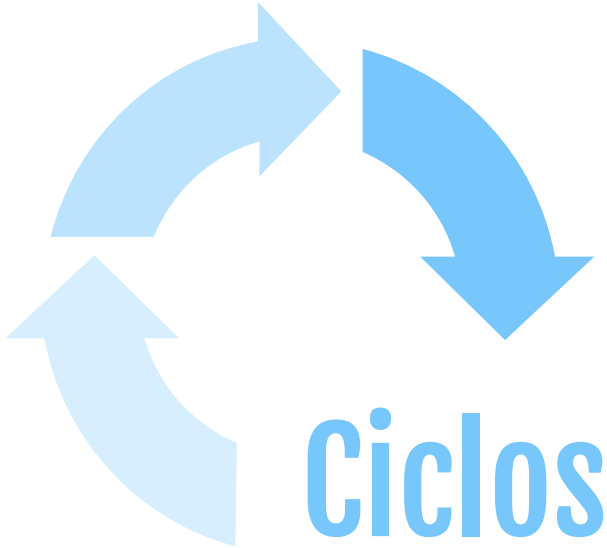


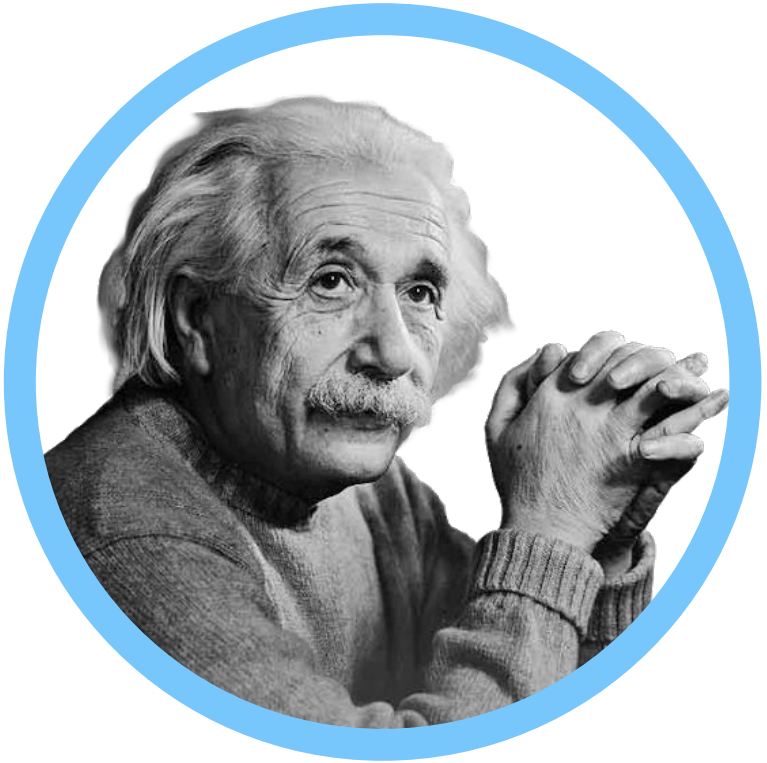
Ciclos

¿Repetir algo y obtener resultados distintos?

Joaquín Badillo

Locura es hacer lo mismo una y otra vez, esperando resultados diferentes.

— Albert Einstein (probablemente)



Joaquín Badillo

La idea principal

El día de hoy vamos a ver cómo hacer que una computadora pueda realizar un procedimiento de forma repetida.

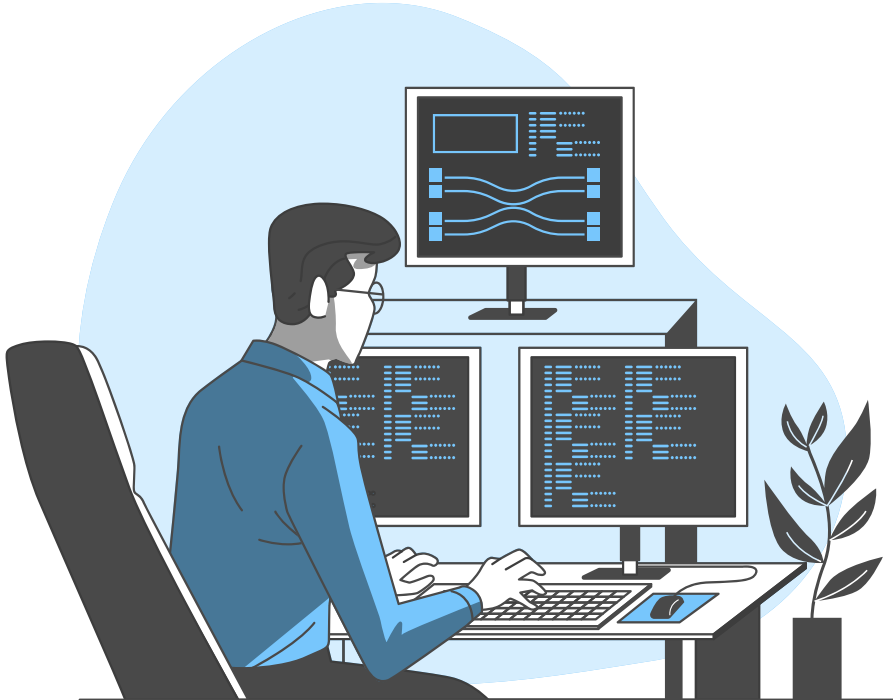
Pero repetir un *procedimiento* no es lo mismo que hacer exactamente la misma acción una cantidad arbitraria de veces.

Por ejemplo hacer las sumas:
1 + 1
1 + 1 + 1
1 + 1 + 1 + 1

Son todas distintas, pero estamos aplicando el mismo procedimiento (sumar 1).

La idea principal

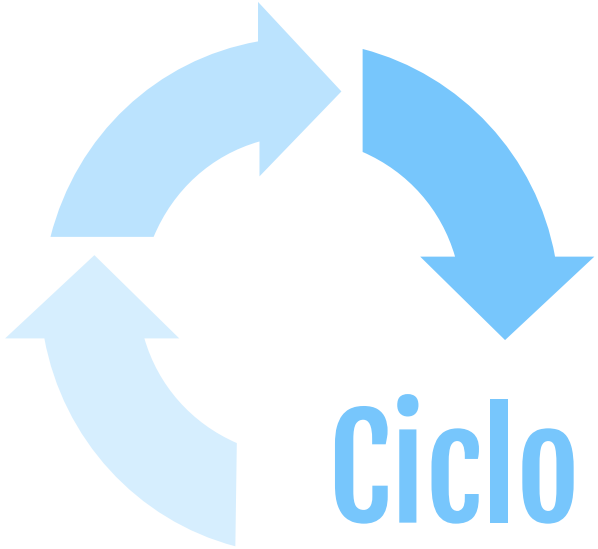
Al programar podemos apoyarnos de las *variables* para tener valores que cambian en cada *iteración*



Joaquín Badillo

También podemos repetir una acción hasta que se cumpla una condición (Por ejemplo pedir una contraseña hasta que sea la correcta)

Joaquín Badillo



Ciclo While

Joaquín Badillo

Ciclo While

Un ciclo *while* repite uno o muchos procedimientos hasta que una expresión lógica (es decir una condición) sea falsa.

En Python

```
while condicional:
        procedimiento1
        procedimiento2
              ⋮
        procedimientoN
```

En español

```
mientras se cumpla esta condición:
        realiza este procedimiento
        y este procedimiento
        etc.
```

Ejemplos en Python

```
itr = 0

while itr < 3:
    print(“Hello loops”)
    itr += 1
```

El resultado que verían sería que se imprime la frase *Hello loops* 3 veces en pantalla. Además se creó una variable, *iter*, cuyo valor al final de la ejecución es 3.

Ejemplos en Python

```
suma = 0

while suma < 10:
    suma += 1

print(suma)
```

En este código se crea una variable, *suma*, que se incrementa de 1 en 1 hasta llegar a 10, después se imprime

Ejemplos en Python

```
password = “secret!”
userPassword= input(“Enter the password: ”)

while userPassword != password:
    print(“Incorrect Password!”)
    userPassword= input(“Enter the password: ”)

print(“Welcome”)
```

Pide al usuario una contraseña y continúa pidiendo la contraseña hasta que estas coincidan. Una vez termina el ciclo se muestra un mensaje de bienvenida

Manos a la obra

Exponenciación con Números Positivos

Actividad:

Crea una función que realice la operación 2^n (para exponentes positivos y enteros), usando **únicamente** multiplicaciones. Esta función debe recibir el exponente como argumento y se llamará pow2(n).

Recuerda que:

$$2^n = \underbrace{2 \cdot 2 \cdot \ \ldots \ \cdot 2}_{n \text{ veces}}$$

Pregunta Detonante

Joaquín Badillo

Pregunta Detonante

Vamos a modificar uno de los ejemplos vistos anteriormente:

```
suma = 0
cond = suma < 10

while cond:
    suma += 1

print(suma)
```

La condición suma < 10 la almacenaré en una variable llamada *cond*.

Ahora utilizaré la variable cond como condición en el ciclo

¿Este programa funciona igual?

Joaquín Badillo

Pregunta Detonante

Este programa de hecho es inútil

```
suma = 0
cond = suma < 10

while cond:
    suma += 1
    cond = suma < 10

print(suma)
```

¡La computadora no almacena la condición como tal, únicamente su valor!

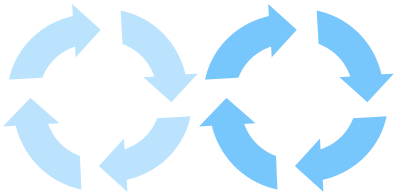
Cuando se creó la variable *cond* su valor era *Verdadero* y seguirá teniendo este valor a menos que dentro del ciclo volvamos a **evaluar** la expresión. Por ende este código crea un ciclo que nunca termina: un ciclo **infinito**.

Joaquín Badillo

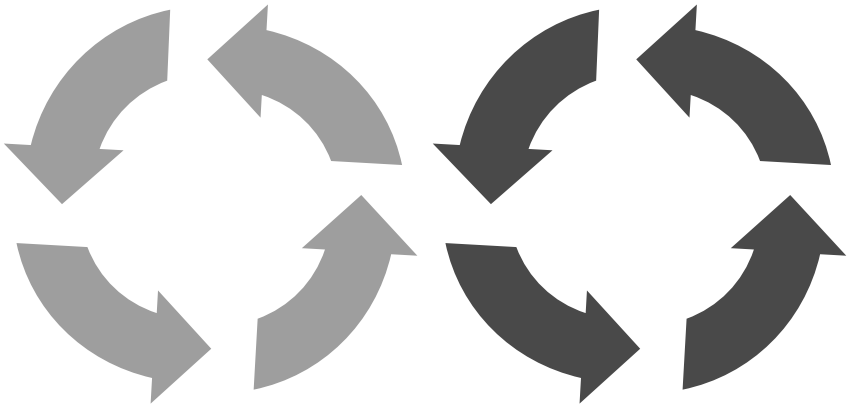
Lección con ciclos

Al trabajar con ciclos tenemos que tener cuidado de checar que la condición se modifique y eventualmente pueda ser falsa, de lo contrario vamos crear un ciclo que nunca acaba.

```
suma = 0
cond = suma < 10

while cond:
    suma += 1
```



¡No es tan obvio!



Joaquín Badillo

Problema Relevante

3n + 1 (también conocido como la Conjetura de Collatz)

Considera el siguiente procedimiento.
Me das un número entero y positivo:
- Si tu número es par, lo divido entre 2
- Si tu número es impar lo multiplico por 3 y le agrego 1.

Ahora, la pregunta interesante...

Imagina que sigo repitiendo este procedimiento con el resultado y termino hasta que llegue al 1. Por ejemplo si me das inicialmente el número 6 todos los números que obtendría hasta detenerme serían:
$6 \longrightarrow 3 \longrightarrow 10 \longrightarrow 5 \longrightarrow 16 \longrightarrow 8 \longrightarrow 4 \longrightarrow 2 \longrightarrow 1$
(Llegué al 1 y por lo tanto me detuve)

¿Este procedimiento eventualmente terminará sin importar que número (entero y positivo) me diste inicialmente?

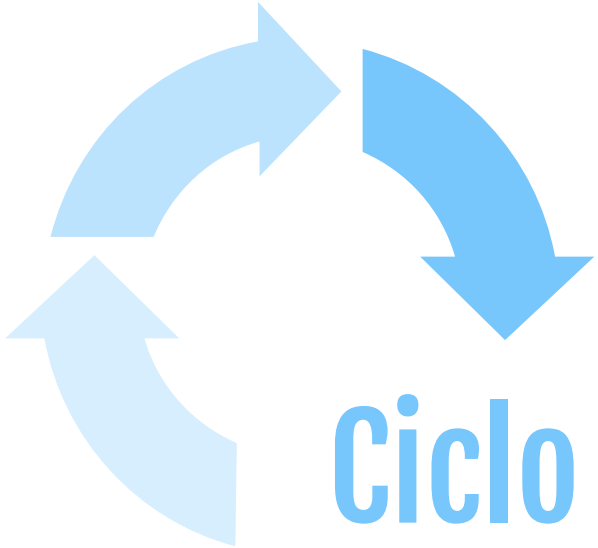
En Python:

```
def f(n):
    while n != 1:
        if n % 2 == 0:
            n /= 2
        else:
            n = 3*n + 1
```

¿Esta función siempre terminara para una entrada (n) entera y positiva?

Nadie a podido responder esta pregunta todavía.
(Una respuesta requiere de una demostración de por qué siempre terminará o bien de un ejemplo en el que no termine)

Joaquín Badillo



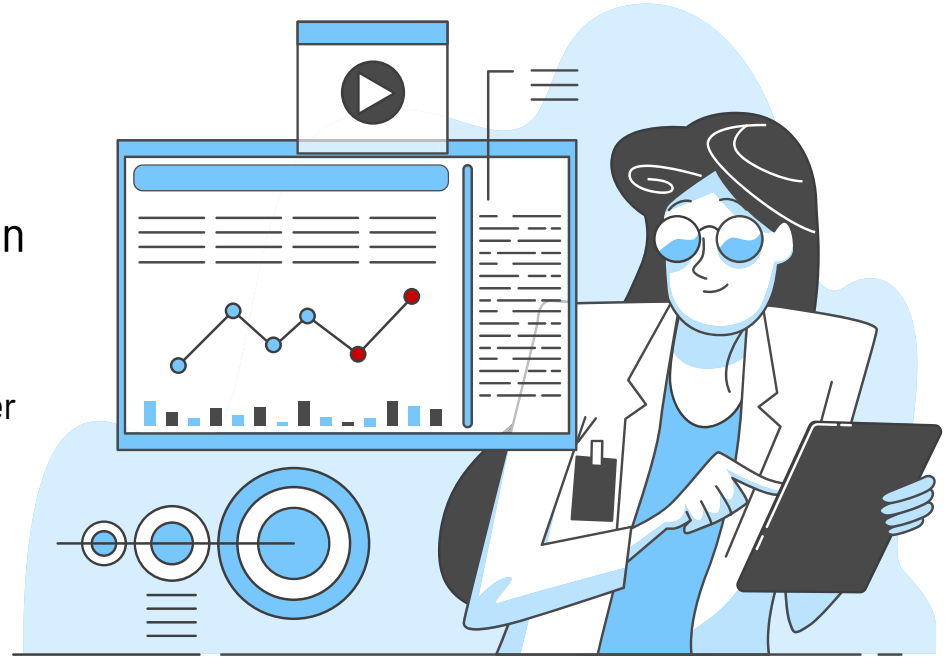
Ciclo For

Joaquín Badillo

Ciclo For

En *Python* un ciclo *for* repite uno o muchos procedimientos hasta terminar de *recorrer* un *iterable*.

¡Los ciclos *for* en otros lenguajes de programación suelen ser distintos!

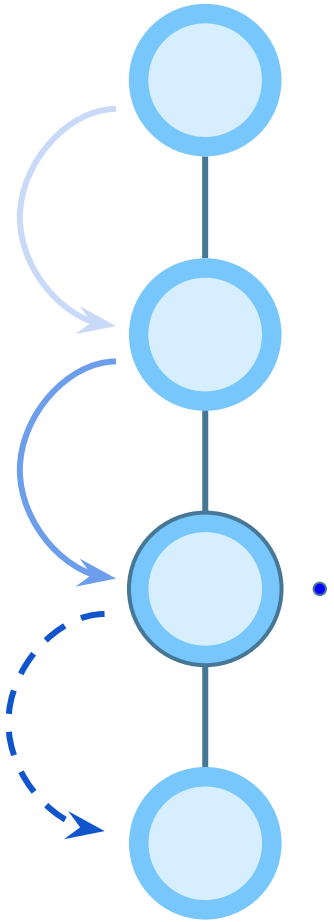


Glosario de Términos

Estructura de datos: Una manera de almacenar datos de forma organizada, con un conjunto de operaciones para utilizarla (acceder a los datos, agregar datos, eliminar...).

Recorrido: Pasar por cada uno de los elementos de una *estructura de datos* con algún tipo de orden.

Iterable: Estructura de datos que se puede recorrer. (listas, strings, diccionarios, tuplas, rangos, conjuntos...)



Joaquín Badillo

Joaquín Badillo

Ciclo For

Sintaxis de Python

for *variable* in *iterable*:
    procedimiento1
    procedimiento2
    ⋮
    procedimientoN

En Español

por cada elemento en el iterable:
    realiza este *procedimiento*
    y este otro *procedimiento*
    etc.

Ejemplos en python

```
nums = [“Tres”, “Dos”, “Uno”]

for value in nums:
    print(value)
```

Se tiene una lista llamada *nums* que contiene los *strings* “Tres”, “Dos” y “Uno”. Con un ciclo for se imprime cada uno de los valores contenidos en la lista, por lo que en pantalla se ve una cuenta regresiva.

Ejemplos en python

```
suma = 0
values = [1, 2, 3, 4, 5]

for value in values:
    suma += value

print(suma)
```

Se tiene una lista llamada *values* que contiene los números naturales del 1 al 5. Además se tiene una variable *sum* cuyo valor es inicialmente 0. Usando un ciclo for se agrega cada uno de estos valores a *sum* y se imprime el resultado. En pocas palabras sumamos los números naturales hasta el 5.

Ejemplos en python

```
suma = 0

for value in range(6):
    suma += value

print(suma)
```

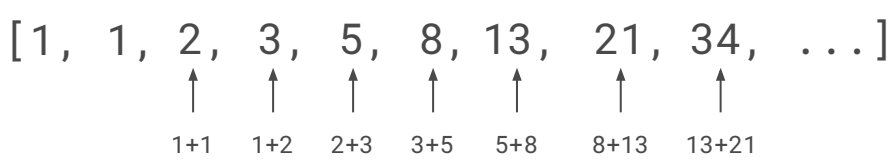
Para no tener que crear listas de números a mano, *Python* trae una función llamada *range*. Si a esta función le das un número entero positivo (llamémoslo *n*) te dará una especie de lista (en realidad es una estructura nueva llamada rango), que empieza en 0 y termina en n-1.

Ejemplos en python

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

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

    return current
```

Esta es una función para obtener el *n*-ésimo número de Fibonacci.

Los números de Fibonacci son una sucesión (una “lista”) de números que se obtiene empezando con los números 1 y 1, tal que el siguiente número se obtiene sumando los 2 anteriores:



Manos a la obra

Actividad: *Lista de Tareas*

Joaquín Badillo

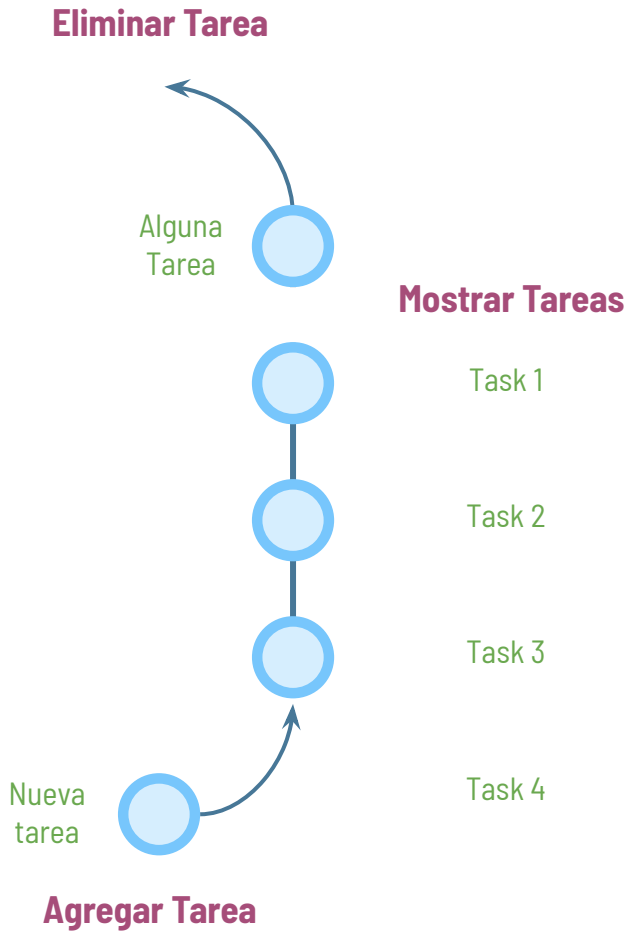
Hoy vamos a crear una lista de tareas muy sencilla. Primero debemos observar que de forma general cualquier lista de tareas soporta 3 funciones básicas:

Agregar Tarea **Eliminar Tarea** **Mostrar Tareas**

addTask(tasks, task)
La función de agregar tarea toma una lista (la lista de tareas) y un string (la tarea) y agrega la tarea a la lista.

removeTask(tasks, task)
La función de eliminar tarea toma una lista (la lista de tareas) y un string (la tarea) y elimina la tarea de la lista.

printTasks(tasks)
La función de mostrar tareas toma una lista (la lista de tareas) y muestra por pantalla todas las tareas.



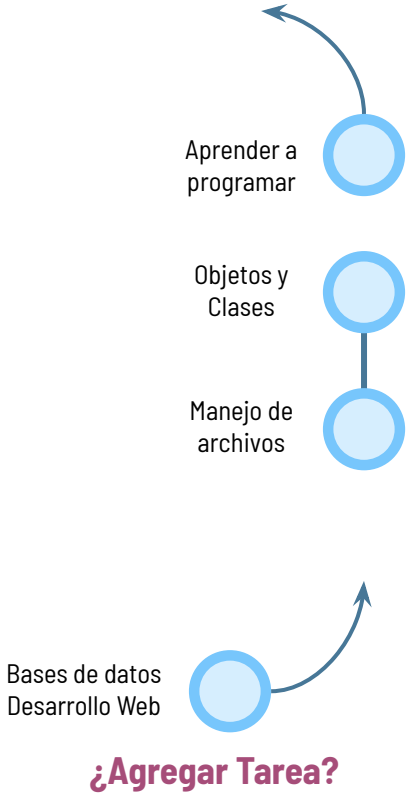
Nuestra lista de tareas

Propongo la siguiente lista de tareas para el curso:

1. Aprender a programar
    - 1.1. Aprender cómo darle instrucciones a una computadora
    - 1.2. Aprender un poco acerca de eficiencia

2. Aprender acerca de programación orientada a objetos
    - 2.1. Crear una clase que represente la tarea
    - 2.2. Crear una clase que represente la lista de tareas
        - 2.2.1. ¿Podemos agregar subtareas como estoy haciendo aquí?

3. Aprender a guardar información de forma permanente
    - 3.1. Manejo de Archivos
    - 3.2. Guardar las tareas en un archivo



Pregunta detonante:

¿Se puede tener un ciclo for infinito en Python?

Joaquín Badillo

Lista Circular

Una lista circular es una lista que cuando llega al "final" regresa al primer elemento.

```
from itertools import cycle

circularList = cycle([1, 2, 3])

for num in circularList:
    print(num)
```

Estoy agregando la función *cycle* que otras personas hicieron. Esta función implementa una lista circular (formalmente es un iterador)

Los ciclos infinitos también son posibles en los *ciclos for* de Python, pero es muy raro encontrar este problema de forma inesperada.

¿Quién usaría una lista circular?

Imagina una lista de canciones (una *playlist*).

Ahora supongamos que haces una aplicación que reproduce la música en esta playlist. (Cómo reproducir la música desde un programa es irrelevante ahora)

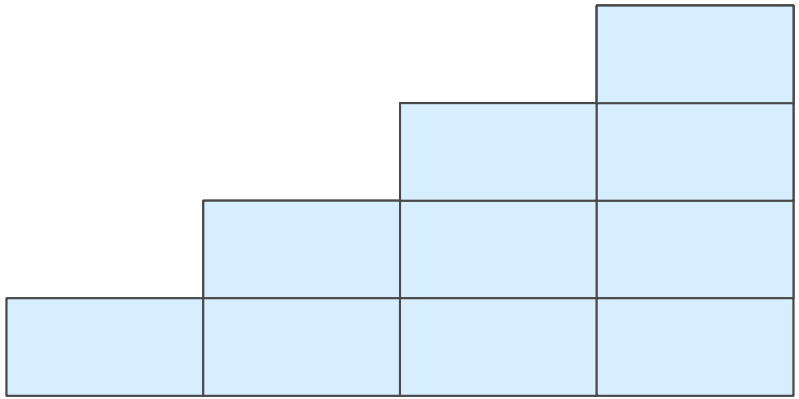
Podrías hacer que la aplicación deje de reproducir canciones una vez que pasaron todas. O bien podrías hacer que **cicle**. Una lista circular implementa esta segunda funcionalidad!

Las listas circulares también pueden ser útiles para algo conocido como *Scheduling* en sistemas operativos... aunque esto se hace en lenguajes como C

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).

Joaquín Badillo

Idea Principal

¿Cómo se vería la recursión con el caso base?

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

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

$2^1 = 2$

Joaquín Badillo

Idea Principal

¿Cómo se vería la recursión con el caso base?

$$2^3 = 2^2 \cdot 2$$
$$2^2 = (2) \cdot 2$$
$$2^1 = 2$$

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 < 0):
        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).

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

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.

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

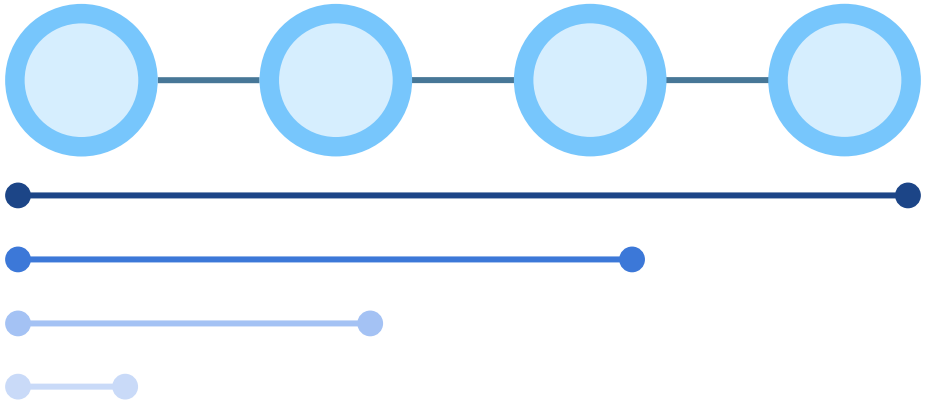
De hecho... la base 2 si tiene algo especial (pues es la que usa una computadora internamente). Exponenciar naturales también se puede lograr usando *bit shifting* (<<).

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

Si gustas puedes investigar acerca de esta operación

Joaquín Badillo

Estructuras Recursivas



Recursividad

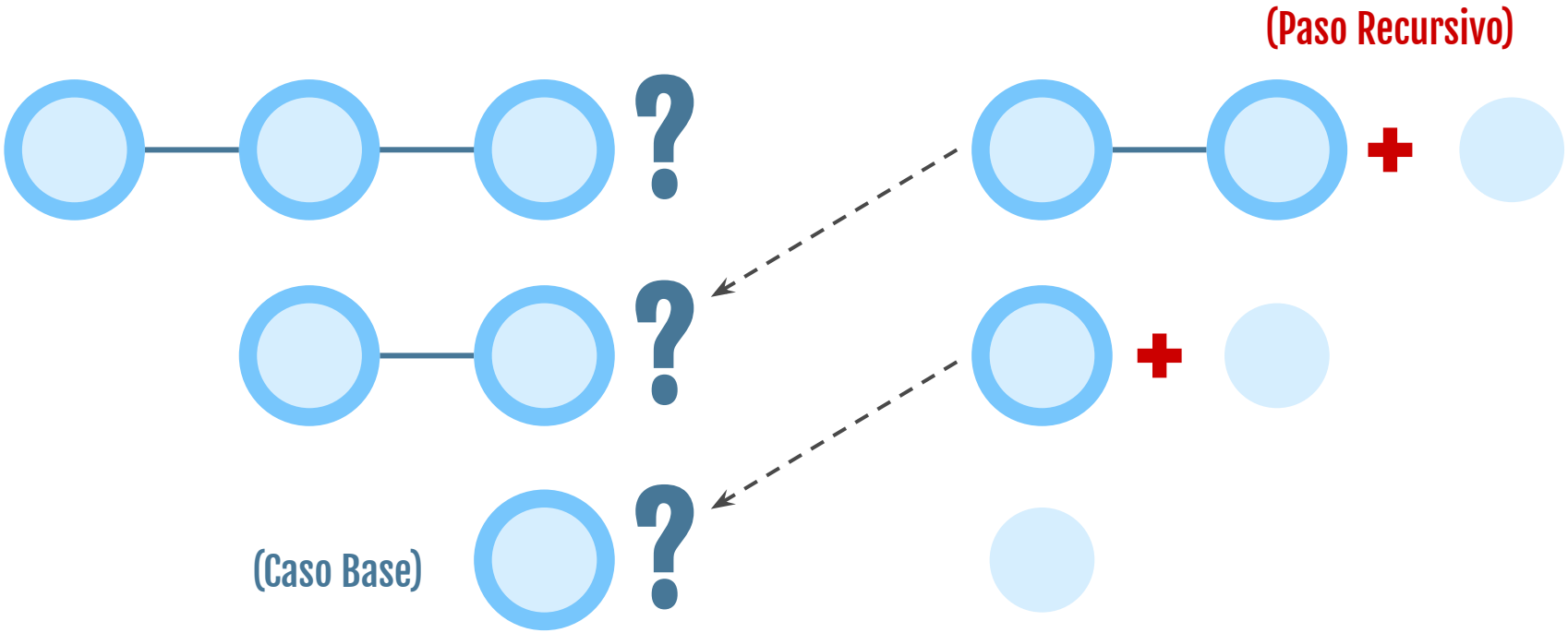
- **¿Qué es una lista de tamaño 3?**
  - Una lista de tamaño 3 es una lista de tamaño 2 a la que se le agregó un elemento

- **¿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

Recursividad



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.

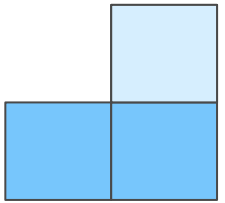
X(2)
Primero necesito hacer X(1).

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.

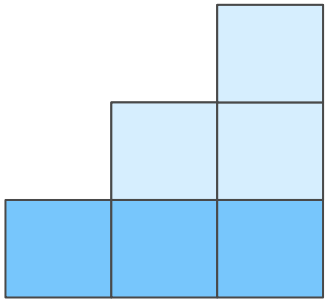
X(2)
Primero necesito hacer X(1). (Listo)
Ahora agrega una lista de 2 cuadrados abajo

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

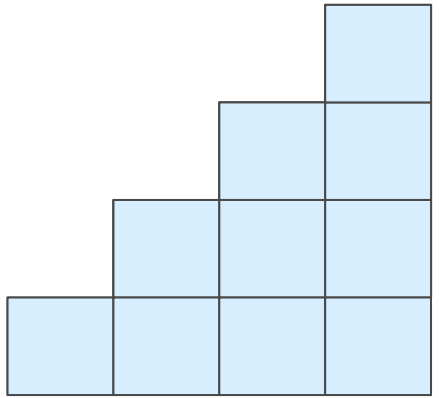


Joaquín Badillo

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

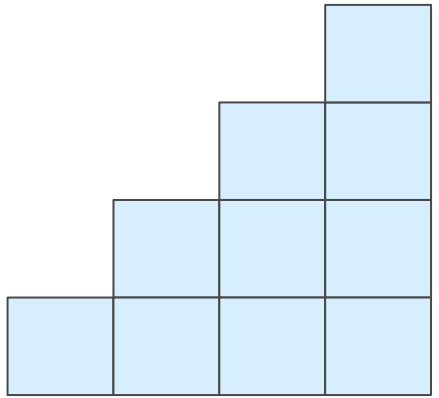
Paso a Paso

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

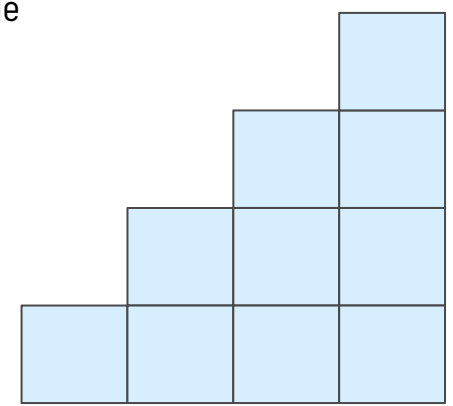
Tuvimos que hacer el mismo *procedimiento* (crear una pirámide) con valores de su altura decrecientes

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")



La sucesión de Fibonacci es una recursión

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

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?

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)
```

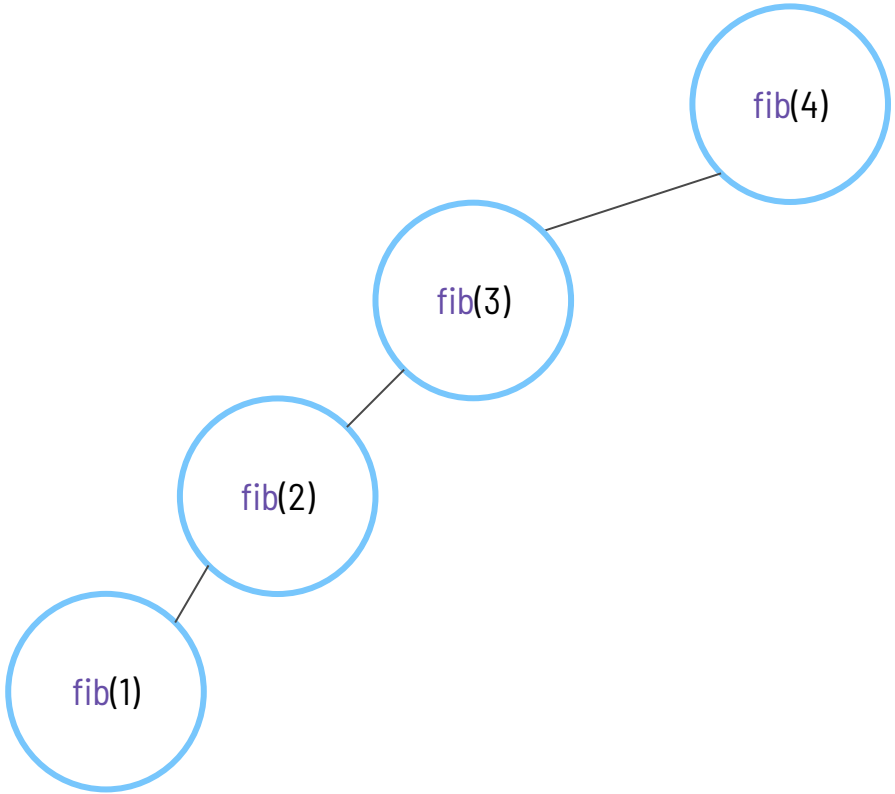
(Recursión)

Joaquín Badillo

¿Por qué la solución recursiva es tan lenta?

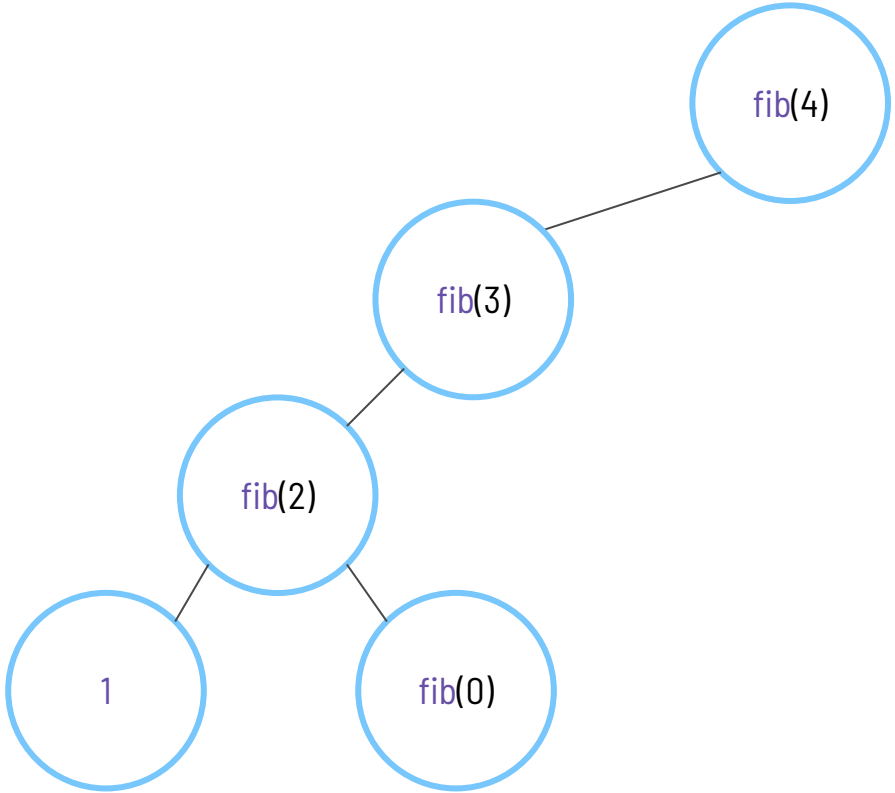
Joaquín Badillo

Procedimientos



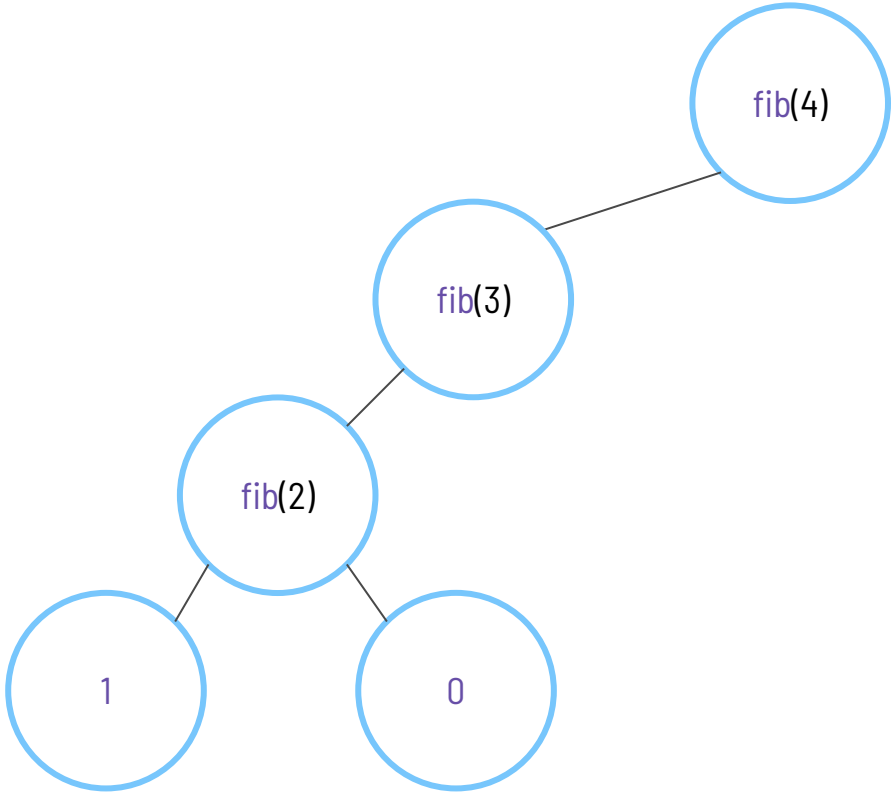
Joaquín Badillo

Procedimientos



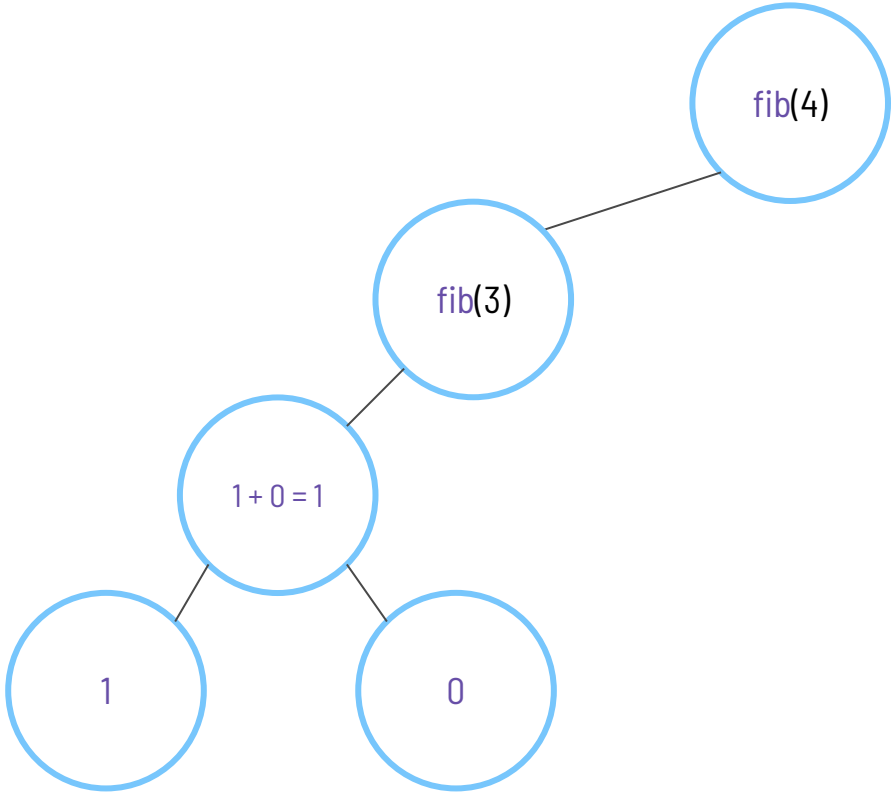
Joaquín Badillo

Procedimientos



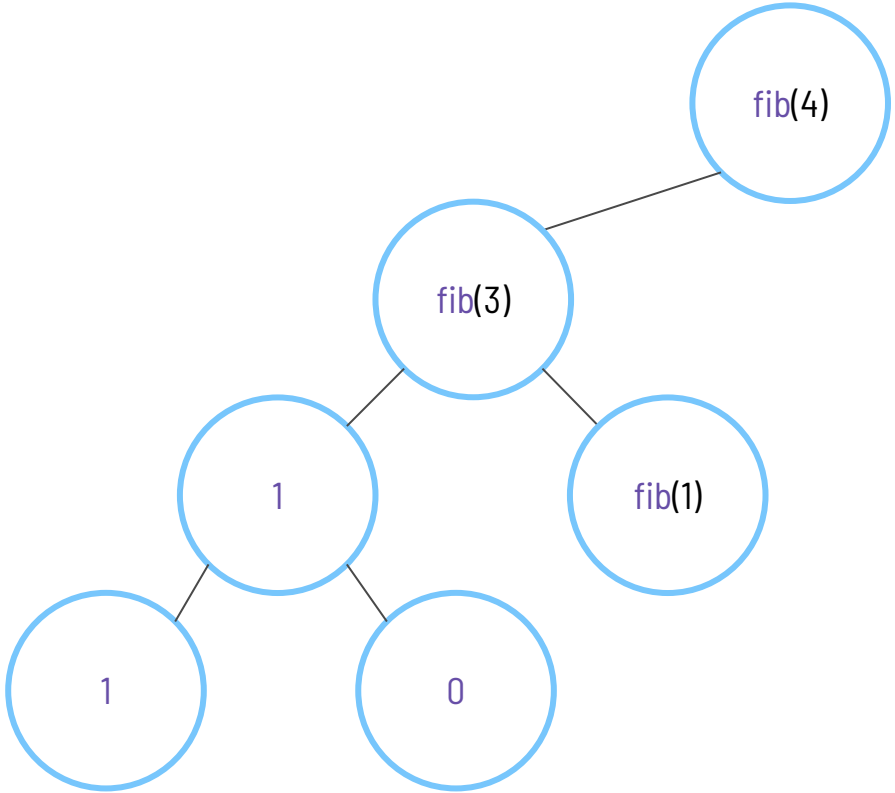
Joaquín Badillo

Procedimientos



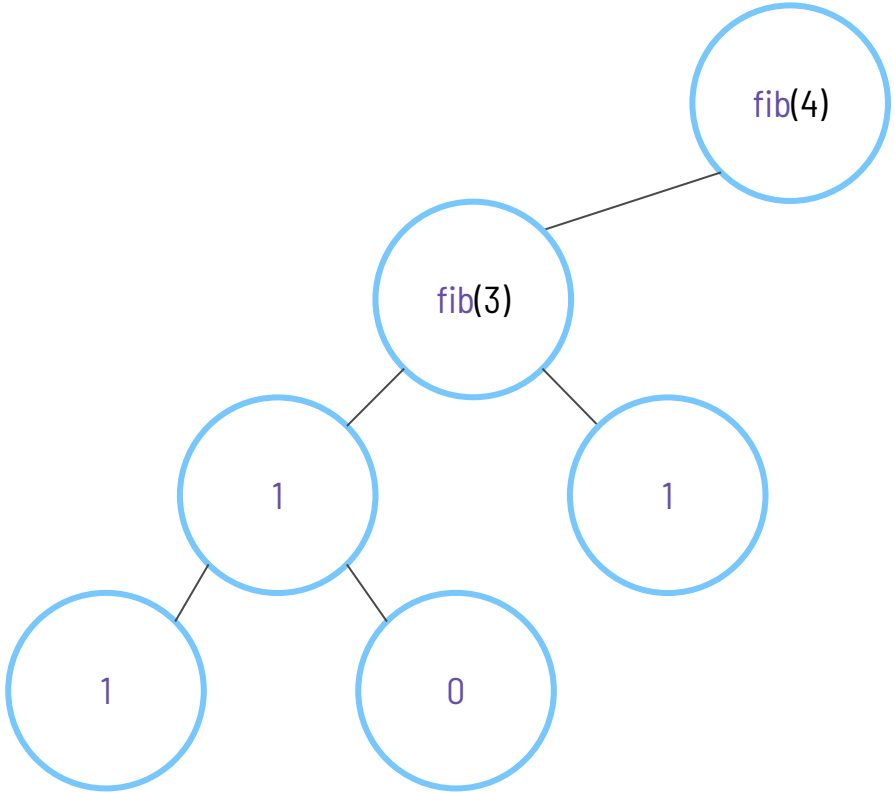
Joaquín Badillo

Procedimientos



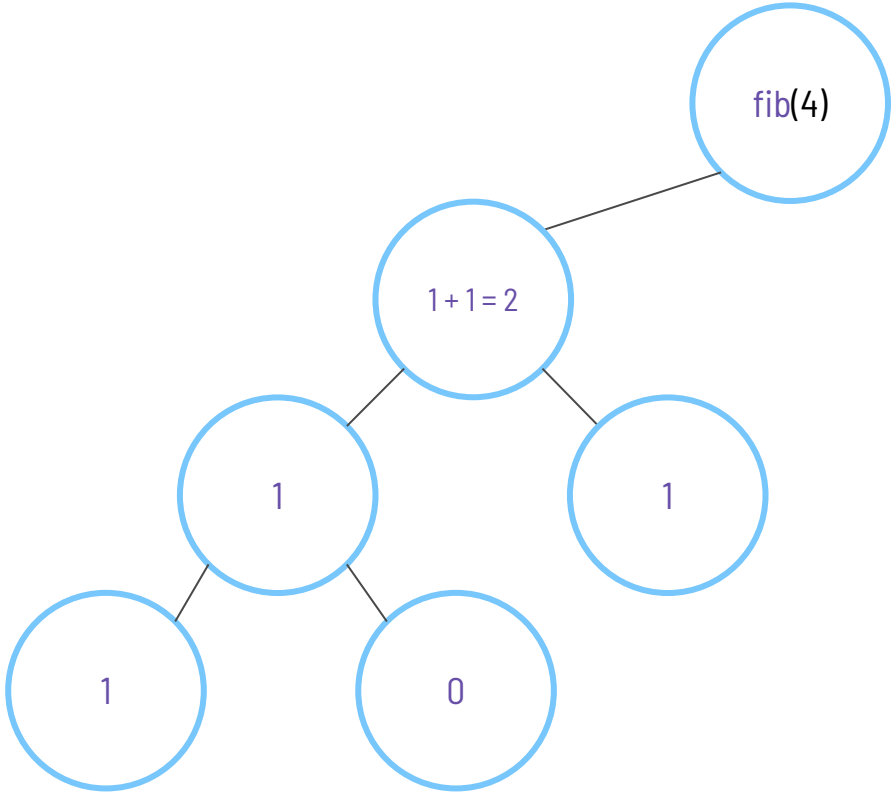
Joaquín Badillo

Procedimientos



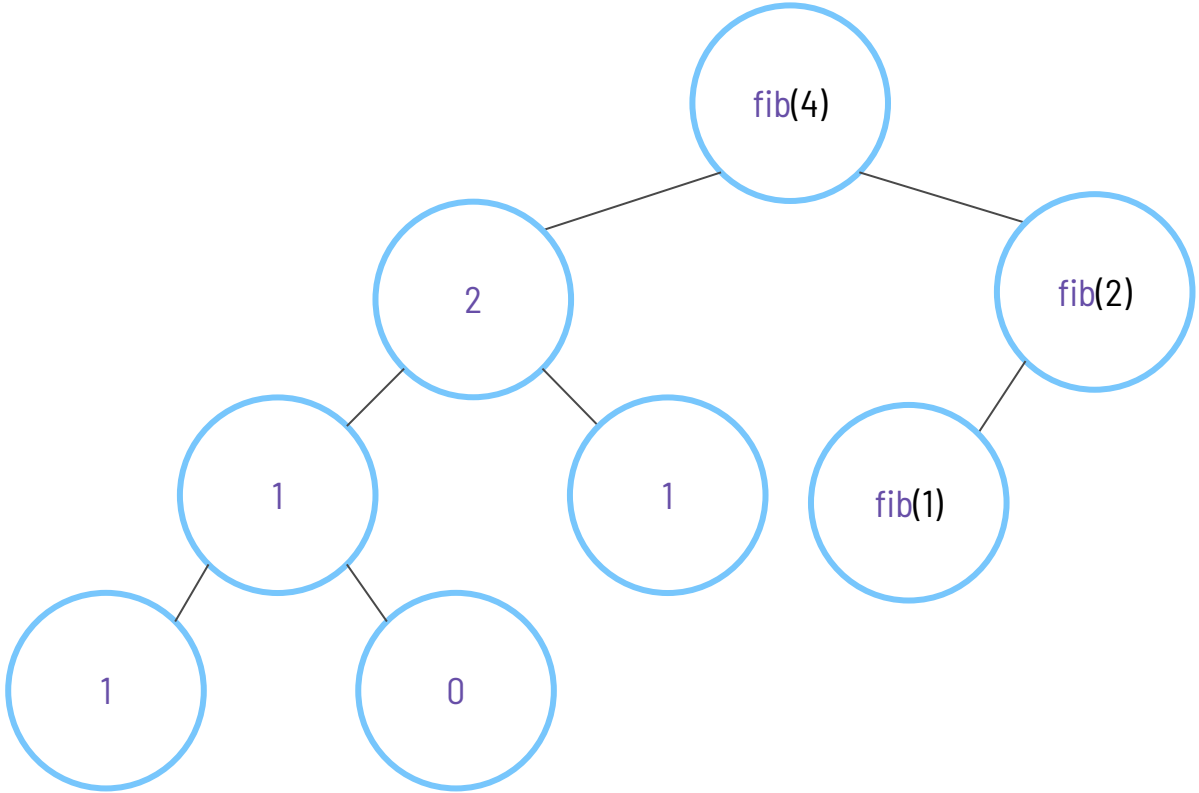
Joaquín Badillo

Procedimientos



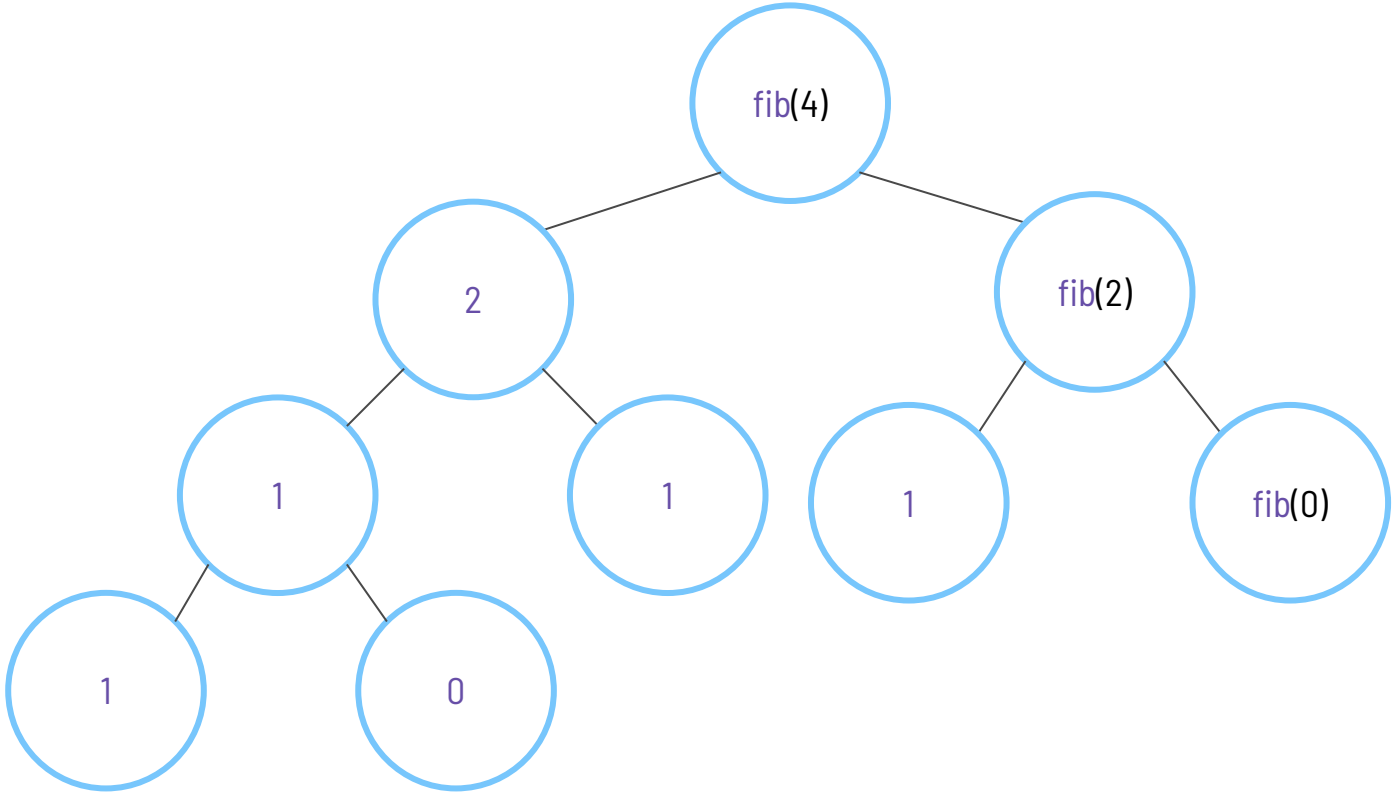
Joaquín Badillo

Procedimientos



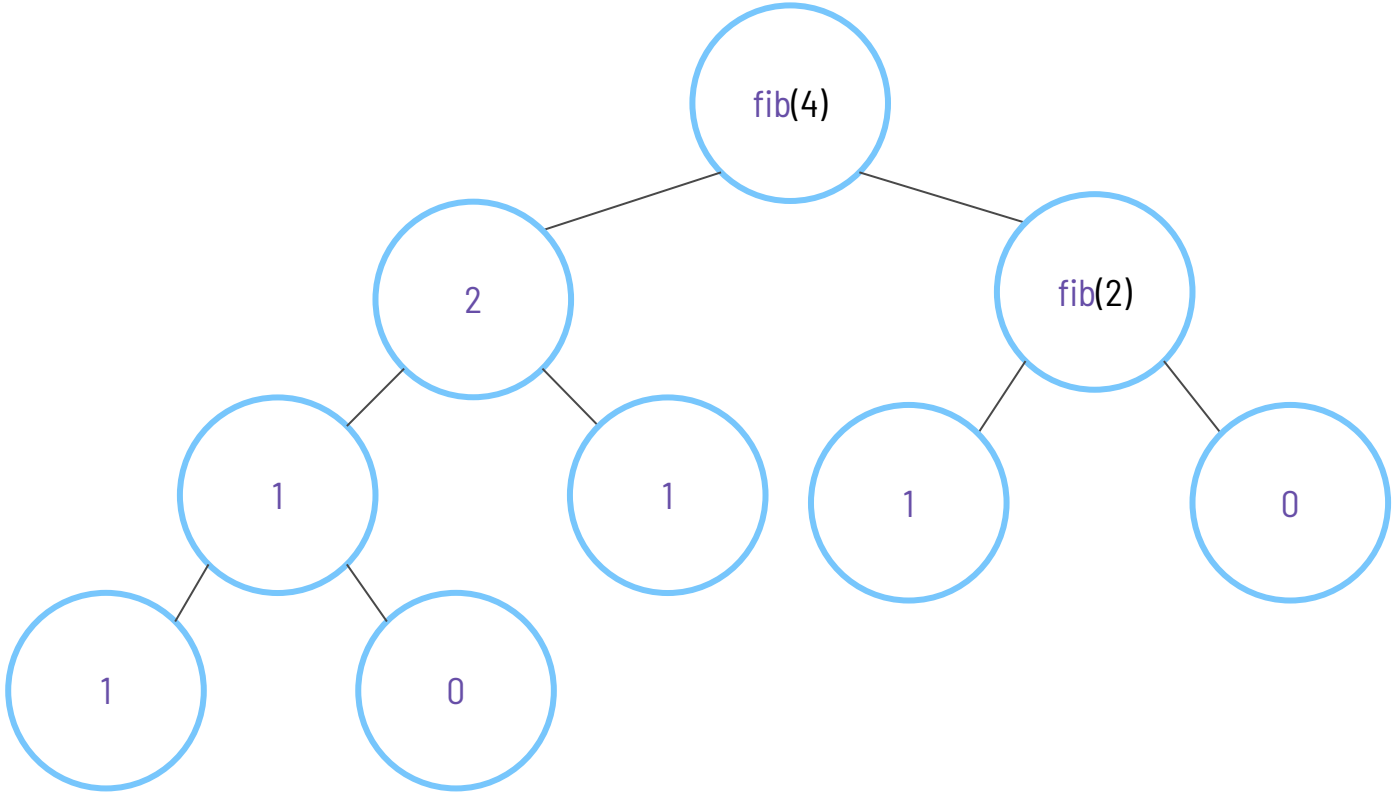
Joaquín Badillo

Procedimientos



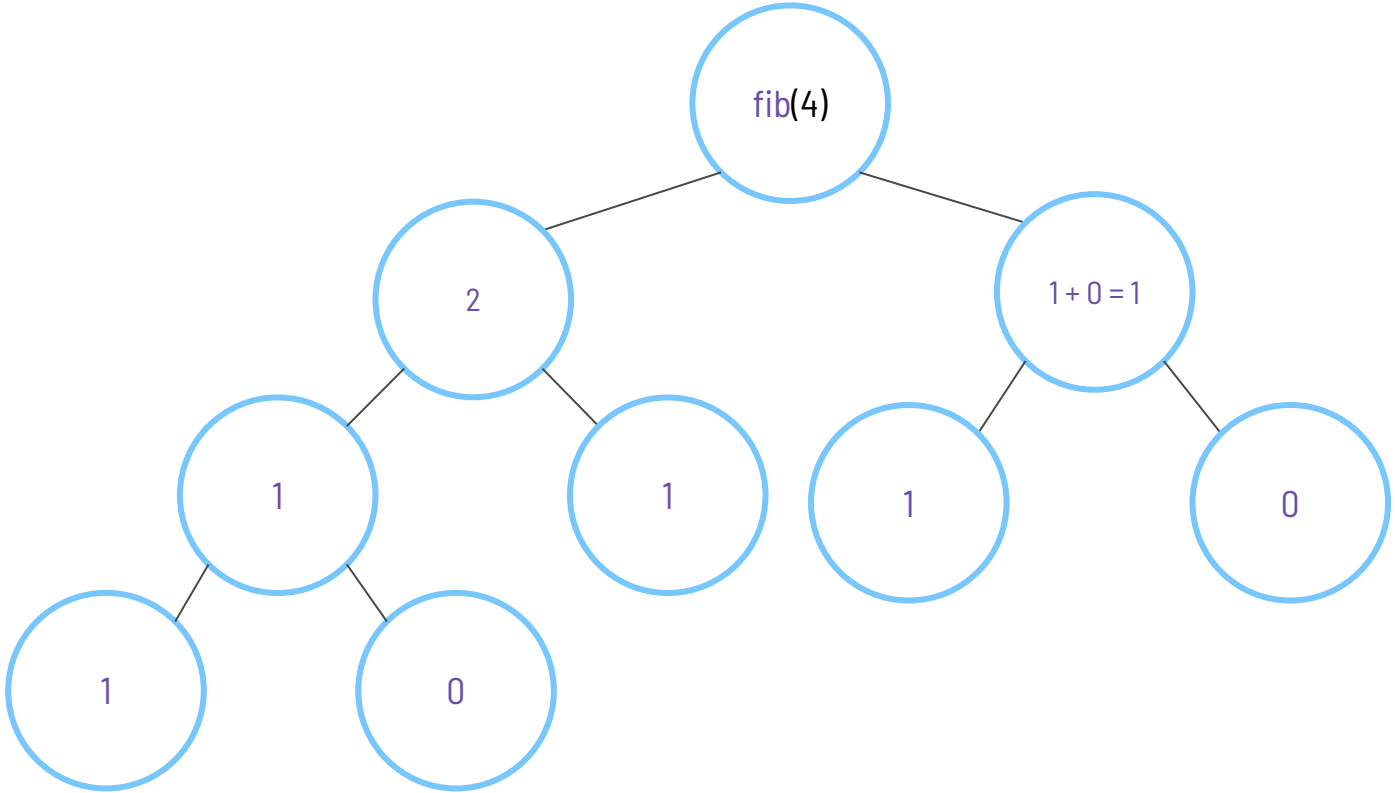
Joaquín Badillo

Procedimientos



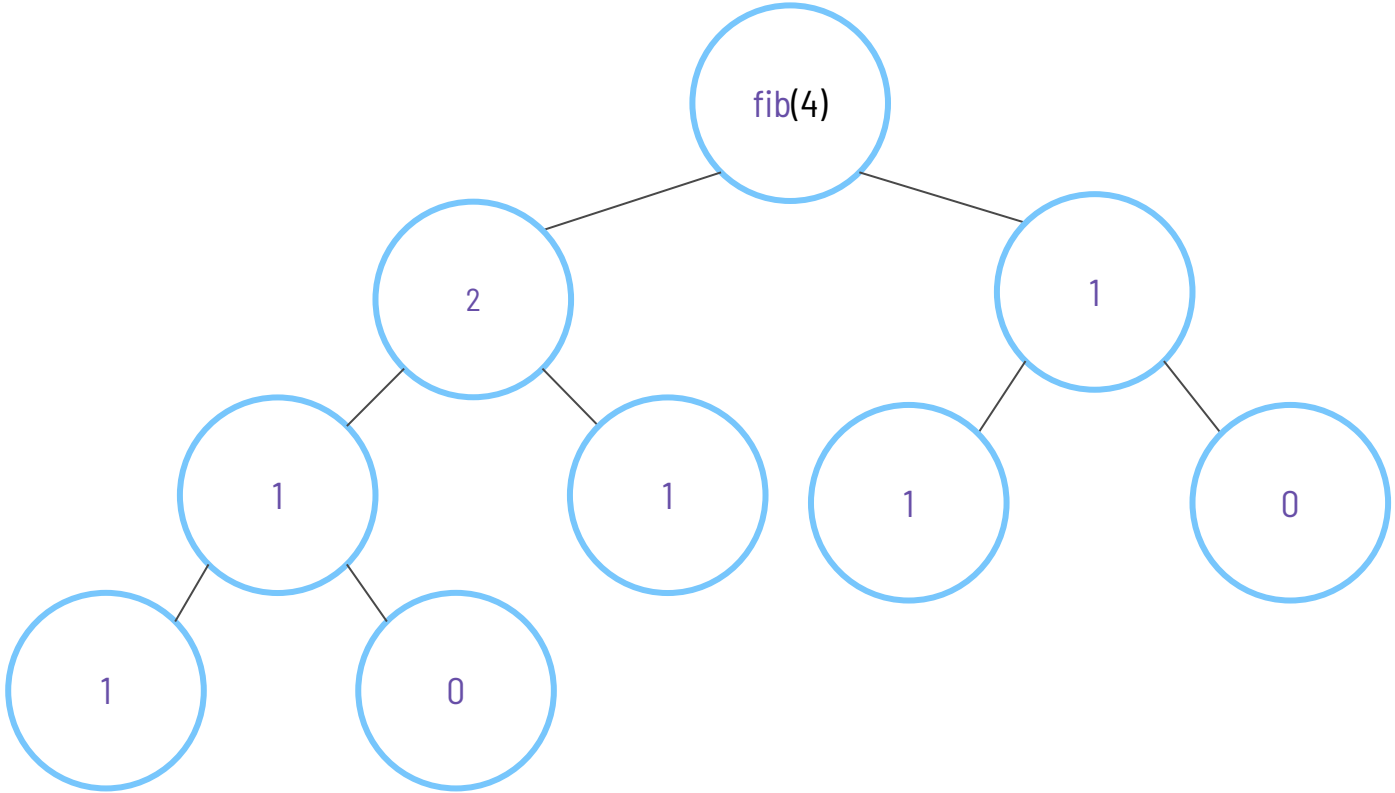
Joaquín Badillo

Procedimientos



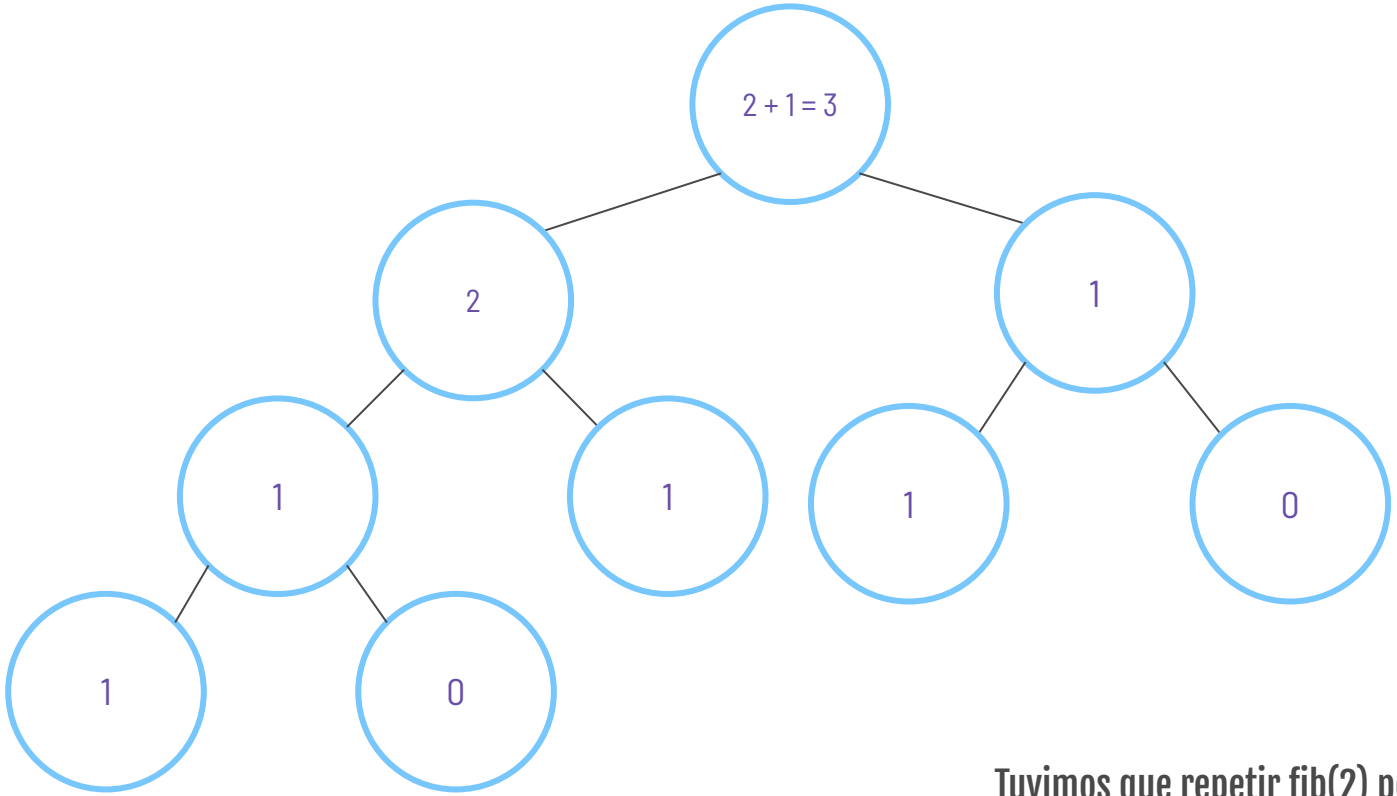
Joaquín Badillo

Procedimientos



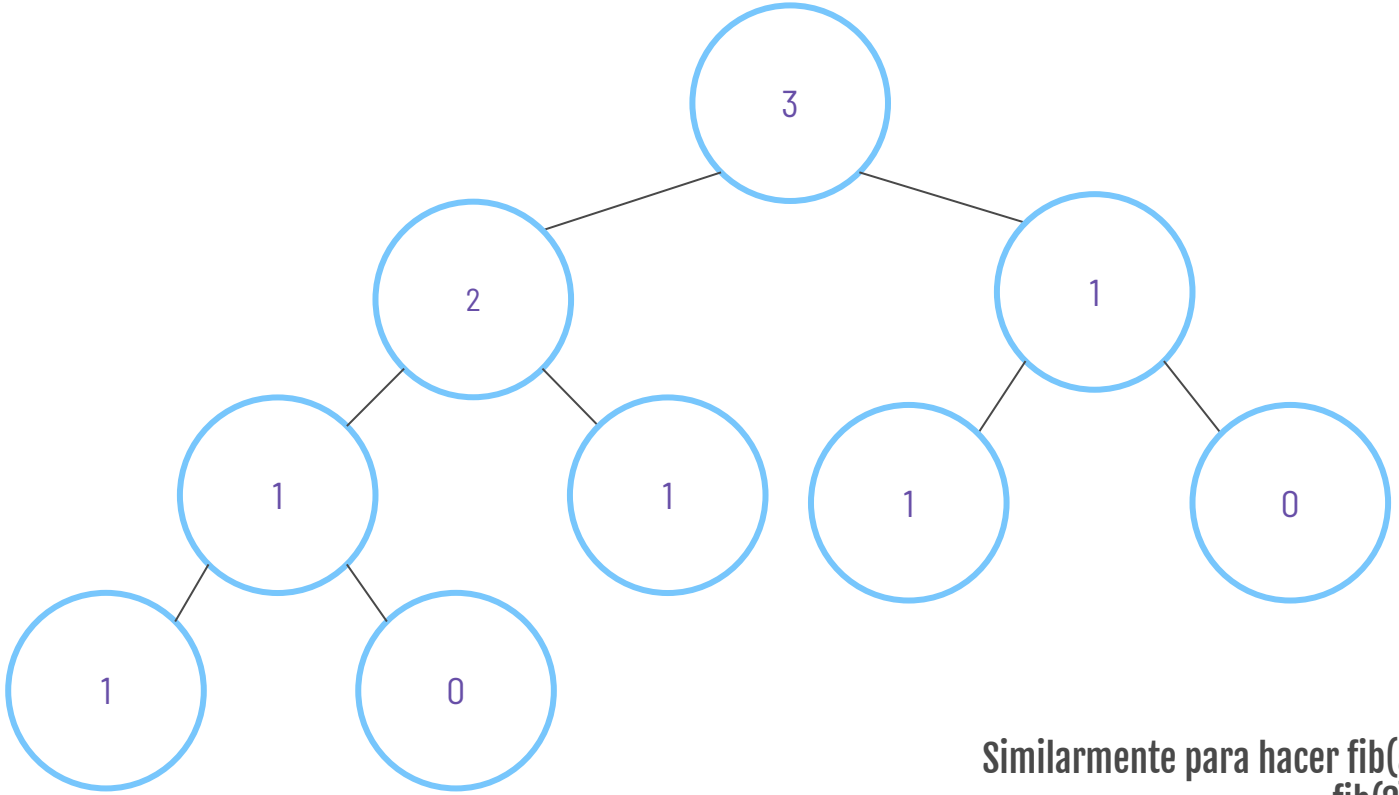
Joaquín Badillo

Procedimientos



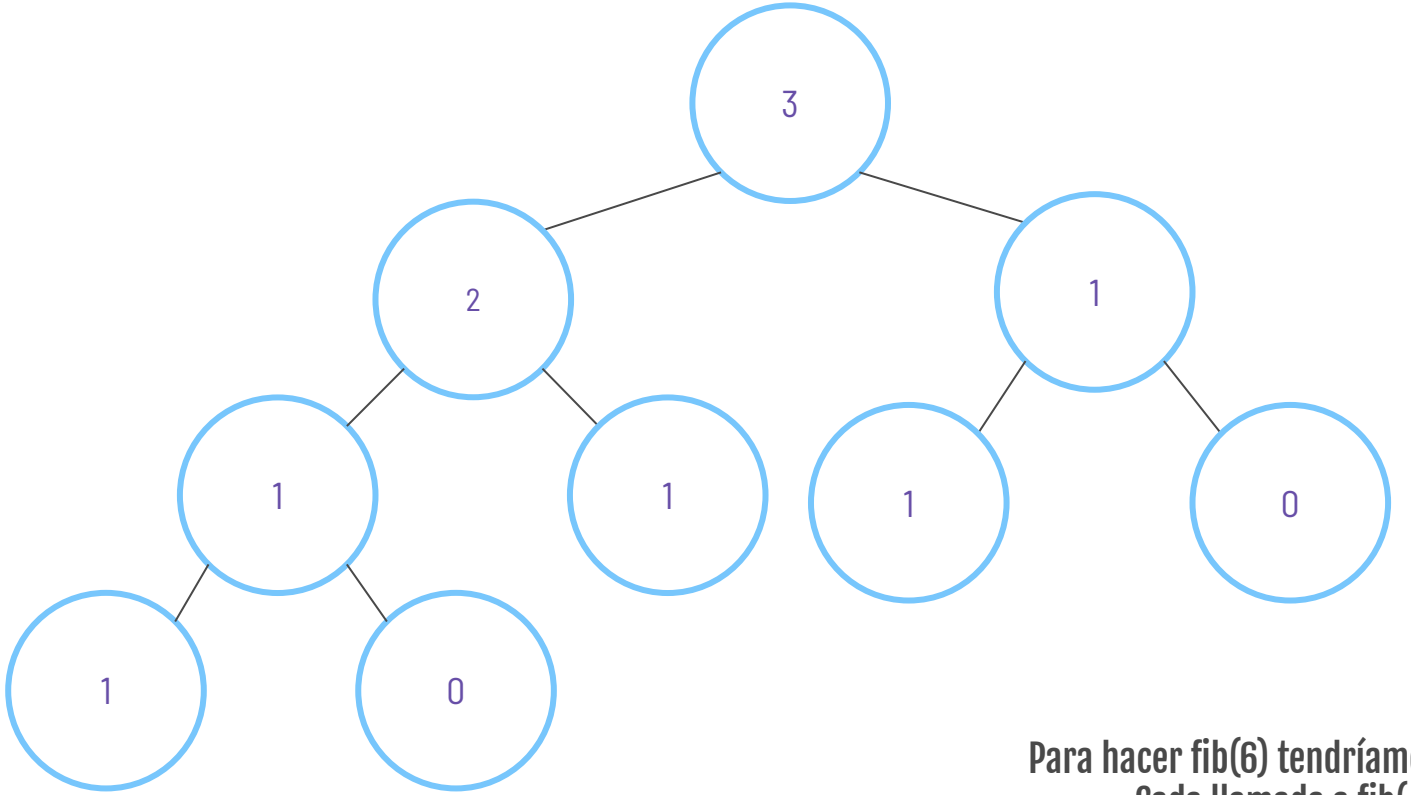
Tuvimos que repetir fib(2) porque la computadora no memorizó que fib(2) = 1

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)

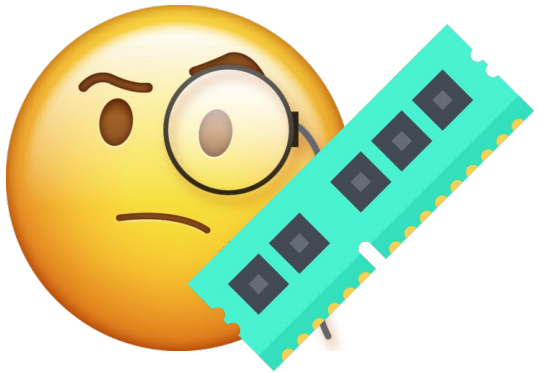
(Recursión con cola)

```
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*