# Recursion

*Function calls itself*

## Advantages:

- No loops needed.
- Solves problems without iteration.

*"To understand recursion you must understand recursion"*

# Iterative Vs Recursive

## a*b

In [7]:
```
# iterative method
def multiply(a, b): # a = 2 b = 3
    result = 0
    for i in range(b):
        result += a
    print(result)
multiply(7,3)
```

```
21
```

In [1]:
```python
# Recursive Method

# 1. Base Case: Define stopping condition.

# 2. Decompose: Break main problem into smaller subproblems until base case is reached.

def mul(a, b):
    if b == 1:
        return a
    else:
        return a + mul(a, b-1)
```

In [2]:
```python
print(mul(3, 4))
```

```
12
```

In [9]:
```python
# Factorial via Recursion
def fact(number):
    if number == 1:
        return 1
    else:
        return number * fact(number-1)
print(fact(5))
```

```
120
```

In [10]:
```python
# Palindrome
def palin(text):
    if len(text) <= 1:
        print("palindrome")
    else:
        if text[0] == text[-1]:
            palin(text[1:-1])
        else:
            print("not a palindrome")
```

In [15]:
```
palin("nitin")
```

```
palindrome
```

In [16]:
```
palin("malayalam")
```

```
palindrome
```

In [17]:
```
palin("python")
```

```
not a palindrome
```

In [18]:
```
palin("mom")
```

```
palindrome
```

In [6]:
```
# The Rabbit Problem: Fibonacci Number

# Scenario ---> 2 newborn rabbits: 1 male + 1 female monthly.
#               Reproduce after 1 month, immortality.

def fib(m):
    if m == 0 or m == 1:
        return 1
    else:
        return fib(m-1) + fib(m-2)
print(fib(4)) # T = O(2^n)

# Key Concepts:
#     Fibonacci Sequence
#     Reproduction Rate
#     Population Growth
```

```
5
```

In [3]:
```
import time
start = time.time()
print(fib(12))
print(time.time() - start)
```

```
233
0.0
```

In [4]:
```
print(fib(24))
print(time.time() - start)
```

```
75025
4.907310247421265
```

In [5]:
```
print(fib(36))
print(time.time() - start)
```

```
24157817
15.004727840423584
```

# Memoization

Memoization refers to remembering method call results based on inputs.

- Returns cached results, avoiding recomputation.
- Speeds up computations; stores previous results.
- Used in dynamic programming for recursive solutions.
- Reduces time complexity; avoids redundant calculations.
- Optimizes recursive algorithms by reusing results.

In [16]:
```
def memo(m, d):
    if m in d:
        return d[m]
    else:
        d[m] = memo(m-1, d) + memo(m-2, d)
        return d[m]
d = {0:1, 1:1}
print(memo(48, d))
```

```
7778742049
```

In [17]:
```
print(memo(48, d))
print(time.time() - start)
```

```
7778742049
6.157997369766235
```

In [18]:
```
print(memo(500, d))
print(time.time() - start)
```

```
225591516161936330872512695036072072046011324913758190588638866418474627738686883405015987052796968498626
6.17300009727478
```

In [19]:
```
print(memo(1000, d))
print(time.time() - start)
```

```
70330367711422815821835254877183549770181269836358732742604905087154537118196933579742249494562611733487750449241765991
08818636326545022364710601205337412127386733911119813937312559876769009190224524532340350 1
6.189241170883179
```

In [20]:
```
print(d) # Dict in memory, execution time reduced
```

In [21]:
```python
# Recursive PowerSet Function in Python

# PowerSet: Given set S, return power set P(S) (all subsets of S).

# Input: String
# Output: Array of Strings (power set)

# Example: S = "123", P(S) = ['', '1', '2', '3', '12', '13', '23', '123']

def powerset1(xs):
    res = [[]]
    if len(xs) <= 0:
        return "Please Enter a parameter"
    if len(xs) == 1:
        res.append([xs[0]])
        return res
    else:
        z = []
        for i in powerset1(xs[1:]):
            z.append(i)
            z.append([xs[0]] + i)
        return z

final = powerset1('123')
print(final)
print(len(final))
```

```
[[], ['1'], ['2'], ['1', '2'], ['3'], ['1', '3'], ['2', '3'], ['1', '2', '3']]
8
```