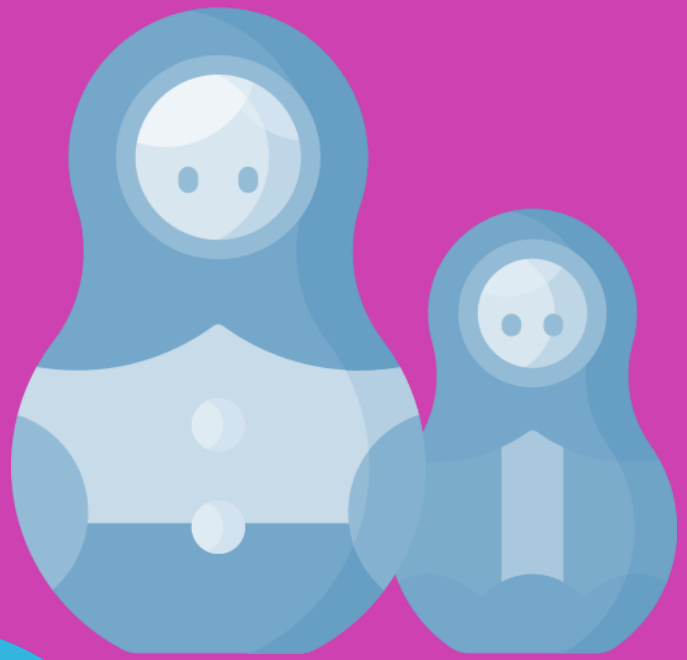
RECURSIVIDAD

Algoritmos y estructuras de datos I



¿QUÉ ES LA RECURSIVIDAD?

Se define como un proceso en el que una función se llama a sí misma para resolver un problema, implicando definir un problema en función de sí mismo.

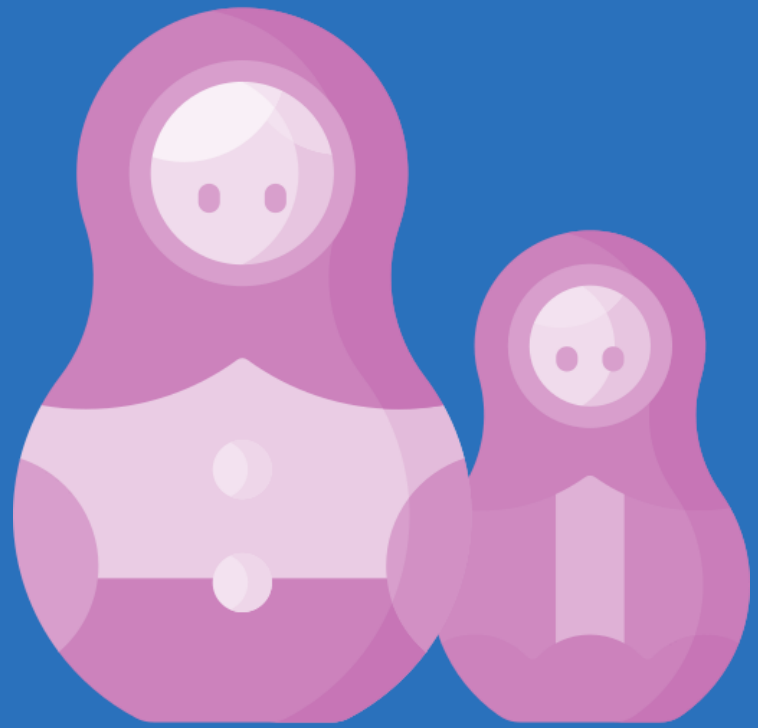
Una función recursiva debe tener una condición de terminación, conocida como caso base, que detiene las llamadas recursivas. Además, debe realizar una o más llamadas recursivas con parámetros que eventualmente converjan hacia el caso base.

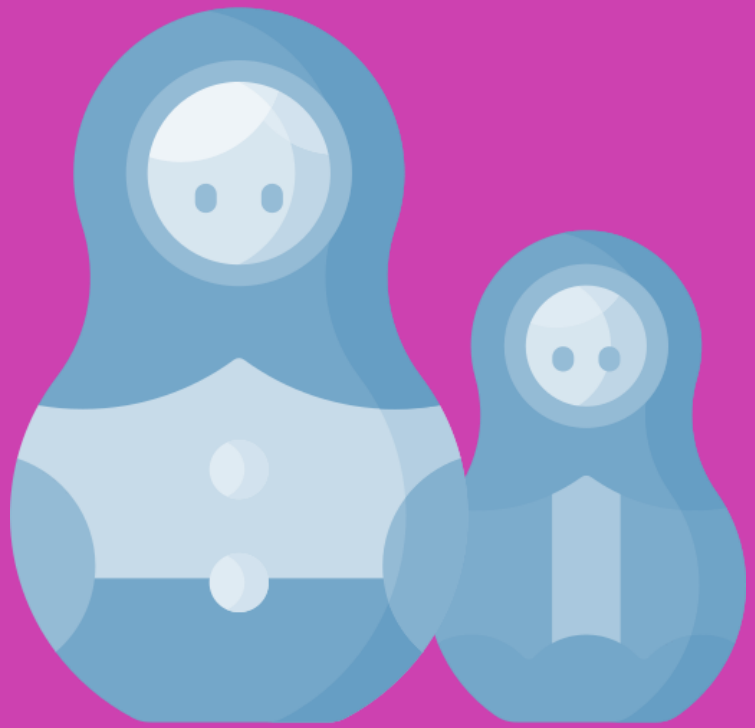
La descomposición del problema debe permitir alcanzar el caso base en un número finito de pasos, lo que garantiza que el proceso recursivo finalice de manera adecuada y no entre en un bucle infinito.

CASO BASE

Para desarrollar algoritmos recursivos hay que partir del supuesto de que ya existe un algoritmo que resuelve una versión más sencilla del problema. De esta manera, conocemos la condición de terminación o solución donde la función podría devolver inmediatamente los resultados.

Es importante establecer el momento en que se dejará de llamarse a sí misma, de lo contrario se obtendrá una función recursiva infinita.





ESTRUCTURA RECURSIVA

Cuando se invoca la resolución de un subproblema, el flujo de ejecución principal se suspende temporalmente, y el enfoque se desplaza al subproblema.

El resultado devuelto se utiliza entonces para resolver el problema en un nivel superior de la recursión. A medida que se completan las llamadas recursivas y se obtienen los resultados de los subproblemas, el proceso avanza hacia la solución del problema original.

Finalmente, cuando se resuelve el problema original, se obtiene el resultado deseado.

Es esencial tener en cuenta que cada llamada recursiva debe acercar el proceso hacia el caso base, asegurando así que el algoritmo eventualmente se detenga.

FUNCIÓN RECURSIVA SIN RETORNO

Es una función que se llama a sí misma repetidamente para resolver un problema sin devolver un valor explícito al proceso que la invoca.

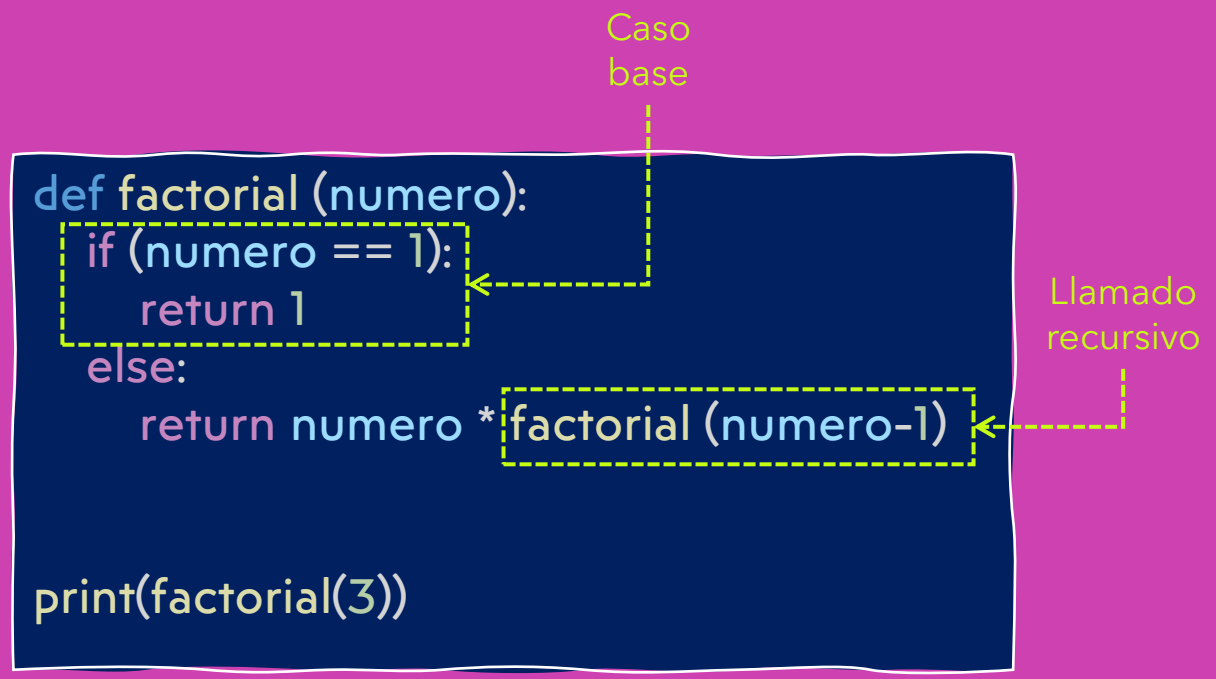
Aunque la función no devuelve un valor, sigue siendo crucial establecer una condición de terminación para evitar que el proceso recursivo se ejecute indefinidamente.

```
def cuentaRegresiva(numero):
    numero -= 1

    if numero > 0:
        print(numero)
        cuentaRegresiva(numero)
    else:
        print ("Fin de la cuenta regresiva.")


cuentaRegresiva(11)
```

Llamado recursivo

Caso base

FUNCIÓN RECURSIVA CON RETORNO



El cálculo del factorial de un número es un ejemplo clásico para comprender la recursividad en la programación.

El factorial de un número *n*, denotado como *n!*, se define como el producto de todos los números enteros positivos desde 1 hasta *n*. Por ejemplo, el factorial de 3!, se calcula multiplicando 3×2×1.

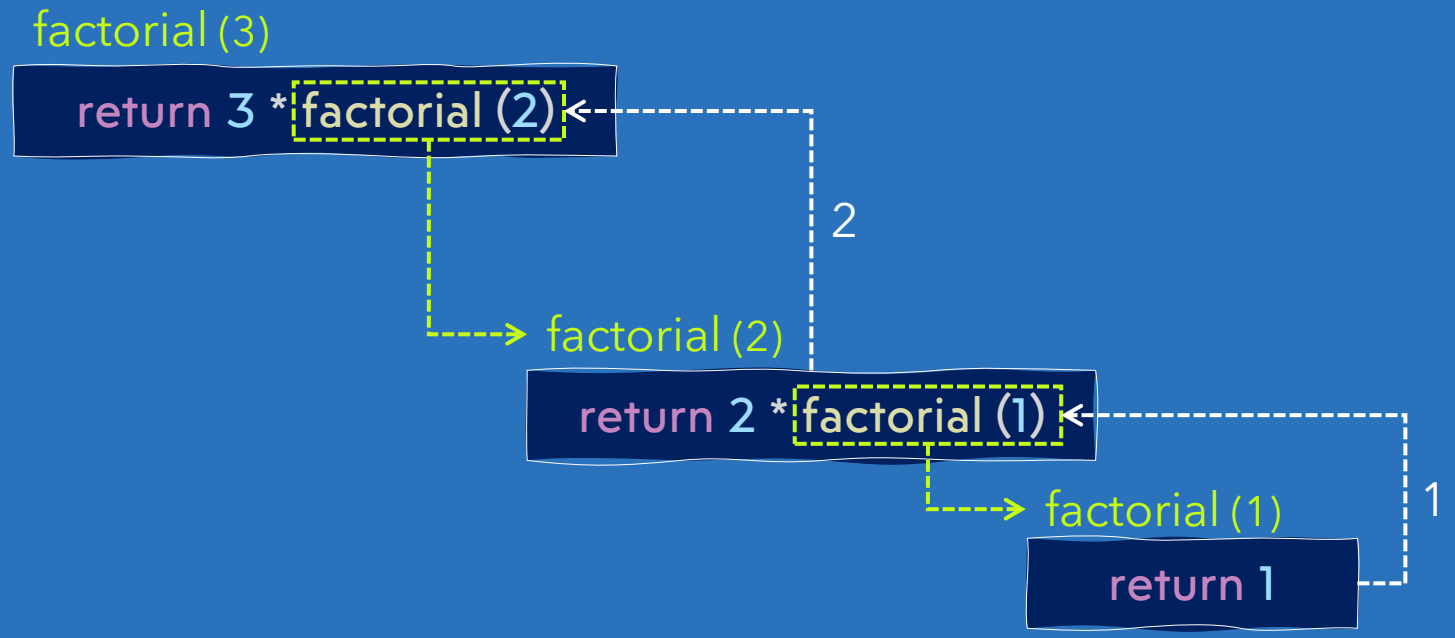
Cuando implementamos la función factorial en un lenguaje de programación utilizando recursividad, lo que estamos haciendo es descomponer el problema en subproblemas más pequeños.

Por ejemplo, para calcular 3!, en lugar de calcular directamente 3×2×1, podemos dividirlo en 3×2!, donde 2! es el factorial de 2. Similarmente, 2! se puede expresar como 2×1!. Llegamos a 1!, que es el caso base de la recursión y su valor es 1.

Lic. Yaps David P.

TRAZA DEL VALOR RECURSIVO

Es una representación visual del proceso mediante el cual se calcula un valor recursivo en un algoritmo. Esta traza muestra paso a paso cómo se descompone el problema en subproblemas más pequeños y cómo se resuelven estos subproblemas hasta llegar al caso base.




```
def factorial (numero):
    if (numero == 1):
        return 1
    else:
        return numero * factorial (numero-1)


print(factorial(3))
```

*factorial(3) = 3 * factorial(2) * factorial(1)*

Lic. Yaps David P.

```python
def decimalABinario(n):
    #Caso base: cuando el número es 0 o 1
    if n == 0:
        return "0"
    elif n == 1:
        return "1"
    else:
        #Llamada recursiva para el cociente de n dividido por 2
        #y concatenación del resto (n % 2)
        return decimalABinario(n // 2) + str(n % 2)


numeroDecimal = 10
binario = decimalABinario(numeroDecimal)
print(f"El número {numeroDecimal} en binario es {binario}")
```

CONVERSIÓN NUMÉRICA

El código define una función recursiva decimalABinario que convierte un número decimal a su representación binaria. La función maneja dos casos base: cuando n es 0 o 1, retornando "0" o "1" respectivamente.

En el caso recursivo, la función se llama a sí misma con el cociente de n dividido por 2 (n // 2), concatenando el residuo (n % 2) al resultado. Esta concatenación construye la cadena binaria desde el dígito menos significativo al más significativo.

IMPRIMIR ÁRBOL GENEALÓGICO

Para construir un árbol genealógico utilizando un diccionario y mostrar las dependencias de manera tabulada, podemos representar la familia Simpsons como ejemplo.

```
def mostrarArbolGenealogico(familia, persona, nivel=0):
    print("\t" * nivel + persona)
    if persona in familia:
        for hijo in familia[persona]:
            mostrarArbolGenealogico(familia, hijo, nivel + 1)

familiaSimpsons = {
    "Abraham": ["Homero", "Herbert"],
    "Mona": ["Homero", "Herbert"],
    "Homero": ["Bart", "Lisa", "Maggie"],
    "Marge": ["Bart", "Lisa", "Maggie"],
    "Bart": [],
    "Lisa": [],
    "Maggie": []
}

#Mostrar el árbol genealógico tabulado comenzando desde Abraham
print("Árbol genealógico de Abraham Simpsons:")
mostrarArbolGenealogico(familiaSimpsons, "Abraham")
```

Lic. Yaps David P.

```python
def organigrama(diccionario, nivel=0):
    for nombre, detalles in diccionario.items():
        print("\t" * nivel + f"{nombre} ({detalles['puesto']})")
        if detalles['dependencias'] != {}:
            organigrama(detalles['dependencias'], nivel + 1)


estructuraEmpresa = {
    "Oliver": {
        "puesto": "CEO",
        "dependencias": {
            "Gabriela": {
                "puesto": "Directora de Finanzas",
                "dependencias": {
                    "Juan": {
                        "puesto": "Analista Financiero",
                        "dependencias": {}
                    }
                }
            },
            "Irina": {
                "puesto": "Directora de Marketing",
                "dependencias": {}
            }
        }
    }
}
```

IMPRIMIR ORGANIGRAMA

Este código define una función recursiva que toma un diccionario que representa la estructura jerárquica de la empresa y un nivel de indentación. La función recorre el diccionario e imprime los nombres y los puestos de las personas, tabulando las dependencias adecuadamente.

LÍMITE DEL CÁLCULO RECURSIVO

Cada llamada recursiva agrega un marco de pila (que contiene su contexto de ejecución) a la pila de llamadas hasta que llegamos al caso base. Luego, la pila comienza a desenrollarse a medida que cada llamada devuelve sus resultados.

Python no tiene soporte para la eliminación de llamadas de cola. Como resultado, puede causar un desbordamiento de pila si se realizan más llamadas recursivas de las que permite el límite de recursión predeterminado.

Al cambiar el límite de recursión, podemos manejar casos donde se requiere una profundidad de recursión mayor, aunque siempre hay que tener cuidado con el uso de la memoria y el riesgo de desbordamiento de pila.

```python
import sys

#Mostrar el límite actual de recursión (1000)
print("Límite actual de recursión:", sys.getrecursionlimit())

#Cambiar el límite de recursión a 100
sys.setrecursionlimit(100)
print("Nuevo límite de recursión:", sys.getrecursionlimit())

#Función recursiva para calcular la factorial
def factorial (numero):
    if (numero == 1):
        return 1
    else:
        return numero * factorial (numero-1)


#Calcular la factorial de un número grande
n = 111
try:
    print(f"Factorial de {n} es {factorial(n)}")
except RecursionError:
    print(f"No se puede calcular la factorial de {n}.")
```

```python
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print("Fibonacci de 4: ", fibonacci(4))
```

SERIE DE FIBONACCI

Un ejemplo comúnmente utilizado para demostrar el poder de la recursividad es la serie de Fibonacci. Esta serie se define sumando los dos números anteriores para obtener el siguiente número, comenzando con 0 y 1 como los dos primeros elementos.

Por ejemplo, podemos calcular el cuarto elemento de la serie, que sería la secuencia 0, 1, 1, 2, 3, donde cada número es la suma de los dos anteriores. En este caso, el cuarto elemento sería 3.

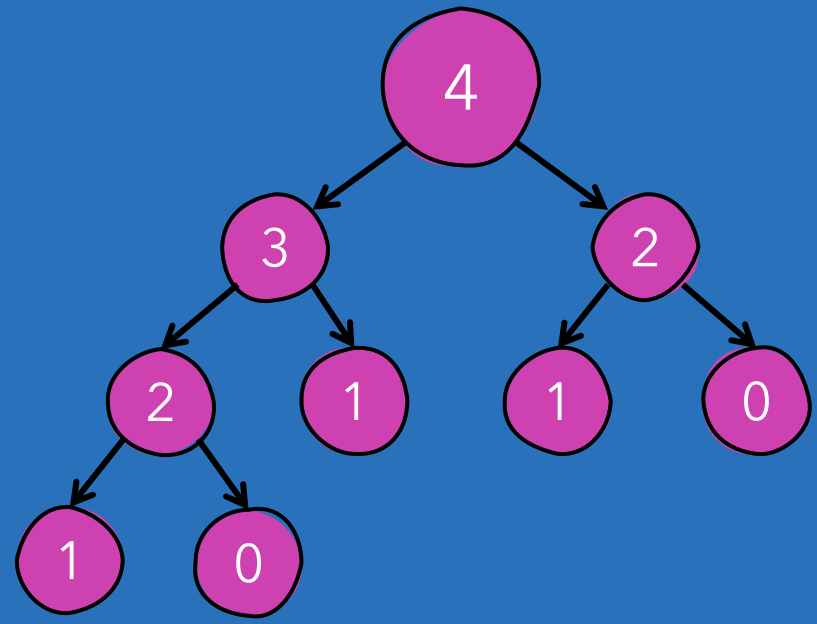
SERIE DE FIBONACCI

Nuestro código puede parecer pequeño cuando aplicamos recursividad.

Cada cálculo o llamado a esta función genera nuevos bloques en la pila de ejecuciones que se debe hacer y esto afecta a la memoria y puede afectar el rendimiento de nuestra aplicación.

Llamado	Total
fibonacci(4)	1
fibonacci(3)	1
fibonacci(2)	2
fibonacci(1)	3
fibonacci(0)	2

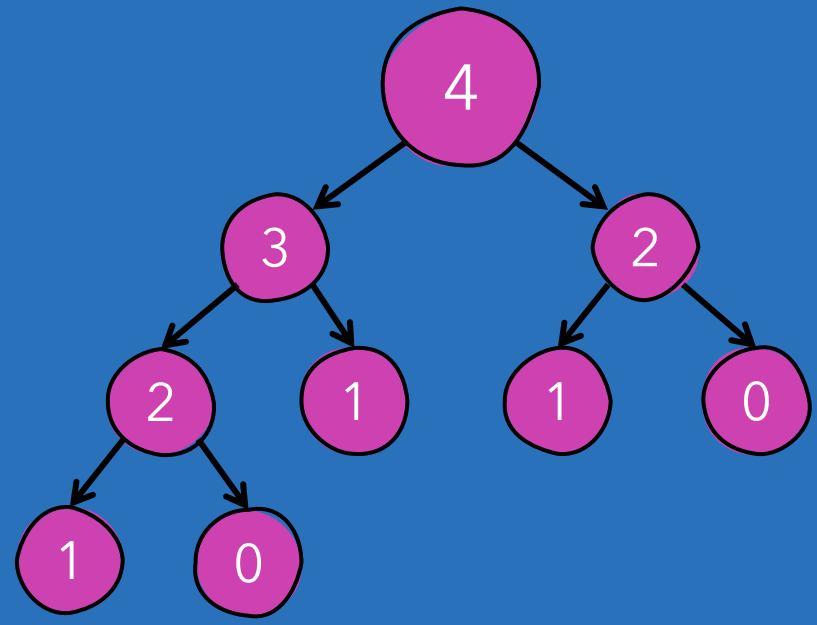
fibonacci(4)



SERIE DE FIBONACCI

Nuestro código puede parecer pequeño cuando aplicamos recursividad. Pero, así estemos llamando a la misma función una y otra vez recursivamente, cada cálculo o llamado a esta función genera nuevos bloques en la pila de ejecuciones que se debe hacer y esto afecta a la memoria y puede afectar el rendimiento de nuestra aplicación.

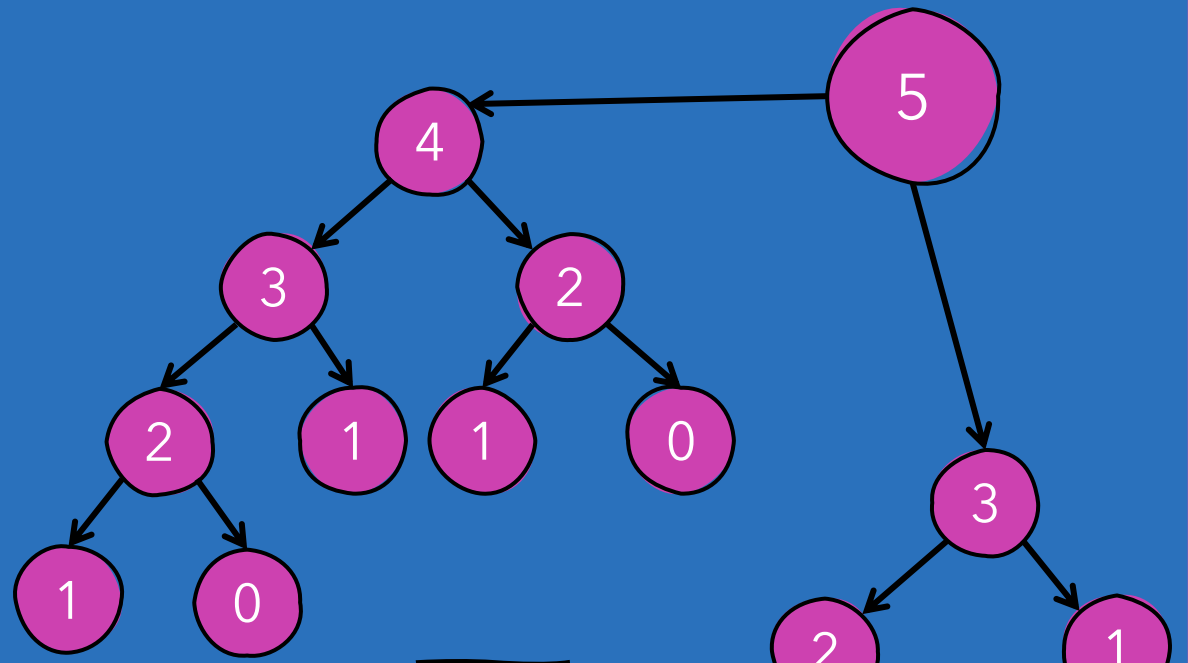
La memoria no es infinita, existe un máximo de funciones y cálculos que podemos hacer. Incluso, si no la usamos toda, gastarla excesivamente causará que nuestras aplicaciones se ejecuten lento y brinden una muy mala experiencia a los usuarios.



SERIE DE FIBONACCI

El tiempo de ejecución aumentará de manera exponencial con respecto al tamaño de la entrada.

Para calcular el n-ésimo número de Fibonacci, la función calcula los números de Fibonacci para n-1 y n-2; por lo que cada número de Fibonacci se calcula varias veces y hay una gran cantidad de llamadas recursivas que se realizan. Esto hace que la función sea muy ineficiente para valores grandes de n.



Llamado	Total
fibonacci(5)	1
fibonacci(4)	1
fibonacci(3)	2
fibonacci(2)	3
fibonacci(1)	5
fibonacci(0)	3

fibonacci(5)

```python
fibonacciCache = {}

def fibonacci(n):
    #Verificamos si tenemos el resultado almacenado
    if n in fibonacciCache:
        return fibonacciCache[n]

    if n == 0 or n == 1:
        resultado = n
    else:
        resultado = fibonacci(n-1) + fibonacci(n-2)

    #Almacenamos el resultado para futuras referencias
    fibonacciCache[n] = resultado
    return resultado
```

MEMOIZATION

Es una técnica de optimización en la cual se almacenan los resultados de funciones que requieren cálculos muy costosos para evitar volver a calcularlos.

La primera vez que se llama a una función con un conjunto de argumentos dado, se calcula el resultado y se guarda en una estructura auxiliar (caché). Por tanto, la próxima vez que se llama a la función con el mismo conjunto de argumentos, se devuelve el resultado guardado en la caché en lugar de volver a calcularlo.

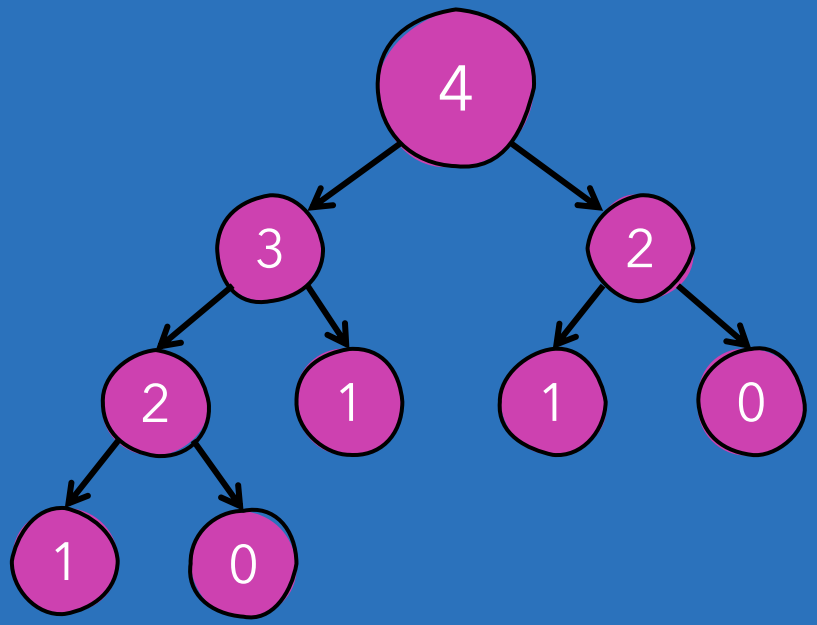
Solo debemos implementar *memoización* en funciones puras, es decir, funciones que siempre devuelven el mismo resultado cuando enviamos los mismos argumentos.

En el ejemplo, implementaremos la técnica utilizando un diccionario como estructura auxiliar.

Lic. Yaps David P.

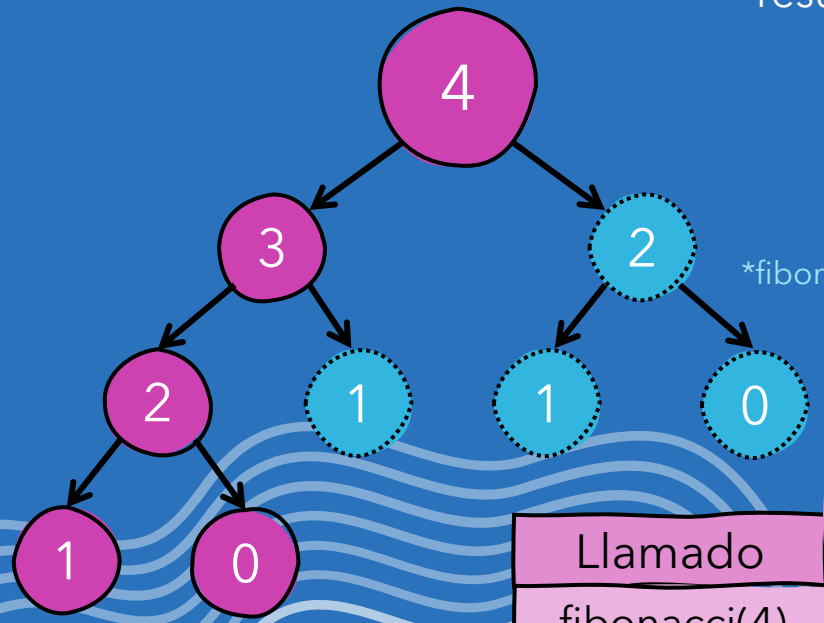
MEMOIZATION

Comparación de los algoritmos implementados con y sin cache de resultados.



Llamado	Total
fibonacci(4)	1
fibonacci(3)	1
fibonacci(2)	2
fibonacci(1)	3
fibonacci(0)	2

fibonacci(4)



*fibonacciCache

Llamado	Total
fibonacci(4)	1
fibonacci(3)	1
fibonacci(2)	1
fibonacci(1)	1
fibonacci(0)	1

fibonacci(4)

Lic. Yaps David P.

AUTOEVALUACIÓN

La autoevaluación es crucial para identificar mejoras y fortalezas de forma objetiva, facilitando la fijación de metas y la toma de acciones para alcanzarlas.