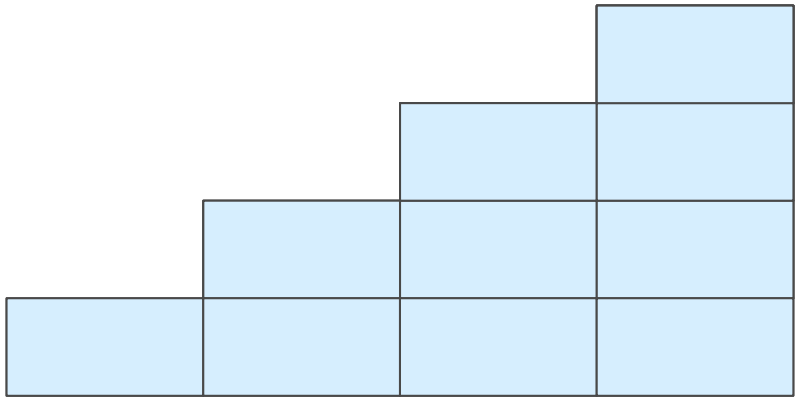
Joaquín Badillo

¿Repetición sin ciclos?

Recursión



Definición

Recursivo [adjetivo]:
Ver definición de ***Recursivo***

¡Algo recursivo es algo que se *contiene* a sí mismo!

En las ciencias computacionales usamos la recursión para resolver problemas *complejos* haciendo referencia al mismo problema pero con menor *complejidad*

Joaquín Badillo

Idea Principal

Imagina que te pido calcular a mano 2^{16}
Suena tedioso ¿no?

Pero ahora te digo, no te preocupes yo ya resolví un caso más sencillo: sé que $2^{15} = 32768$

La operación que te pedí inicialmente (2^{16}) ya es tan sencilla como multiplicar el resultado que te di (2^{15}) por 2

$$2^{16} = 2^{15} \cdot 2$$
$$= 32768 \cdot 2$$
$$= 65536$$

Joaquín Badillo

Idea Principal

¡Observa que podemos definir la operación de exponenciar (con números enteros) usando la exponenciación como parte de la definición!

$2^n = 2^{n-1} \cdot 2$

Solo que yo podría seguir con este proceso de forma indefinida:

$2^3 = 2^2 \cdot 2$

$2^2 = 2^1 \cdot 2$

$2^1 = 2^0 \cdot 2$

$2^0 = 2^{-1} \cdot 2$

$2^{-1} = 2^{-2} \cdot 2$

AHHHHHH
¿Cuándo me detengo?

Idea Principal

Ustedes seguro verán la operación 2^1 como una operación "obvia"

$2^1 = 2$

A esta operación trivial la llamamos el ***caso base***. Una definición recursiva incluye tanto un procedimiento como un caso base (formalmente incluye una propiedad de *terminación* también).

El caso base lo podemos ver como la entrada más sencilla posible (nótese que también podríamos usar $2^0 = 1$ como caso base).

Idea Principal

¿Cómo se vería la recursión con el caso base?

$2^3 = 2^2 \cdot 2$

$2^2 = 2^1 \cdot 2$

$2^1 = 2$ (Caso Base)

Idea Principal

¿Cómo se vería la recursión con el caso base?

$2^3 = 2^2 \cdot 2$

$2^2 = (2) \cdot 2$

$2^1 = 2$ (Caso Base)

Joaquín Badillo

Idea Principal

¿Cómo se vería la recursión con el caso base?

$$2^3 = 2^2 \cdot 2$$
$$2^2 = (2) \cdot 2$$

Idea Principal

¿Cómo se vería la recursión con el caso base?

$2^3 = 2^2 \cdot 2$

$2^2 = 4$

Joaquín Badillo

Idea Principal

¿Cómo se vería la recursión con el caso base?

$$2^3 = (4) \cdot 2$$
$$2^2 = 4$$

Joaquín Badillo

Idea Principal

¿Cómo se vería la recursión con el caso base?

$2^3 = (4) \cdot 2$

Idea Principal

¿Cómo se vería la recursión con el caso base?

$2^3 = 8$

Idea Principal

Esta idea la podemos capturar en un programa con tan solo condiciones y funciones... nada de ciclos.

Necesitamos un condicional para checar si estamos en el caso base y necesitamos una función que representará la operación de elevar 2 a una potencia entera y positiva (llamémosla Pow2).

```
Pow2(n):
    if n == 1:
        return 2
    else:
        return 2 * Pow2(n-1)
```

Caso base

Procedimiento recursivo

Observa que nuestra función Pow2 utiliza la función Pow2; en otras palabras, Pow2 se contiene en su definición

Pregunta Detonante

¿Qué pasa con esta función si se le pide hacer esta operación con un flotante o con un número negativo?

```
Pow2(n):
    if n == 1:
        return 2
    else:
        return 2 * Pow2(n-1)
```

Joaquín Badillo

Lección: Profundidad de Recursión

La profundidad de recursión es el número de llamadas que se tienen que hacer para terminar una función recursiva. En este caso nuestra profundidad de recursión sería infinita... ¡Nunca llegamos al caso base restando 1 y repitiendo el proceso!

Una solución: levantar un error (*manejo de excepciones*)

```
Pow2(n):
    if (type(n) != int or n < 1):
        raise Exception("Error por tipo de dato")

    elif n == 1:
        return 2

    return 2 * Pow2(n-1)
```

Además del concepto de profundidad de recursión, aquí hay otro valioso aprendizaje: Cuando hagas software siempre desconfía del usuario (es decir de quien vaya usar este software). Incluso si ese usuario eres tú.

Problema

Hay otras formas de usar la recursión para resolver el problema anterior (y de hecho hacer que sea más rápido). Imagina nuevamente que queremos computar 2^{16}

Esta vez, en lugar de decirte el valor de 2^{15} te digo que $2^8 = 256$

Todavía tenemos una forma de llegar directamente a la respuesta con **una sola** operación básica de aritmética $(+, -, \times, \div)$

¿Cómo?

¡Multiplicando el número dado por sí mismo!

Problema

En general, si conoces el valor de $2^{n/2}$ puedes conocer inmediatamente el valor de 2^n, multiplicando esa cantidad por sí misma:

$$2^n = 2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}}$$

Pero si solo queremos usar exponentes enteros (ya que la exponenciación de números reales es otro monstruo), la división n/2 puede no resultar en un entero ¿qué pasa si *n* es impar?

Joaquín Badillo

Problema

Imagina que para calcular 2^{15} te doy el resultado de 2^7. Tú ahora puedes calcular 2^{15} con 2 operaciones básicas ¿cuáles?

Multiplica el número que te di por sí mismo... ahora sabes cuánto vale 2^{14}.
Por lo que mencionamos anteriormente ahora puedes multiplicar esta cantidad por 2 y obtener el resultado deseado de 2^{15}.

En general

(División entera)

$$2^n = 2^{\lfloor \frac{n}{2} \rfloor} \cdot 2^{\lfloor \frac{n}{2} \rfloor} \cdot 2 \quad \text{Si } n \text{ es impar}$$

Joaquín Badillo

Problema

Ahora tenemos una definición recursiva de 2^n para un número entero positivo

$$2^n = \begin{cases} 2 & \text{Cuando } n = 1 \\ 2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}} & \text{Cuando } n \text{ es par} \\ 2^{\lfloor \frac{n}{2} \rfloor} \cdot 2^{\lfloor \frac{n}{2} \rfloor} \cdot 2 & \text{Cuando } n \text{ es impar} \end{cases}$$

De tarea tienen que implementar esta definición recursiva en Python, la pueden llamar pow2(n) en un nuevo archivo de Python (.py) o Notebook Interactivo (.ipynb)

Para un valor muy grande del exponente, estaríamos realizando **menos** operaciones que ir multiplicando 2 · 2 · 2 ...

Joaquín Badillo

Problema

¿Tiene algo de especial la base 2? Implementa también un algoritmo recursivo para hacer exponenciación de números positivos con cualquier base. El algoritmo debe estar escrito como una función que tome 2 argumentos, la base y el exponente.

$$\text{pow}(a, n) = a^n$$

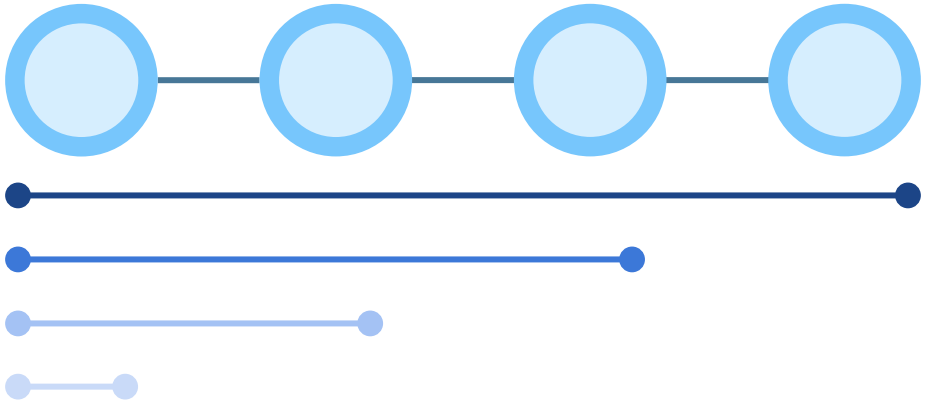
De hecho... la base 2 si tiene algo especial (pues es la que usa una computadora internamente). Exponenciar naturales también se puede lograr usando *bit shifting* (<<).

1 << 1 == 2 1 << 2 == 4 1 << 3 == 8 1 << 4 == 16

Si gustas puedes investigar acerca de esta operación

Joaquín Badillo

Estructuras Recursivas



Joaquín Badillo

Ejemplo de Recursividad (Listas)

Imagina que quieres aprender acerca de listas, entonces te acercas con alguien de Ciencias Computacionales y le preguntas ¿qué es una lista?

Esta persona te responde que una lista almacena datos, tiene un tamaño y por lo general le puedes agregar más contenido en un programa. Es una definición un poco simplificada, pero ilustra bastantes ideas.

Le quieres sacar un poco más de información y como te dijo que las listas tienen un tamaño, decides preguntar

Joaquín Badillo

Ejemplo de Recursividad (Listas)

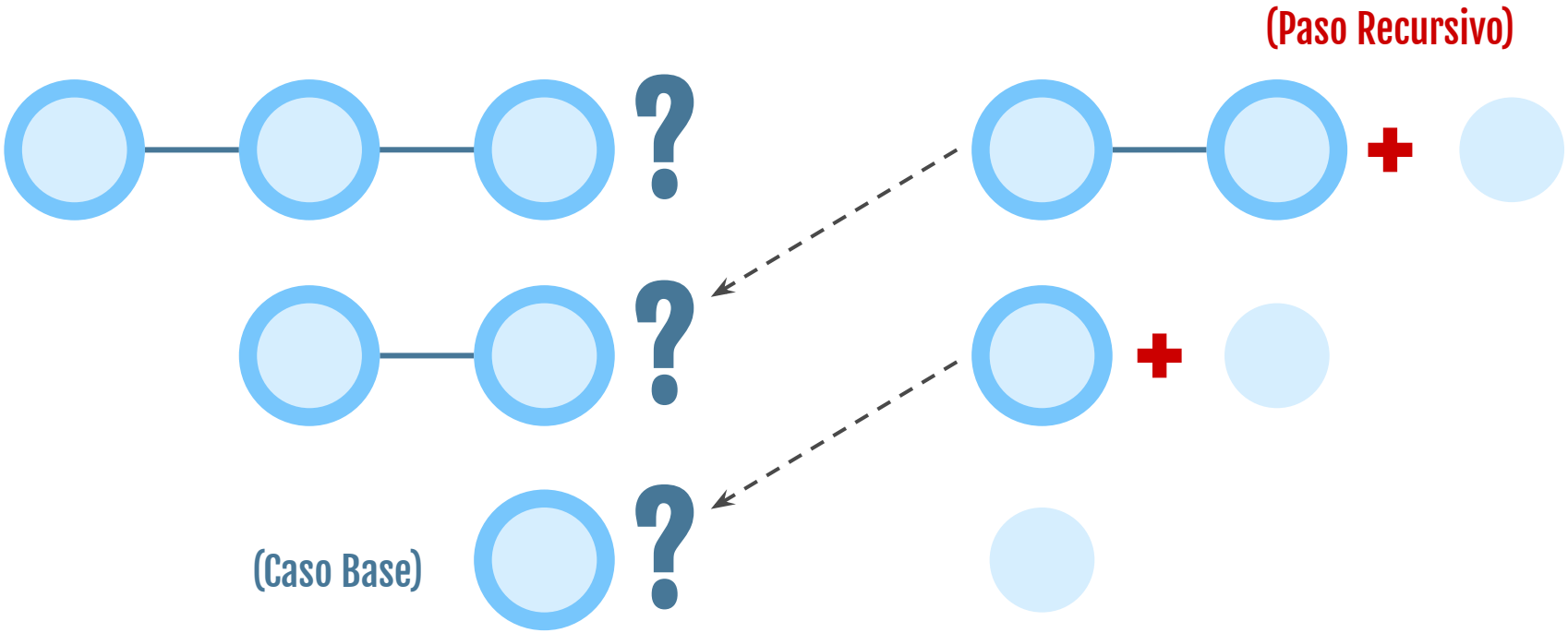
- **¿Qué es una lista de tamaño 3?**
  - Una lista de tamaño 3 es una lista de tamaño 2 a la que se le agregó un elemento

- **¿Qué es una lista de tamaño 2?**
  - Una lista de tamaño 2 es una lista de tamaño 1 a la que se le agregó un elemento

- **¿Qué es una lista de tamaño 1?**
  - Ahh, fácil, solo es un dato

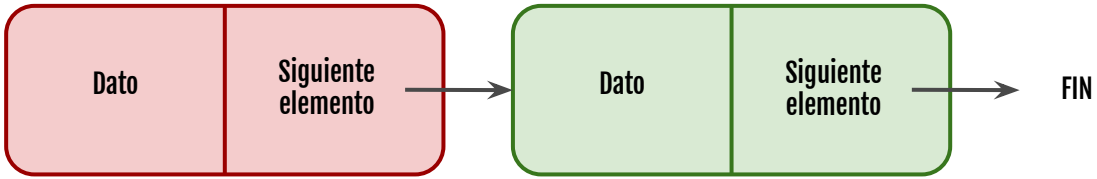
Joaquín Badillo

Lista Recursiva



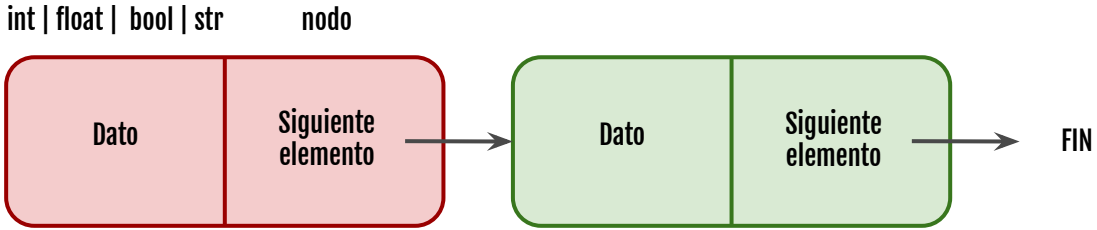
Listas

Por el momento parece innecesario ver una lista como una estructura recursiva y podrías pensar que esta persona solo estaba evadiendo tu pregunta. Pero en realidad hay una gran razón para pensar en las listas de estas manera... Gran parte de lo que hace a una lista (ligada) especial es que son *dinámicas*: pueden crecer y decrecer. Esto es posible gracias a que cada dato almacena 2 cosas: el valor y una referencia a otro dato.



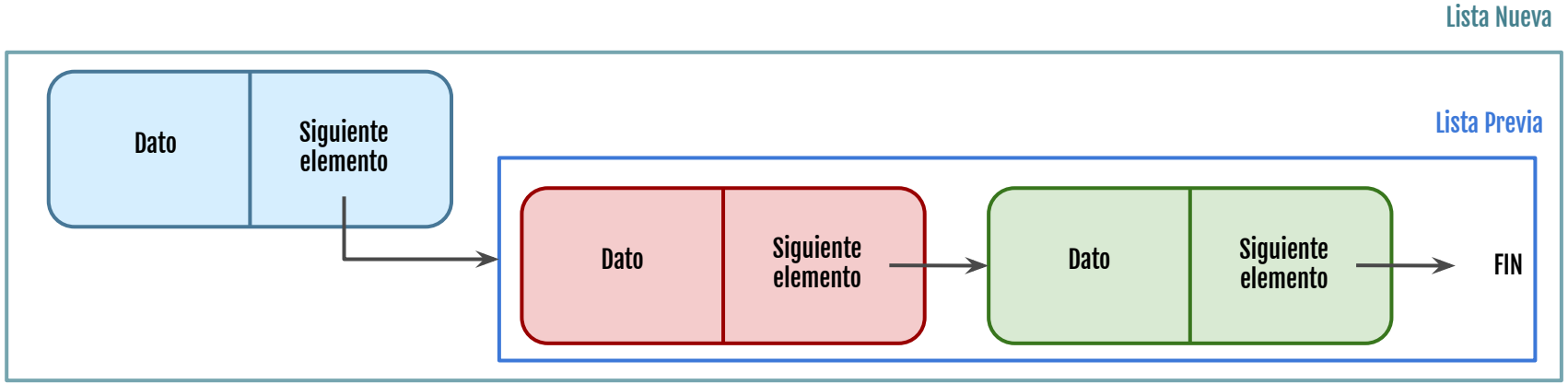
Listas

Viendo los tipos de datos que utilizan estas cápsulas (llamemoslas nodos), notamos la presencia de la recursividad: ¡un nodo contiene otro nodo! Hasta que llegamos al nodo trivial (un nodo especial que indica el fin)



Listas

En Programación Orientada a Objetos veremos como hacer inserciones en una lista. Por el momento, si pensamos recursivamente, una forma de insertar elementos es crear una lista nueva que contiene el dato a insertar y la lista previa:



También veremos posteriormente qué tiene de especial insertar al frente.

Desde la perspectiva de la computadora

Aunque en este caso la recursión no es realmente necesaria; da un nuevo marco de referencia para pensar en cómo resolver problemas computacionales.

Hay estructuras que tendrán una fuerte relación con la recursión, como los árboles. Muchos procesos sobre un árbol se deducen pensando en estos recursivamente (recorridos, tamaño, sumar los datos contenidos, etc.)

Ahora veamos un ejemplo de una estructura que se construye recursivamente.

Desde la perspectiva de la computadora

Sea X(n) una estructura desconocida y llamemos a *n* el tamaño de X.

Receta para X de tamaño n
Si el número es 1, entonces para hacer X(1) solo coloca un cuadrado.
Para hacer un X con otro tamaño *n*, primero tienes que hacer un X de tamaño *n-1. Después debes* colocar una lista de *n* cuadrados abajo

Utiliza este procedimiento para hacer X(4)
Observa que no puedes hacer la última instrucción sin antes descubrir cómo hacer la estructura con el valor *n-1*

Paso a Paso

X(4)
Para hacer un X de tamaño 4, como el número es distinto de 1, tengo que hacer X(3).

X(3)
El tamaño es distinto de 1, entonces tengo que hacer un X de tamaño 2.

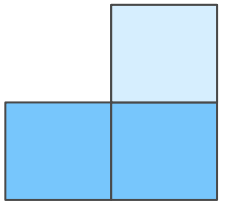
X(2)
Primero necesito hacer X(1).

X(1)
Coloca un bloque

Joaquín Badillo

Paso a Paso

X(4)
Para hacer un X de tamaño 4, como el número es distinto de 1, tengo que hacer X(3).



X(3)
El tamaño es distinto de 1, entonces tengo que hacer un X de tamaño 2.

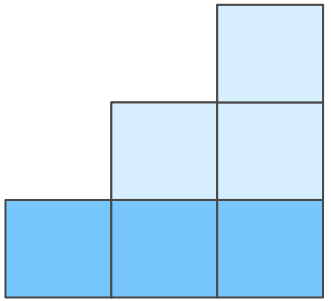
X(2)
Primero necesito hacer X(1). (Listo)
Ahora agrega una lista de 2 cuadrados abajo

Joaquín Badillo

Paso a Paso

X(4)
Para hacer un X de tamaño 4, como el número es distinto de 1, tengo que hacer X(3).

X(3)
El tamaño es distinto de 1, entonces tengo que hacer un X de tamaño 2. (Listo)
Ahora agrega una lista de 3 cuadrados abajo

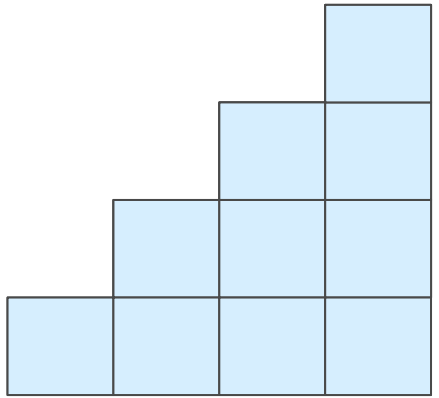


Joaquín Badillo

Paso a Paso

X(4)

Para hacer un X de tamaño 4, como el número es distinto de 1, tengo que hacer X(3). (Listo)
Ahora agrega una lista de 4 cuadrados abajo



Joaquín Badillo

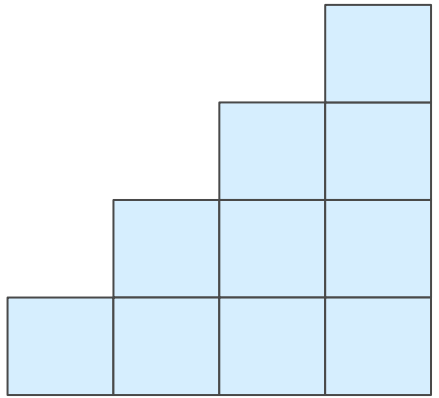
Paso a Paso

X en función de *n* es una pirámide de altura *n*

La definición recursiva resulta de las siguientes observaciones...

Caso base:
Una pirámide de altura 1 es un bloque

Paso recursivo:
Una pirámide de altura *n* es una pirámide de altura *n-1* a la que le agregas una fila de longitud *n* debajo



En otras palabras:
Podemos crear una pirámide más *compleja* (es decir de mayor altura) haciendo referencia a una pirámide más sencilla, hasta que llegamos a la pirámide *obvia*, la pirámide de altura 1.

Joaquín Badillo

Desde la perspectiva de la computadora

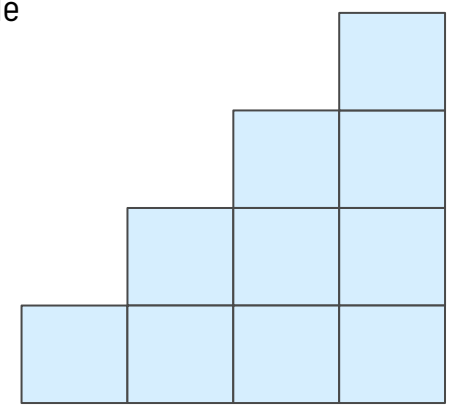
Tuvimos que hacer el mismo *procedimiento* (crear una pirámide) con valores de su altura decrecientes

n: 4 → 3 → 2 → 1

Una vez que llegamos a la pirámide trivial, la de altura 1 solo colocamos un bloque y tuvimos que regresar a terminar las operaciones que dejamos pendientes (agregar las listas de cuadrados faltantes).

n: 1 → 2 → 3 → 4

Nótese que para hacer las operaciones faltantes tuvimos que dejar anotados nuestros procedimientos (o al menos "recordarlos")



Procedimientos y Estructura:
La sucesión de Fibonacci

$$F_n = \begin{cases} 0 & \text{Cuando } n = 0 \\ 1 & \text{Cuando } n = 1 \\ F_{n-1} + F_{n-2} & \text{Cuando } n > 1 \end{cases}$$

En un lenguaje más normal:
Los primeros dos números en la sucesión de Fibonacci son el 0 y el 1.
Para obtener cualquier otro valor suma los dos anteriores

Joaquín Badillo

¿Qué conviene usar?

Ciclos contra Recursión

Joaquín Badillo

Situación Detonante: Fibonacci

```
def fib(n):
    prev = 1
    current = 0

    for i in range(n):
        temp = current
        current += prev
        prev = temp

    return current
```

(Ciclo)

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n-1) + fib(n-2)
```

(Recursión)

Joaquín Badillo

¿Por qué la solución recursiva es tan lenta?

Joaquín Badillo

Fibonacci y los árboles

Este problema es interesante porque además de usar una definición recursiva, veremos la ejecución de este programa se puede analizar o interpretar con una estructura

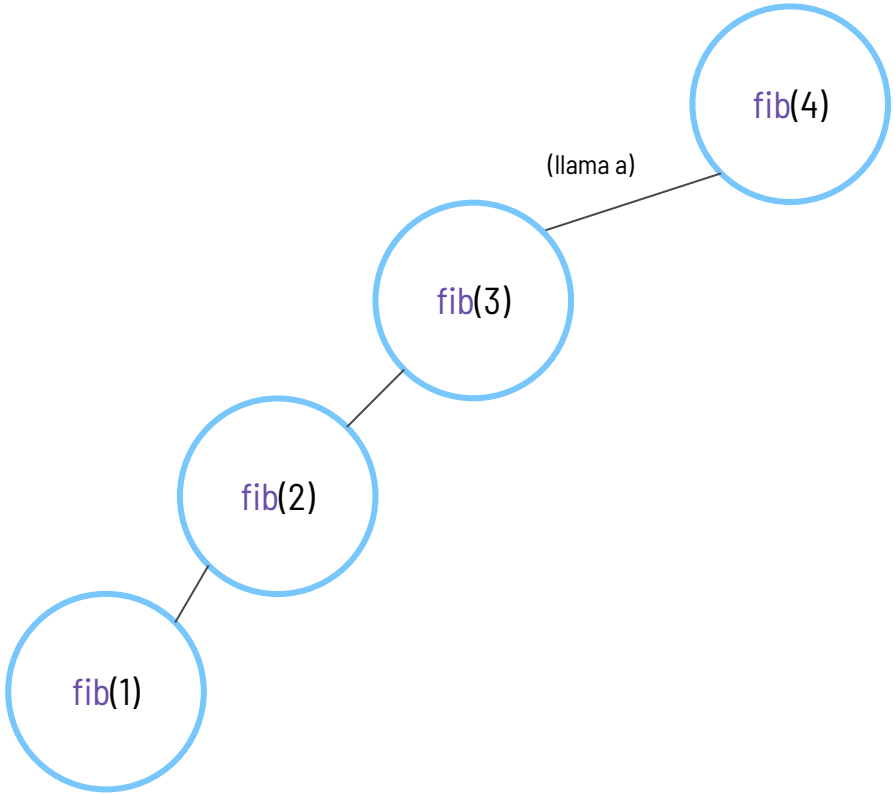
Joaquín Badillo

Llamadas

La clave para este análisis es pensar en una forma de representar cada llamada que se hace la función recursiva a si misma, para ello usaremos como ejemplo la ejecución de fib(4).

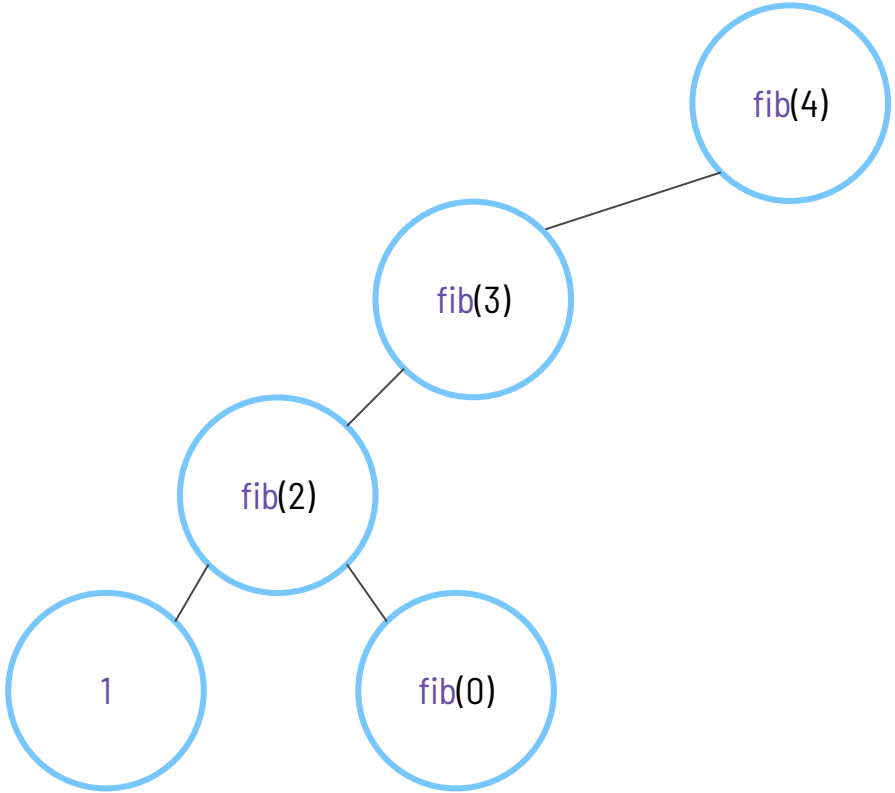
Tengan abierto el programa de fibonacci en un editor de texto o en un Notebook, porque vamos a ejecutarlo paso a paso

Procedimientos



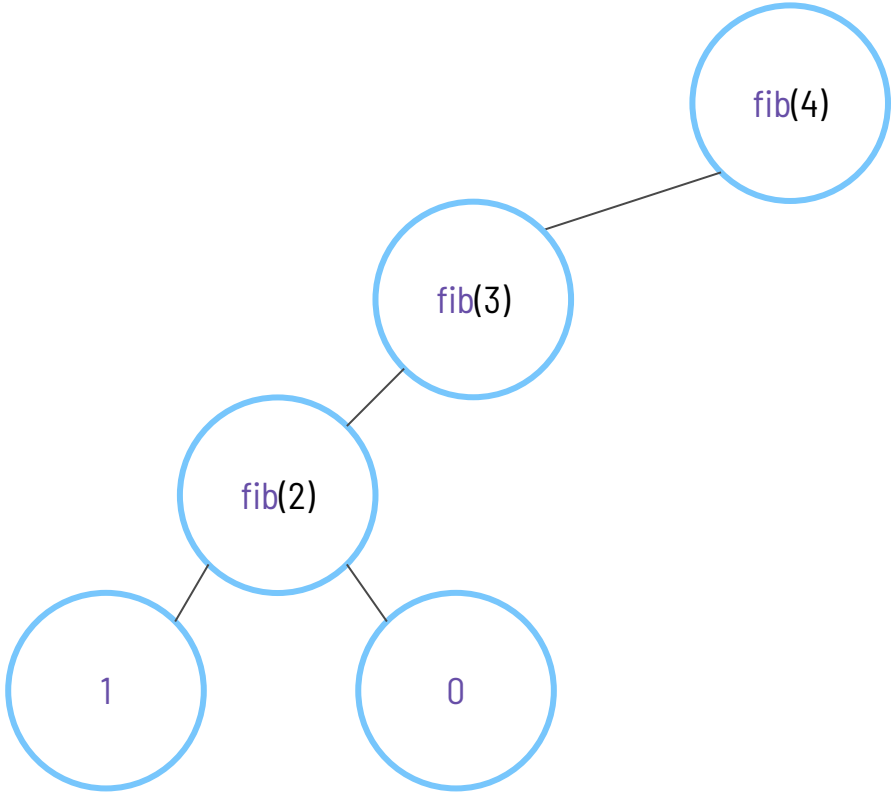
Joaquín Badillo

Procedimientos



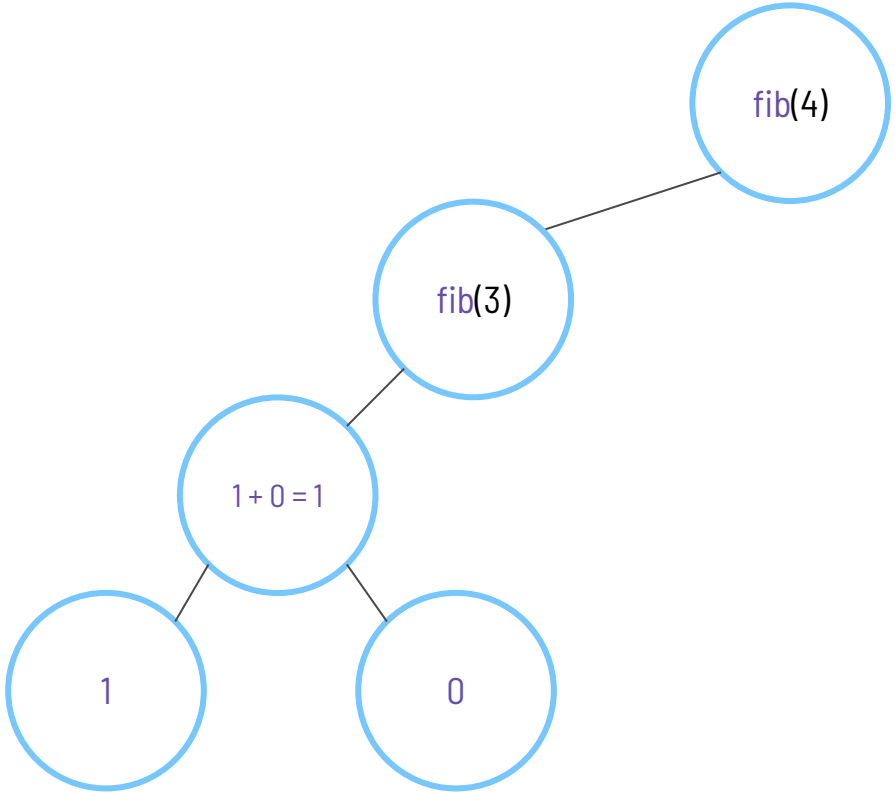
Joaquín Badillo

Procedimientos



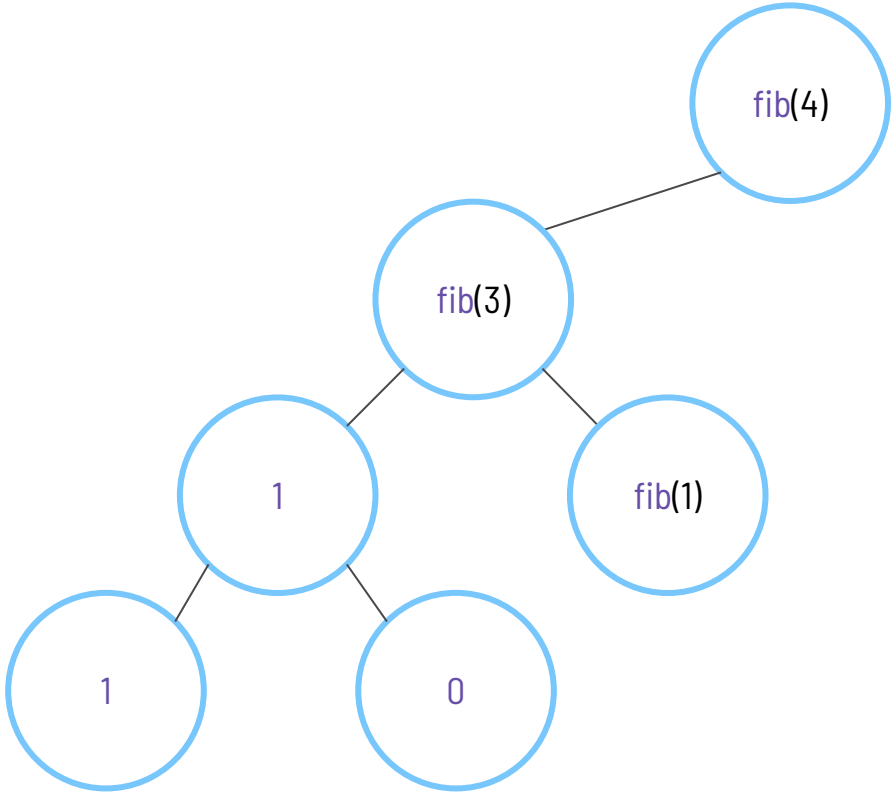
Joaquín Badillo

Procedimientos



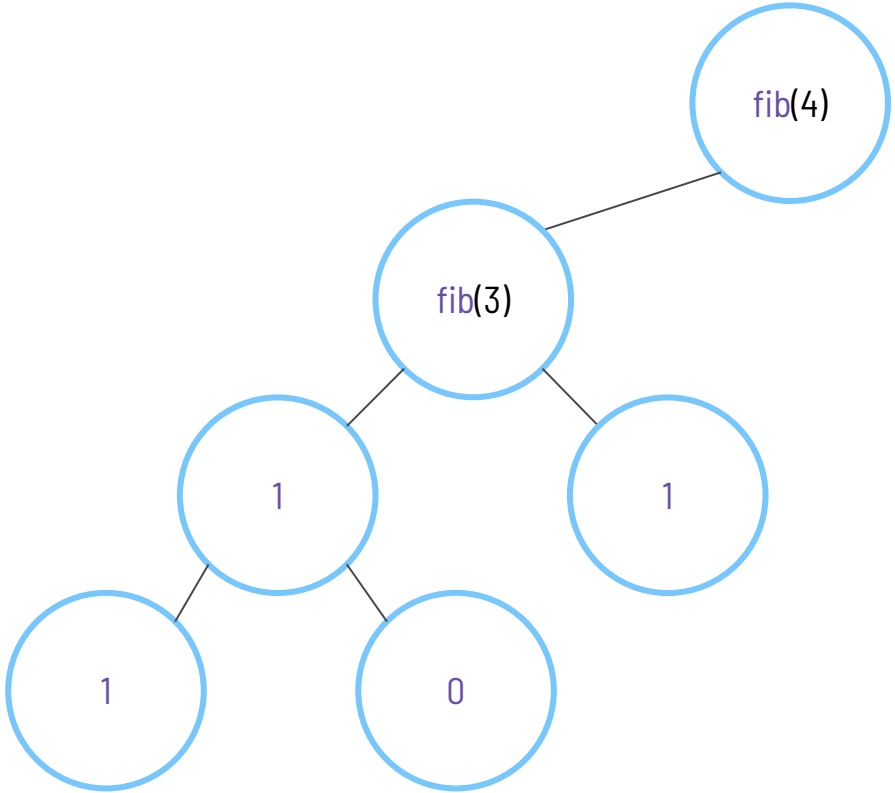
Joaquín Badillo

Procedimientos



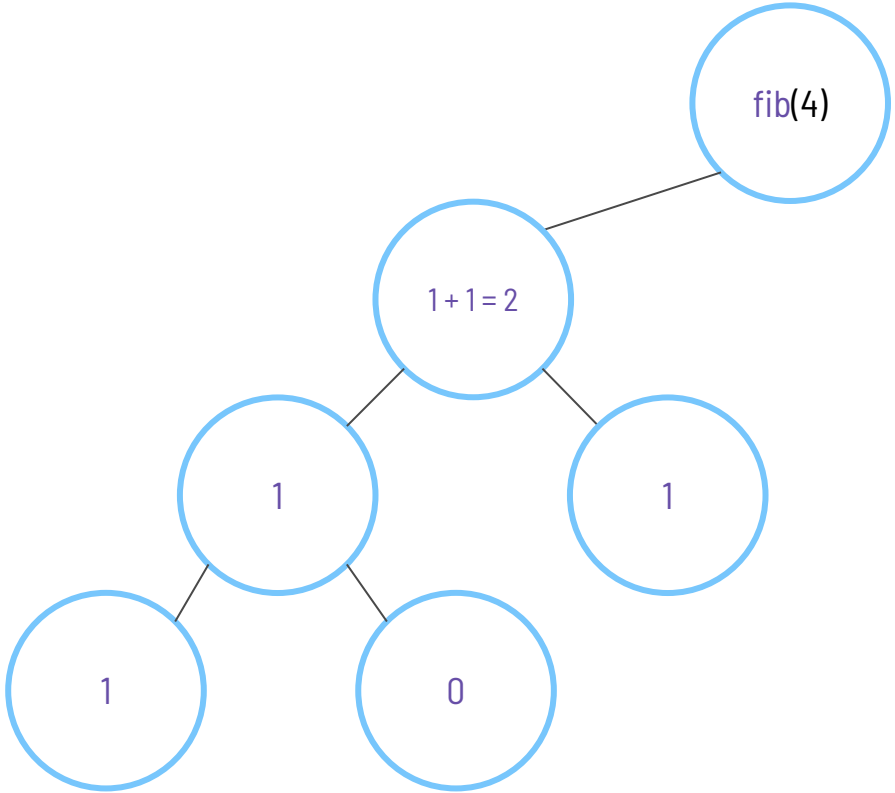
Joaquín Badillo

Procedimientos



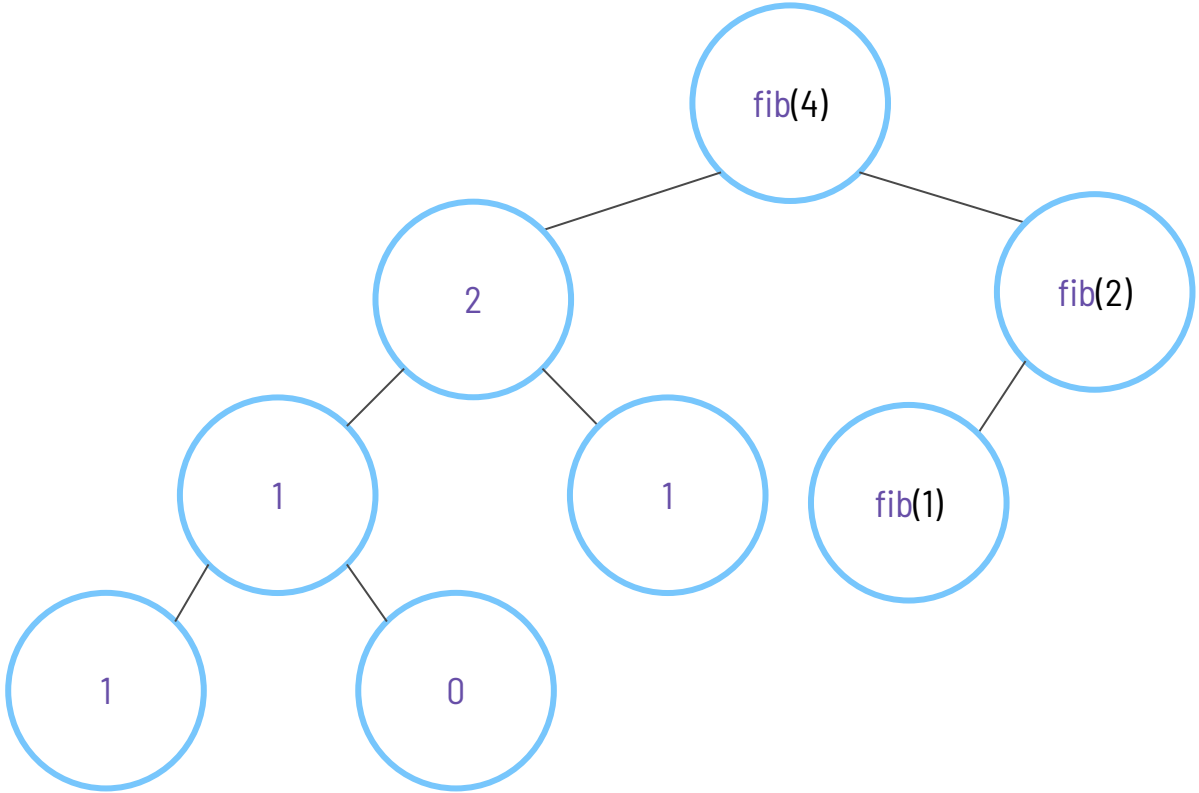
Joaquín Badillo

Procedimientos



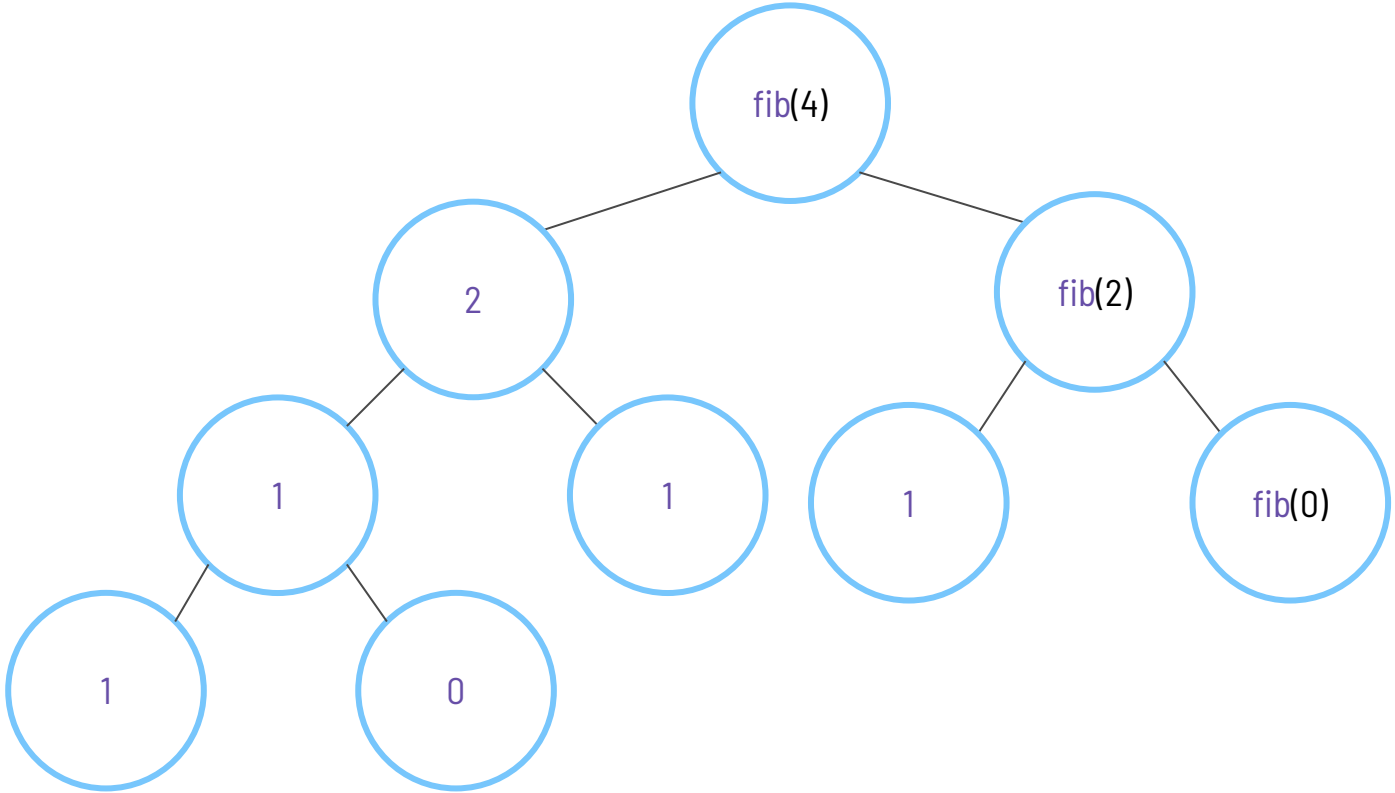
Joaquín Badillo

Procedimientos



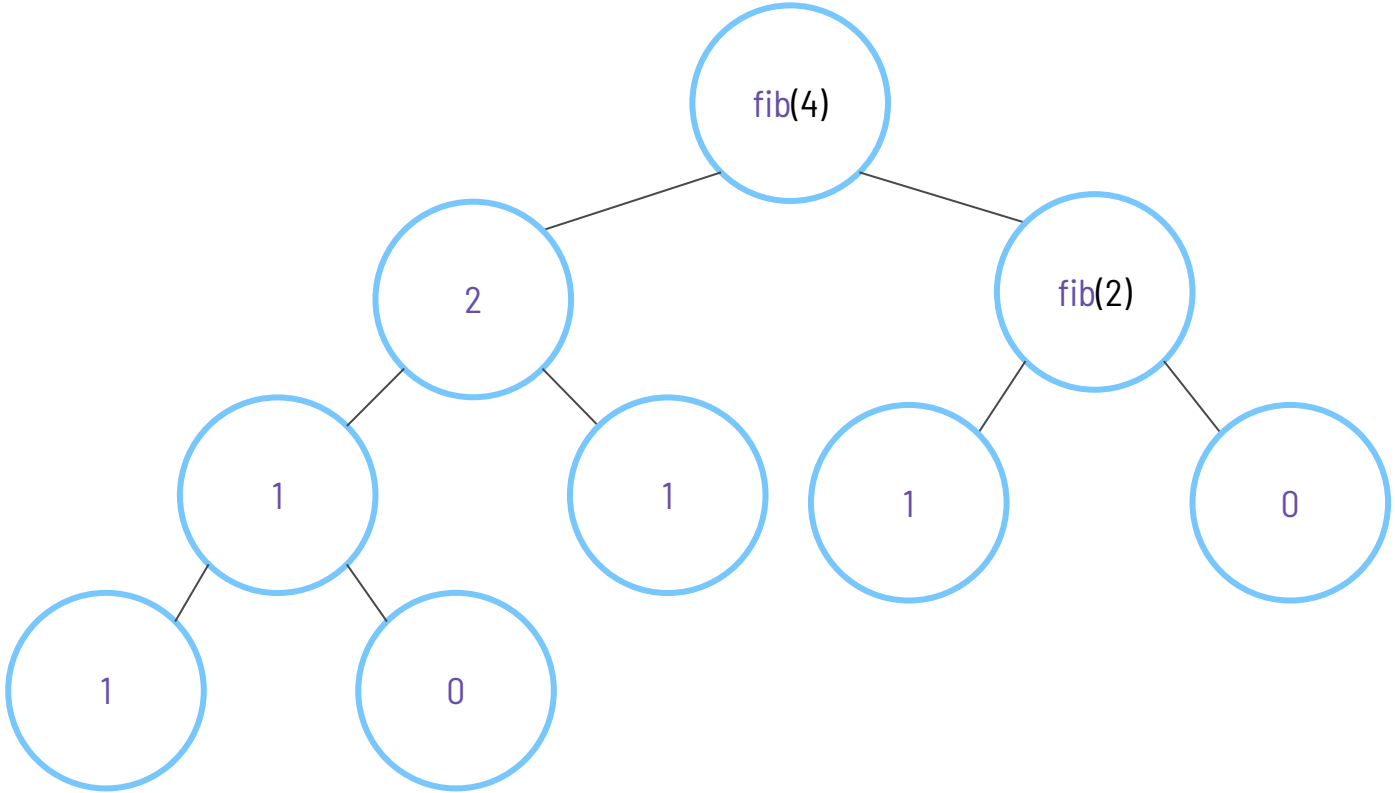
Joaquín Badillo

Procedimientos



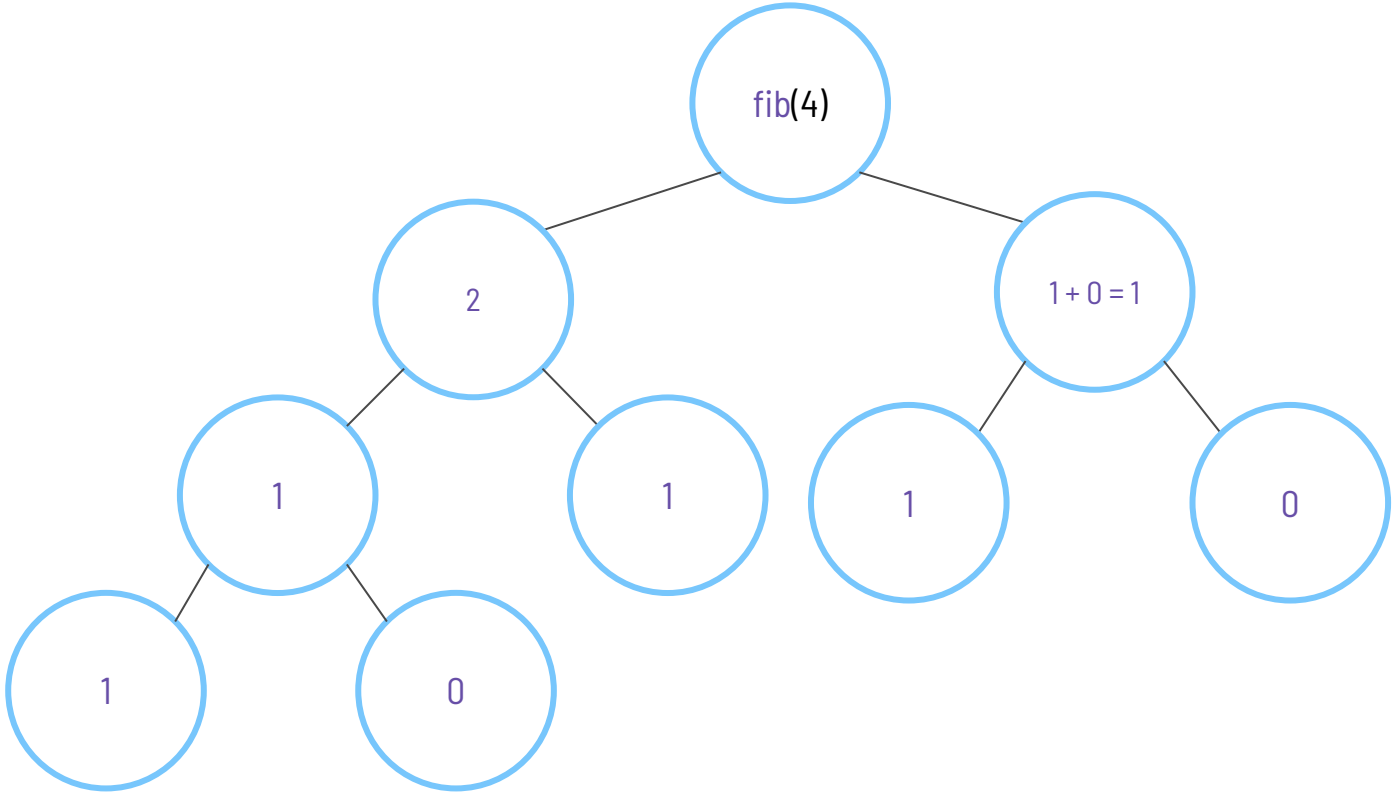
Joaquín Badillo

Procedimientos



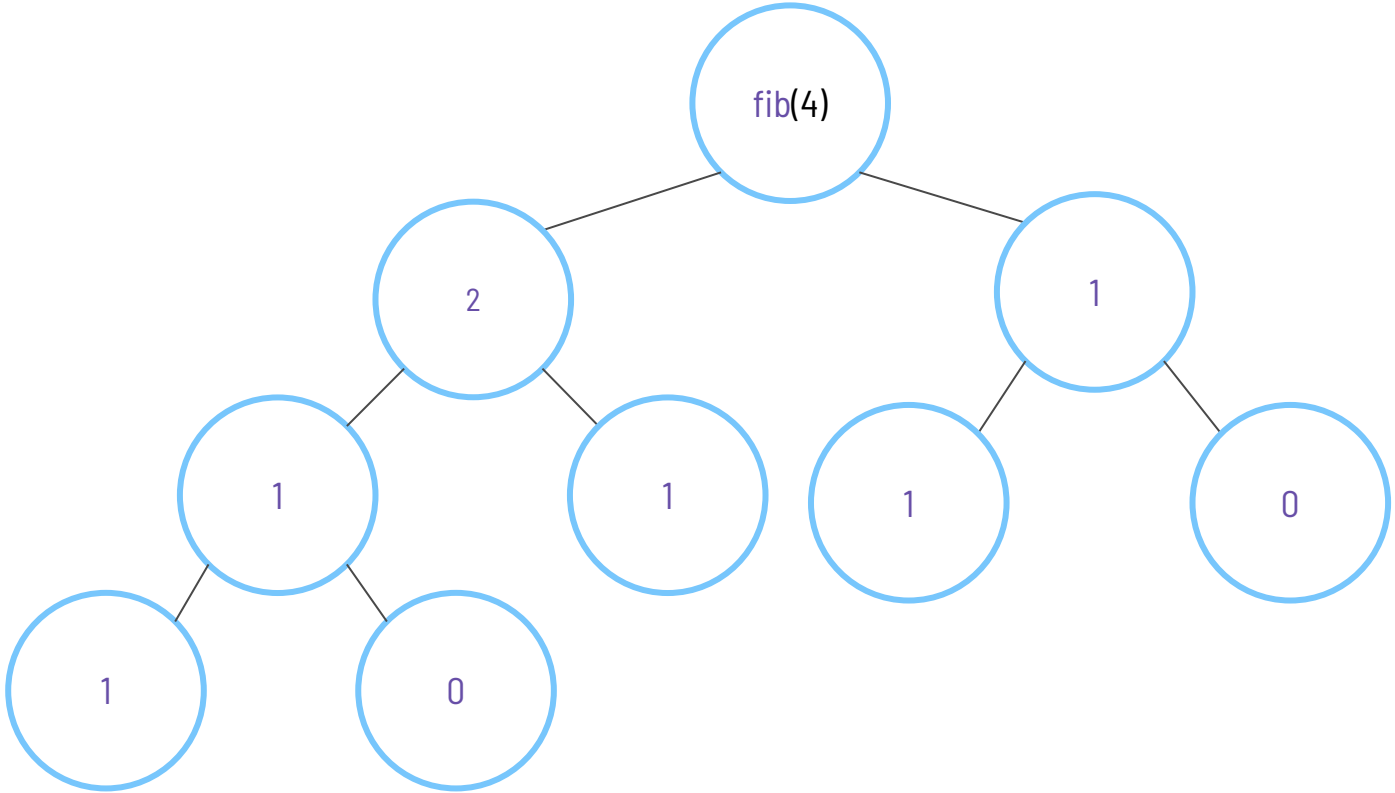
Joaquín Badillo

Procedimientos



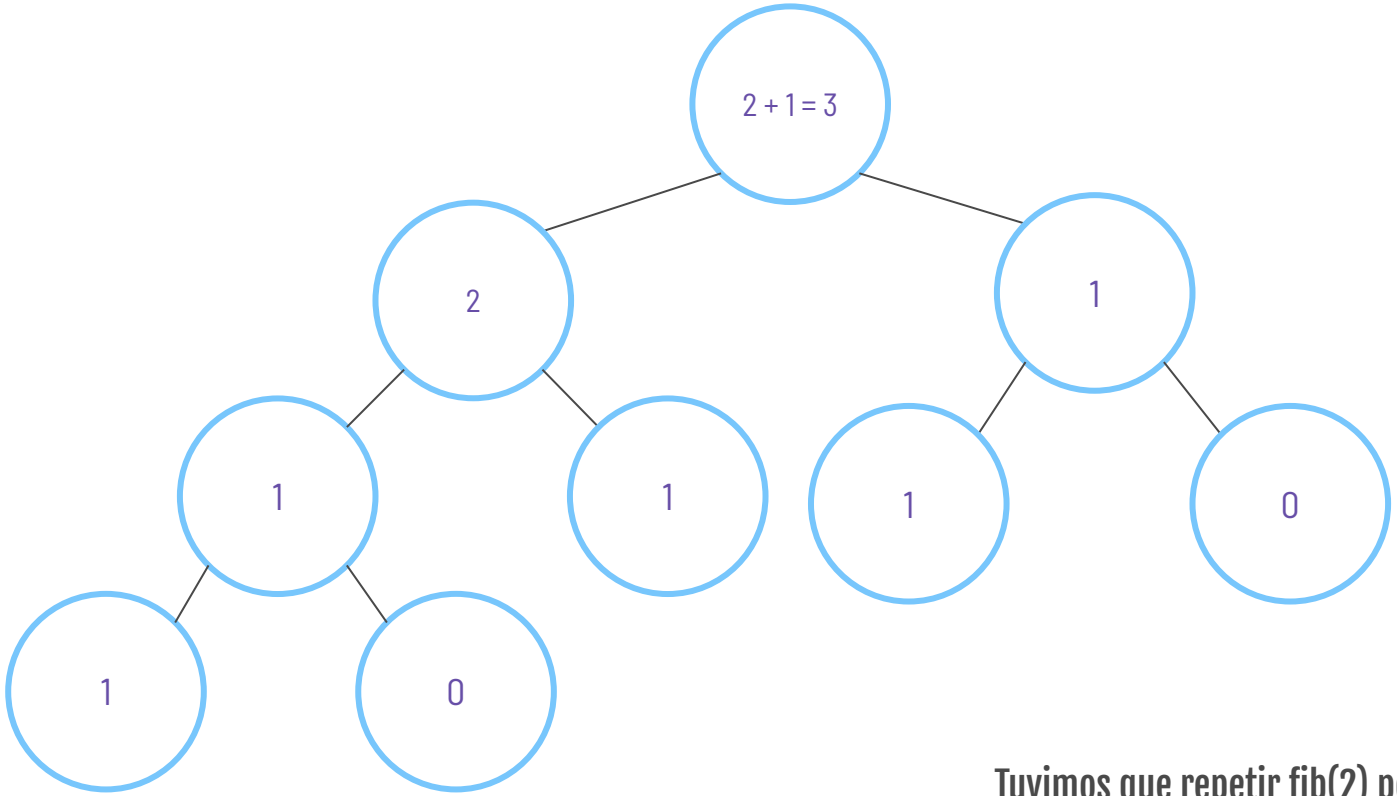
Joaquín Badillo

Procedimientos



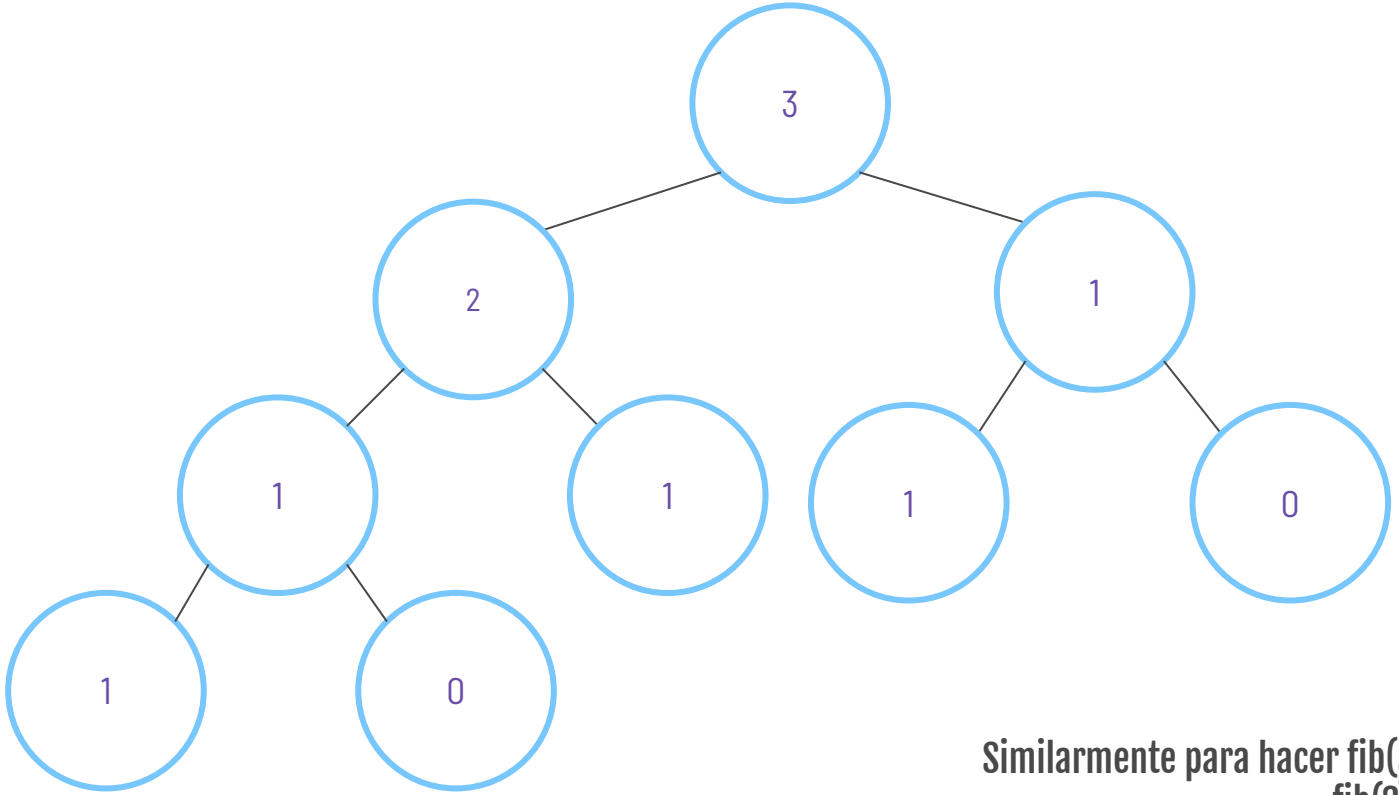
Joaquín Badillo

Procedimientos



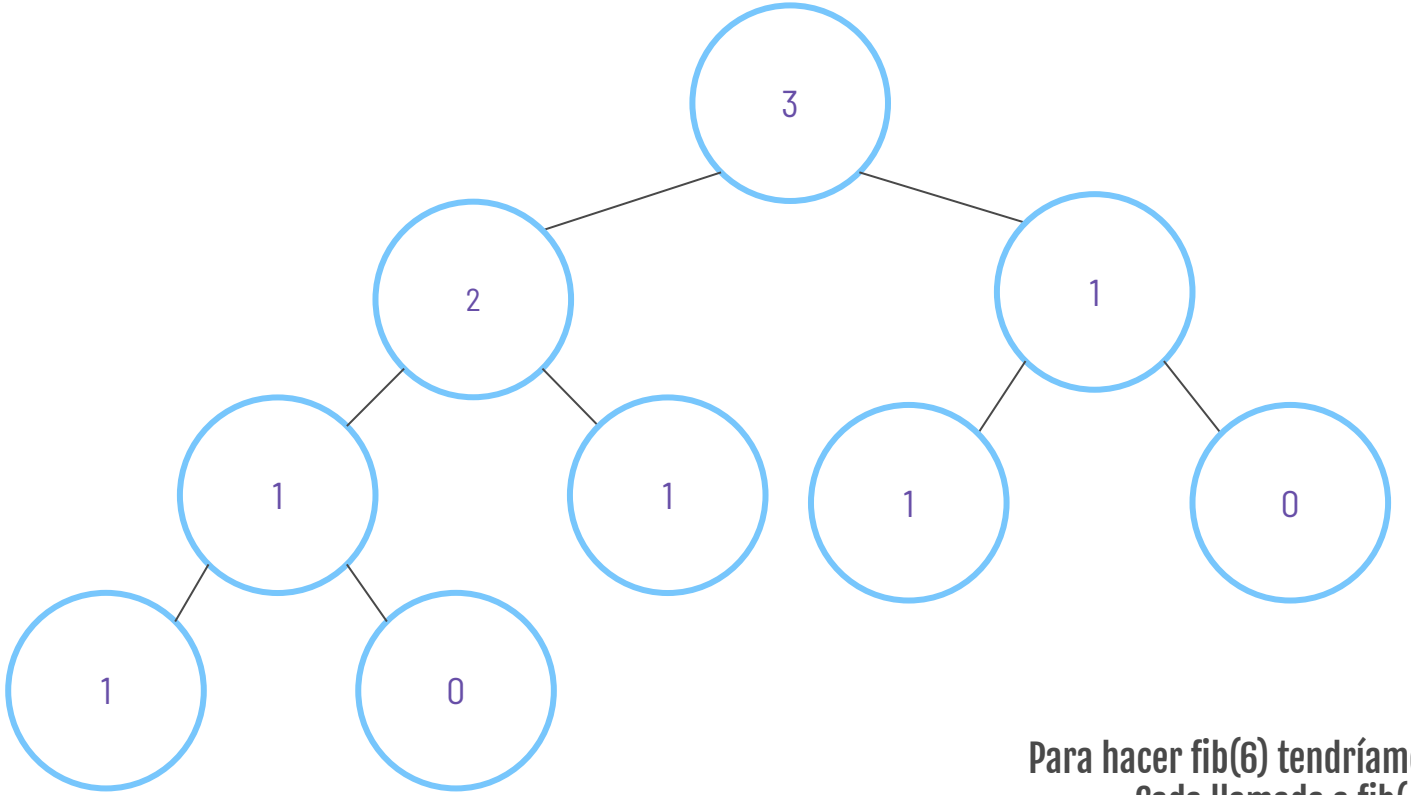
Tuvimos que repetir fib(2) porque la computadora no memorizó que fib(2) = 1

Procedimientos



Similarmente para hacer fib(5) tendríamos que repetir fib(3).

Procedimientos



Para hacer fib(6) tendríamos que repetir fib(4)...
Cada llamada a fib(4) repite fib(3)

Optimizaciones

```
memo = dict()

def fib(n):
    if n <= 1:
        return n

    elif n not in memo:
        memo[n] = fib(n-1) + fib(n-2)

    return memo[n]
```

(Memoización)

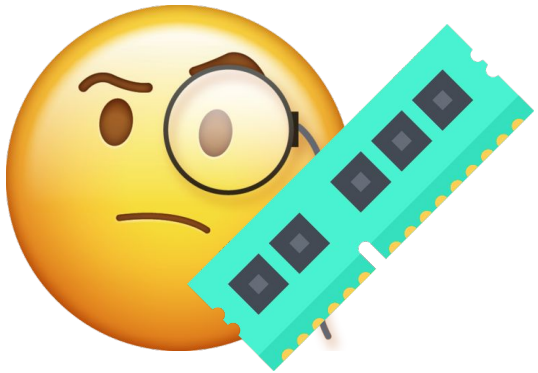
(Recursión con cola)

```
def fib(n, prev = 0, current = 1):
    if n <= 0:
        return 0

    elif n == 1:
        return current

    return fib(n-1, current, current + prev)
```

Joaquín Badillo

Pregunta detonante:

¿Qué están haciendo mejor estas funciones recursivas?



Discusión

La memoria de la computadora es un recurso limitado

Estamos usando memoria para mejorar la recursión

¿Cuál es el beneficio?

Conclusiones

Joaquín Badillo

Hay procedimientos (o bien *algoritmos*) que se definen naturalmente con recursión (recorrido de un árbol, *mergesort*, mcd, ...)

Por otra parte, pensar recursivamente nos permite llegar a nuevas ideas para resolver un mismo problema ¿Cómo hicieron la función de potencia con ciclos? Probablemente multiplicaron *n* veces y se acabó. Pero con **recursión** hacen menos multiplicaciones en total (pueden usar *print* cada que hagan una multiplicación en su función y verificarlo con números grandes del exponente)

Lo que hicimos con la exponenciación fue aplicar un patrón para resolver problemas conocido como *Dividir y Vencer* (también llamado *Divide y Vencerás*)

Atribución

Ilustraciones de *Stories*
Plantilla inspirada en *Technology Consulting* de *Slidesgo*