Recursion

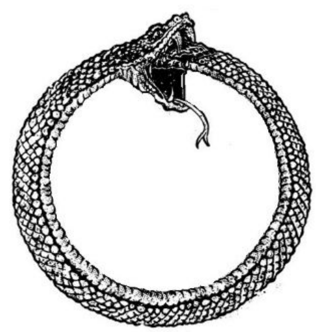
Objectives:

- What is Recursion
- Recursion with Returned Value
- Visualizing Recursion
- Recursion with Returned Value

1. What is Recursion

The process in which a function calls itself directly or indirectly is known as `recursion`. The corresponding function is called as `recursive function`.

- A `recursive function` **repeats the same action (statement)** in each iteration.



https://www.slideshare.net/HaseebQureshi5/recursion-for-the-rest-of-us-cs-fundamentals-series

Base Case

- In each iteration, it breaks down a problem into smaller subproblems which is small enough to be solved trivially. Such small problem is called **base case**.
- The solution of the bigger problem is implemented by calling the funtion itself.
- The solution to the base case is implemented directly without calling to itself.

Termination Condition

- Recursion function must have a **termination condition**, which stops the function from calling itself infinitely.

Why Use Recursion?

- Recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

Types of Recursive Function

- Recursive Function without Return Value

- Recursive Function with Return Value

Recursion instead of Loop

Exercise:

Implement a **count-down()** function using `while-loop` .

- It takes in a parameter `n` , which is the starting number to count down.

Sample Output: Calling to `count_down(3)` gives following printout.

```
3 2 1 Done!
```

In [3]:
```
def count_down(n):
    while n > 0:
        print(n, end=' ')
        n = n - 1
    print('Done!')

count_down(3)
```

```
3 2 1 Done!
```

Question:

How to convert above function `count_down()` into a recursive function?

Analysis Steps:

- What is the common actions in each loop? (Hint: check the loop statements)
- What is its termination condition?
- What is its base case, i.e. what does it do when it is at termination condition?

Analysis Result:

- In each iteration, it prints out current `n` value.
- The termination condiction is `n > 0` .
- If termination condition is True, it prints `Done` .

Thus,

- In the recrusive function, it check termination condiction `n > 0` .
- If termination condition is true, it prints `Done` , else it prints `current n value` and recurses (call its own function).

Exercise:

Convert above function `count_down()` into recursive function `recursive_down()` .

In [9]:
```
def recursive_down(n):
    if n > 0:
        print(n, end=' ')
        recursive_down(n-1)
    else:
        print('Done!')

recursive_down(3)
```

```
3 2 1 Done!
```

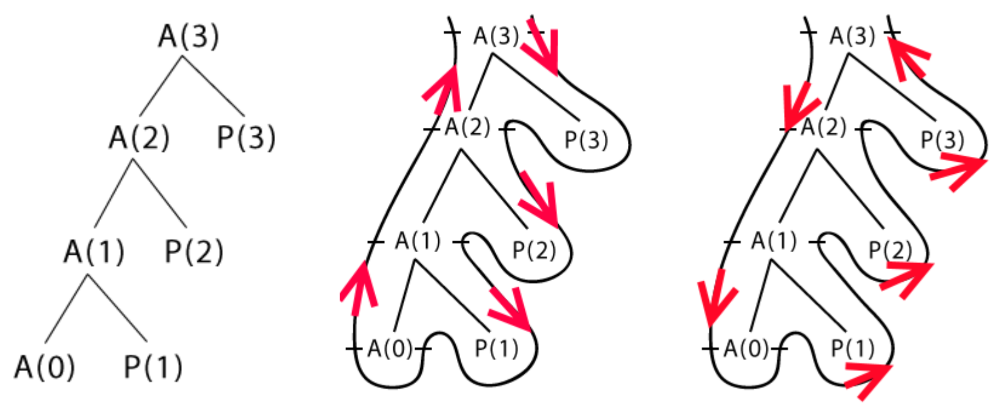
2. Visualizing Recursive Execution (Important)

Question:

- How do you visualize above recursive function execution?

Draw out function calls diagram and follow execution order.

Funciton Calls	Execution Order (Count-down)	Execution Order (Count-up)



Question:

Give minimal modification to `recursive_down()` function for it to **count-up** instead of **count-down**?

Sample Output:

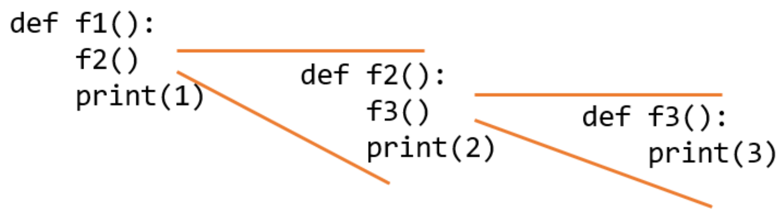
```
Done!
1 2 3
```

In [46]:
```python
def recursive_down(n):
    if n > 0:
        recursive_down(n-1)
        print(n, end=' ')
    else:
        print('Done!')

recursive_down(3)
```

```
Done!
1 2 3
```

Calling Stack of Functions

Call stack of functions determines the precedence of printouts in following code sample.



In [ ]:

2. Recursion with Return Value

In above count-down example, the recursion function does not return any value. It is common for recursion function to return value back to calling function.

Factorial

The factorial value of a explicit number `n` is typically represented as `n!` , and $F_n = n * F_{n-1}$.

```
n! = n * (n-1) * (n-2) * . . . * 1
n! = n * (n-1)!
```

Exercise:

Implement function `facorial_loop(n)` to calculate **factorial** value of an integer `n` using `while-loop` .

In [6]:
```
def facorial_loop(n):
    result = 1
    while n >= 1:
        result = result * n
        n = n - 1
    return result

fac_loop(4)
```

Out[6]: 24

Question:

How to convert above function into recursive function `facorial_recurse()` ?

<u>Analysis Steps:</u>

- What is the common actions in each iteration? (Hint: check the loop statements)
- What is its termination condition?
- If termination condition is True, what does it do?

<u>Analysis Result:</u>

- It sets `result = result * n` , and it recurse with `n-1` (common action)
- The termination condiction is `n == 1` .
- If condition is True, it returns `1` .

<u>Recursive Call:</u>

```
facorial_recurse(n) = n * facorial_recurse(n-1)
```

Exercise:

Implement function `facorial_recurse(n)` to calculate factorial value of an integer n using `recursion` .

In [9]:
```
def facorial_recurse(n):
    if n > 1:
        return n * facorial_recurse(n-1)
    else:
        return 1

facorial_recurse(4)
```

Out[9]: 24

Fibonacci

The Fibonacci Sequence is the series of numbers, where next number is found by adding up the two numbers before it.

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0$ and $F_1 = 1$.

Try Code:

Sample solution using `while-loop` :

```
def fib_loop(n):
    if n <= 1:
        return n
    else:
        f0, f1 = 0, 1
        i = 2
        while i <= n:
            f2 = f0 + f1
            f0, f1 = f1, f2
            i = i + 1
        return f2

fib_loop(5)
```

In [48]:
```
def fib_loop(n):
    if n <= 1:
        return n
    else:
        f0, f1 = 0, 1
        i = 2
        while i <= n:
            f2 = f0 + f1
            f0, f1 = f1, f2
            i = i + 1
        return f2

fib_loop(5)
```

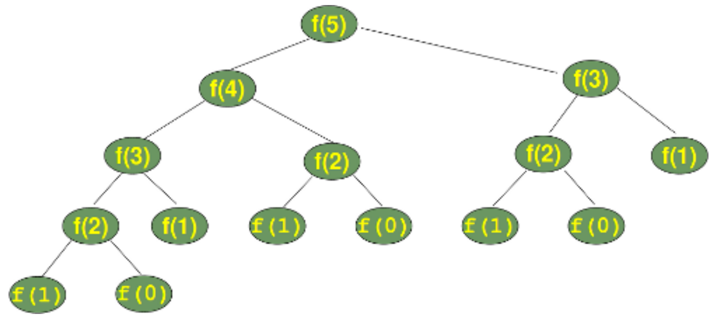
```
593 ns ± 9.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

We can also ask following questions:

- What is the common actions in each iteration? (Hint: check the loop statements)
- What is its termination condition?
- If termination condition is True, what does it do?

Fibonacci

```
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2);
...
fibonacci(1) = 1
fibonacci(0) = 0
```



Exercise:

Implement a recursive function `fib_recure()` to generate Fibonacci number.

- It takes in a parameter `n` and returns `nth` Fibonacci number.

In [54]:
```
def fib_recure(n):
    if n==0 or n == 1:
        return n
    else:
        return fib_recure(n-1) + fib_recure(n-2)
```

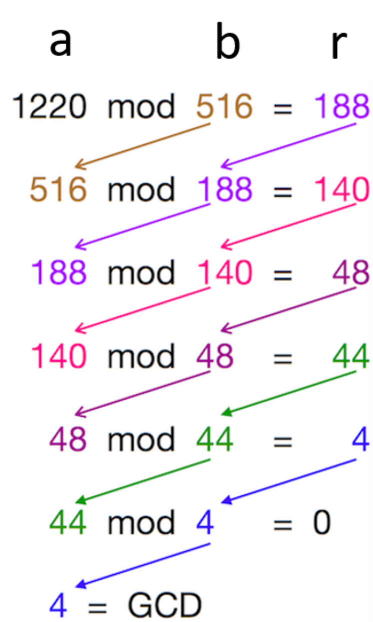
Great Common Divisor

The **greatest common divisor (gcd)** of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.

The [Euclidean Algorithm (https://en.wikipedia.org/wiki/Euclidean_algorithm)](https://en.wikipedia.org/wiki/Euclidean_algorithm) provides an efficient method for computing the greatest common divisor (GCD) of two numbers, e.g. `a` and `b`.

- If `a % b == 0`, then `b` is the GCD
- else result is the same as GCD of `b` and `a % b`

For example, when `a = 1220` and `b = 516`, GCD is found to be 4.



Exercise:

Implement a function `gcd_loop()` using `while-loop` to calculate great common divisor (gcd) using Euclidean Algorithm.

- It takes in 2 parameters `a` and `b`.
- It returns gcd of `a` and `b`.

In [41]:
```
def gcd_loop(a,b):
    r = a % b
    while r != 0:
        a, b = b, r
        r = a % b
    return b

gcd_loop(1220, 516)
```

Out[41]: 4

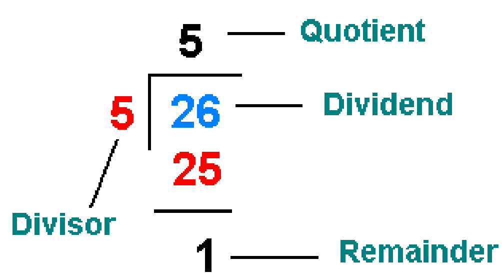
Exercise:

Implement a recursive function `gcd_recurse(a, b)` which returns great common divisor (gcd) of its 2 input parameters `a` and `b`.

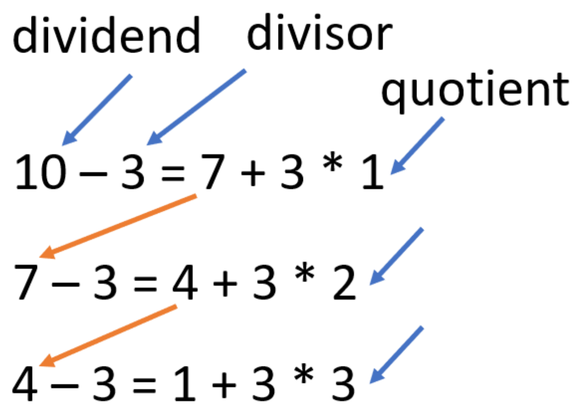
In [40]:
```
def gcd_recurse(a,b):
    r = a % b
    if r == 0:
        return b
    return gcd_recurse(b, r)

gcd_recurse(1220, 516)
```

Out[40]: 4

Quotient & Remainder



To find the quotient and remainder of two numbers, we can also do it recursively.



Exercise:

Implement a function `div_loop(dividend, divisor)` using `while-loop` , which returns quotient and remainder of its 2 input parameters `dividend` and `divisor` .

In [44]:
```
def div_loop(dividend, divisor):
    q = 0
    while dividend >= divisor :
        dividend = dividend - divisor
        q = q + 1
    return q, dividend

div_loop(10, 3)
```

Out[44]: (3, 1)

Exercise:

Implement a **recursive** function `div_recurse(dividend, divisor)` which returns quotient and remainder of its 2 input parameters `dividend` and `divisor` .

In [45]:
```
def div_recurse(dividend, divisor, q = 0):
    if dividend >= divisor:
        return div_recurse(dividend-divisor, divisor, q+1)
    else:
        return q, dividend

div_recurse(10, 3)
```

Out[45]: (3, 1)

Maintain State in Recursion

In recursive function, each recursive call has its own execution context, how to maintain state, i.e. share variables across recrusive calls?

- Use variables of global scope
- Pass the variables through recursive calls

Question:

In the **Quotient and Remainder** example, how do we maintain the state of variable `q` ?

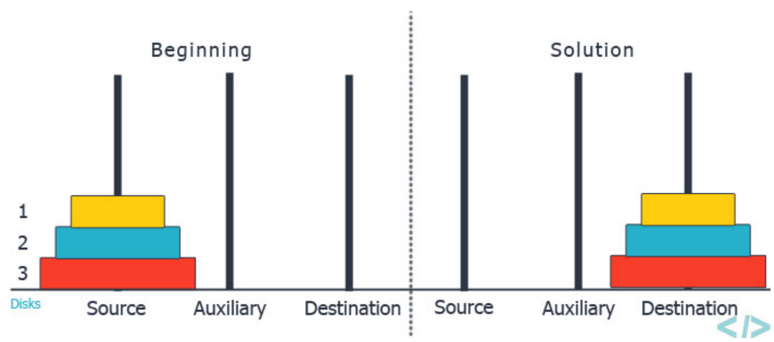
3. Tower of Hanoi

The Towers of Hanoi is a mathematical puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod.

The puzzle starts with the disks on one tower in ascending order of size, the smallest at the top, making a conical shape. The objective of the puzzle is to move entire stack on another tower with satisfying below rules:

Rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the towers and move it onto another tower, on top of the other disks if there is any.
- No disk can be placed on top of a smaller disk.

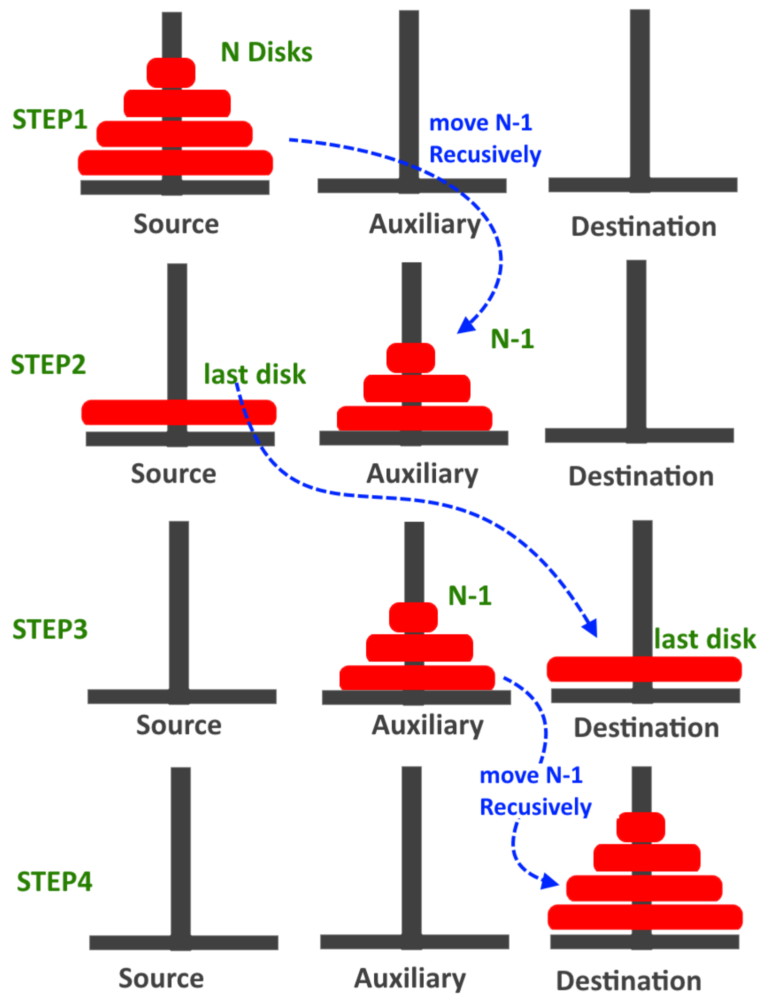


Reference: https://codenuclear.com/towers-of-hanoi/

Algorithm

Approach

- Recursively Move N-1 disk from source to Auxiliary peg.
- Move the last disk from source to destination.
- Recursively Move N-1 disk from Auxiliary to destination peg.



Reference: https://algorithms.tutorialhorizon.com/towers-of-hanoi/

Naming

- `For a tower of n disks` , smallest disk is `disk 1` and largest disk is `disk n`
- `For a tower of n-1 disks` , smallest disk is still `disk 1` and largest disk is `disk n-1`

Recursive

At Step 1, the task is to:

- `Move tower of n-1 disks from source to auxiliary`
- `Move disk n from source to destination`
- `Move tower of n-1 disks from auxiliary to source`

At Step 3, the task become following recursion:

- `Move tower of n-2 disks from auxiliary to source`
- `Move disk n-1 from auxiliary to dest`
- `Move tower of n-2 disks from source to auxiliary`

NOTE: The `auxiliary pole` become `source pole`, and `source pole` become `auxiliary pole` in this recursion.

Implementation

We will implement 2 functions, `move_disk()` and `move_tower()`.

`move_disk(x, src, dest)`

- It represents moving a `disk x` from `src` pole to `dest` pole.
- It prints a message `Move disk {x} from {src} to {dest}`.

In [80]:
```
def move_disk(x, src, dest):
    print('Move disk {} from {} to {}'.format(x, src, dest))

move_disk(2, 'S', 'D')
```

```
Move disk 2 from S to D
```

`move_tower(n, src, aux, dest)`

- It represents moving a `tower of n disk` from `src` pole to `dest` pole. The `aux` pole can be used to facilitate moving.
- This is a recursive function.

base case:

- When n = 0, do nothing
- When n = 1, move the disk from `src` to `dest`

recursion in each loop:

- `Move tower of n-1 disks` from `src` to `aux`
- `Move disk n` from `src` to `dest`
- `Move tower of n-1 disk` from `aux` to `src`

In [79]:
```
def move_tower(n, src, aux, dest):
    if n == 0:
        return
    if n == 1:
        move_disk(n, src, dest)
    else:
        move_tower(n-1, src, dest, aux)
        move_disk(n, src, dest)
        move_tower(n-1, aux, src, dest)
```

Tests

In [82]:
```
move_tower(1, 'S', 'A', 'D')
```

```
Move disk 1 from S to D
```

In [81]:
```
move_tower(2, 'S', 'A', 'D')
```

```
Move disk 1 from S to A
Move disk 2 from S to D
Move disk 1 from A to D
```

In [78]:
```
move_tower(3, 'S', 'A', 'D')
```

```
Move 1 from S to D
Move 2 from S to A
Move 1 from D to A
Move 3 from S to D
Move 1 from A to S
Move 2 from A to D
Move 1 from S to D
```

4. Summary

Advantages

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages

- Sometimes the logic behind recursion is hard to follow through.
- Recursive functions are hard to debug.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

More Memory Usage

Everytime a function calls itself and stores some memory. Thus, a recursive function could hold much more memory than a traditional function.

- Python stops the function calls after a depth of 1000 calls, and throws a `RecursionError: maximum recursion depth exeeded..` error.
- You can increase recursion depth using `sys.setrecursionlimit(limit)` function.

```
import sys

sys.setrecursionlimit(1050)
fib(1050)
```

Reference

- https://www.python-course.eu/recursive_functions.php (https://www.python-course.eu/recursive_functions.php)
- https://realpython.com/python-thinking-recursively/#dear-pythonic-santa-claus (https://realpython.com/python-thinking-recursively/#dear-pythonic-santa-claus)
- https://www.python-course.eu/python3_recursive_functions.php (https://www.python-course.eu/python3_recursive_functions.php)
- https://study.com/academy/lesson/recursion-recursive-algorithms-in-python-definition-examples.html (https://study.com/academy/lesson/recursion-recursive-algorithms-in-python-definition-examples.html)
- IterativePython: Problem Solving with Algorithms and Data Structures - Recursion (http://interactivepython.org/courselib/static/pythonds/Recursion/toctree.html)
- Datacamp: Understanding Recursive Functions in Python (https://www.datacamp.com/community/tutorials/understanding-recursive-functions-python)