(binary_search_tree)
```