uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 4 Recursión

Profesores EDA

Objetivos

1) Describir el concepto de recursión y dar ejemplos de su uso.
2) Identificar el/los caso(s) base(s) y el/los caso(s) general(es) de una función recursiva.
3) Escribir funciones recursivas para resolver problemas.
4) Comparar soluciones recursivas e iterativas para soluciones de problemas sencillos.

Índice

1) **Qué es recursión?**
2) Algunos ejemplos de recursión.
3) Tipos de recursión.
4) Iteración y Recursión

Qué es recursión?

- Una forma para conseguir repetición.
- Una función que se llama a sí misma.

```
def funcionA(...) -> … :
        …
        funcionA(...)
        …
```

- Algunas estructuras de datos son recursivas (por ejemplo, nodos y árboles).

```
class DNode:
    def __init__(self, e: object, prev_node: 'DNode' = None, next_node: 'DNode' = None) -> None:
        self.elem = e
        self.next = next_node
        self.prev = prev_node
```

Las tres leyes de la recursión

1. Un algoritmo recursivo debe llamarse a sí mismo.
2. Un algoritmo recursivo debe tener al menos un **caso base**. Un caso base es aquel que puede ser resuelto directamente.
3. Un algoritmo recursivo debe tener al menos un **caso recursivo** (un caso más complejo donde el problema se divide en subproblemas más pequeños). Las llamadas recursivas deben converger al caso base.

Índice

1) Qué es recursión?
2) **Algunos ejemplos de recursión.**
3) Tipos de recursión.
4) Iteración y Recursión

Ejemplo 1: Función Factorial

$$n! = \begin{cases} 1 & si\ n=0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & si\ n>=1 \end{cases}$$

4!=4.3.2.1=24

4!=4.(3.2.1)=4.3!

Definición Recursiva

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

Ejemplo 1: Implementación de factorial

```
def factorial(n: int) -> int:
    """n!=n*(n-1)*...*2*1"""
    if n < 0 or not isinstance(n, int):
        return None
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Caso Base

Caso Recursivo

Trazabilidad de factorial

factorial(4)

Primera llamada

4 * factorial(3)

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

factorial(4) ⇨ 4 * factorial(3)

3 *factorial(2)

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

factorial(4) ⇨

4 * factorial(3)

3 *factorial(2)

2 * factorial(1)

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

factorial(4) ⇒

4 * factorial(3)

3 *factorial(2)

2 * factorial(1)

1 * factorial(0)

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

factorial(4) ⇨

4 * factorial(3)

3 *factorial(2)

2 * factorial(1)

1 * factorial(0)

1 Caso Base

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

factorial(4) ⇨

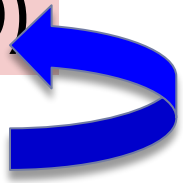
4 * factorial(3)

3 *factorial(2)

2 * factorial(1)

1 * factorial(0)

1

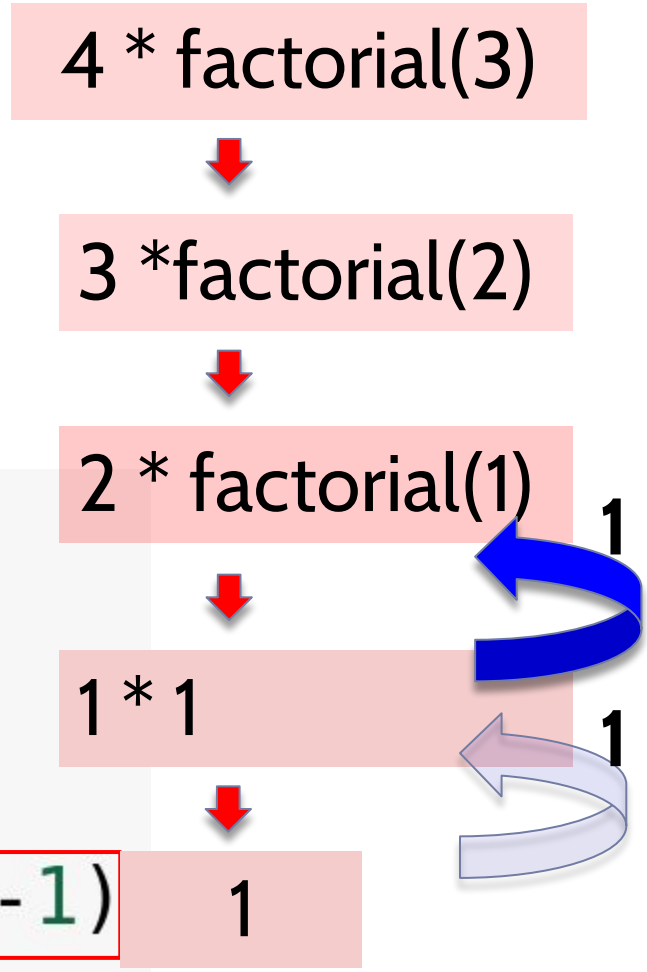


1

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

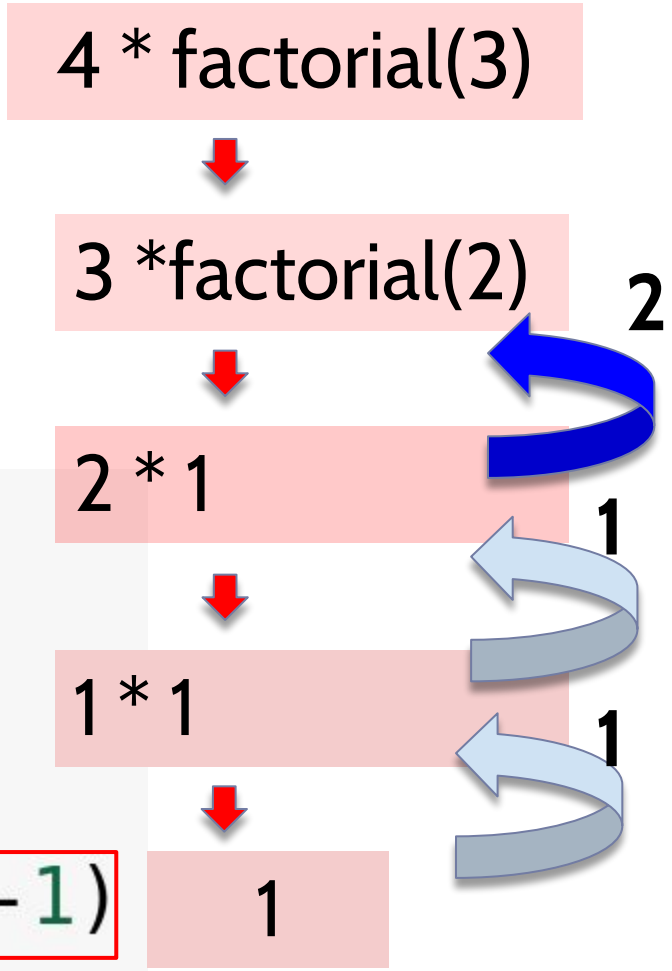
factorial(4) ⇨




```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

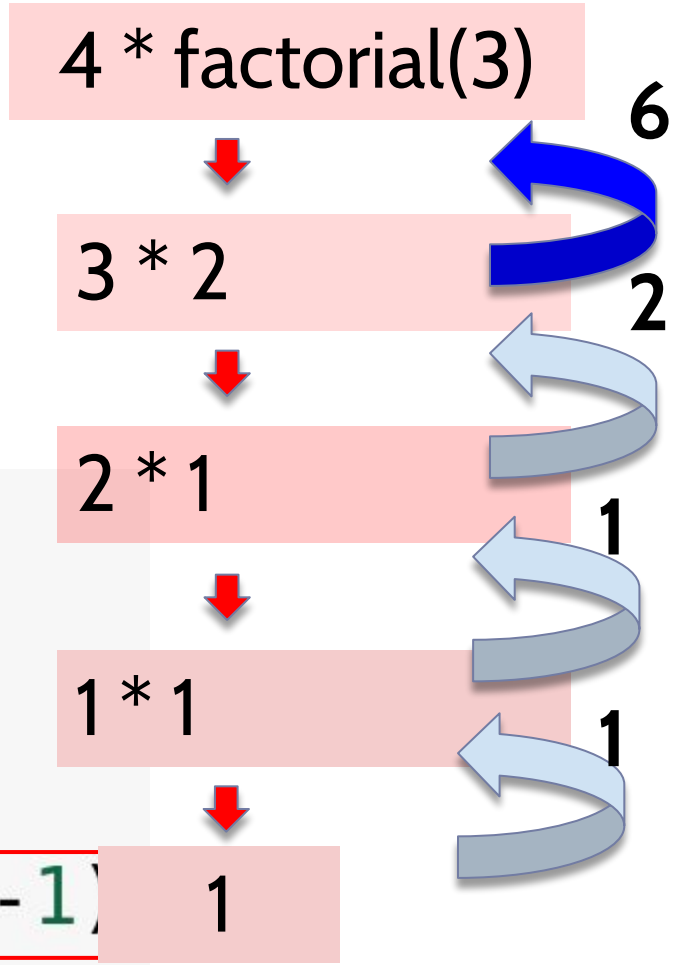
factorial(4) ⇨




```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

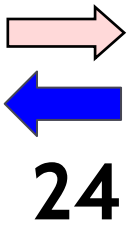
factorial(4) ⇨

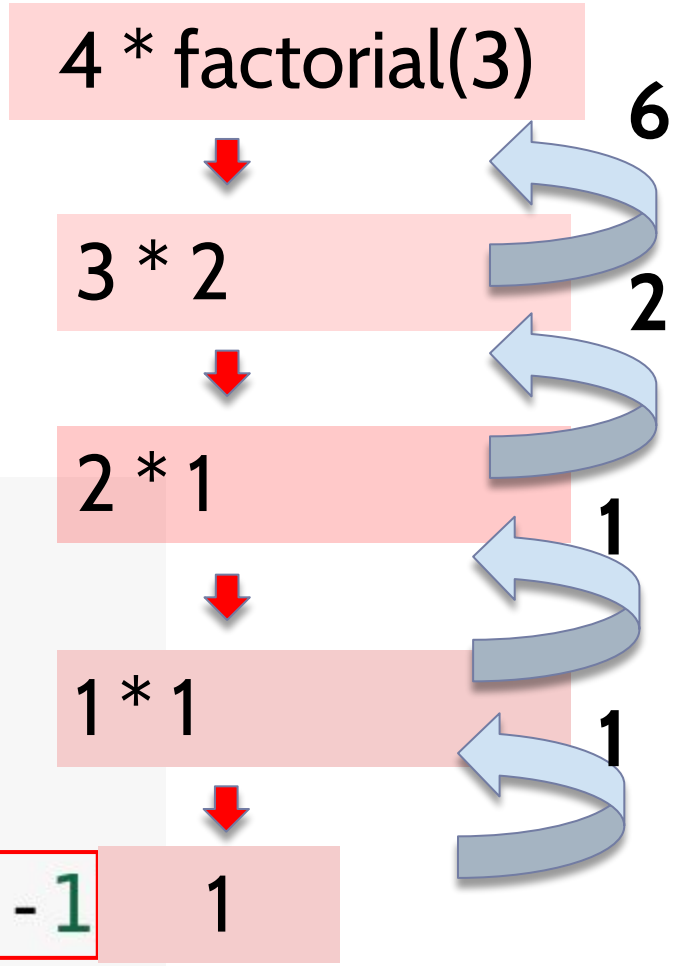



```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

Trazabilidad de factorial

factorial(4)

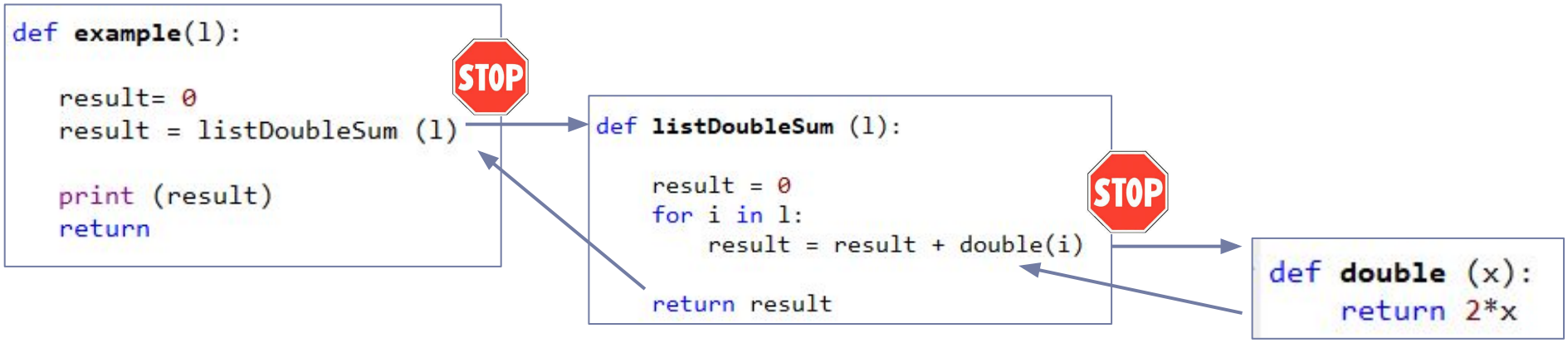




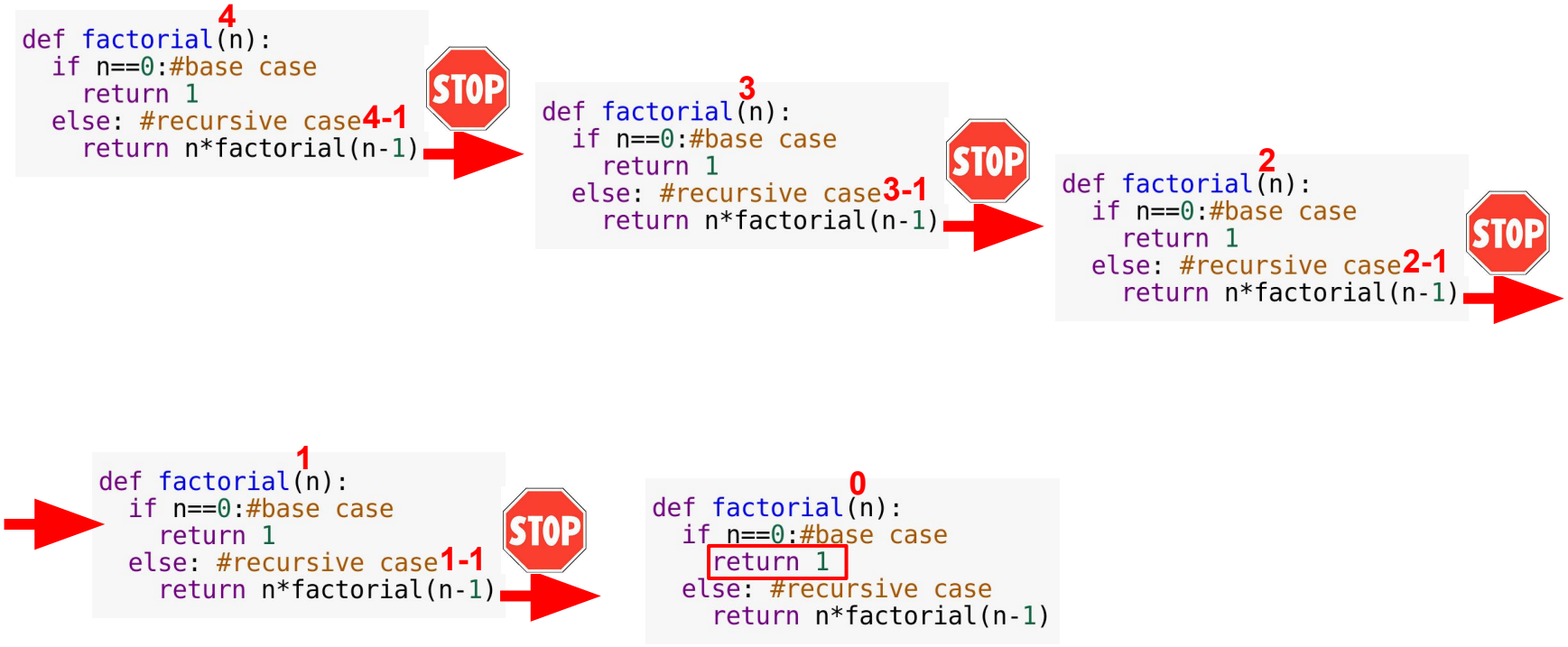

```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1
```

LLamadas a funciones

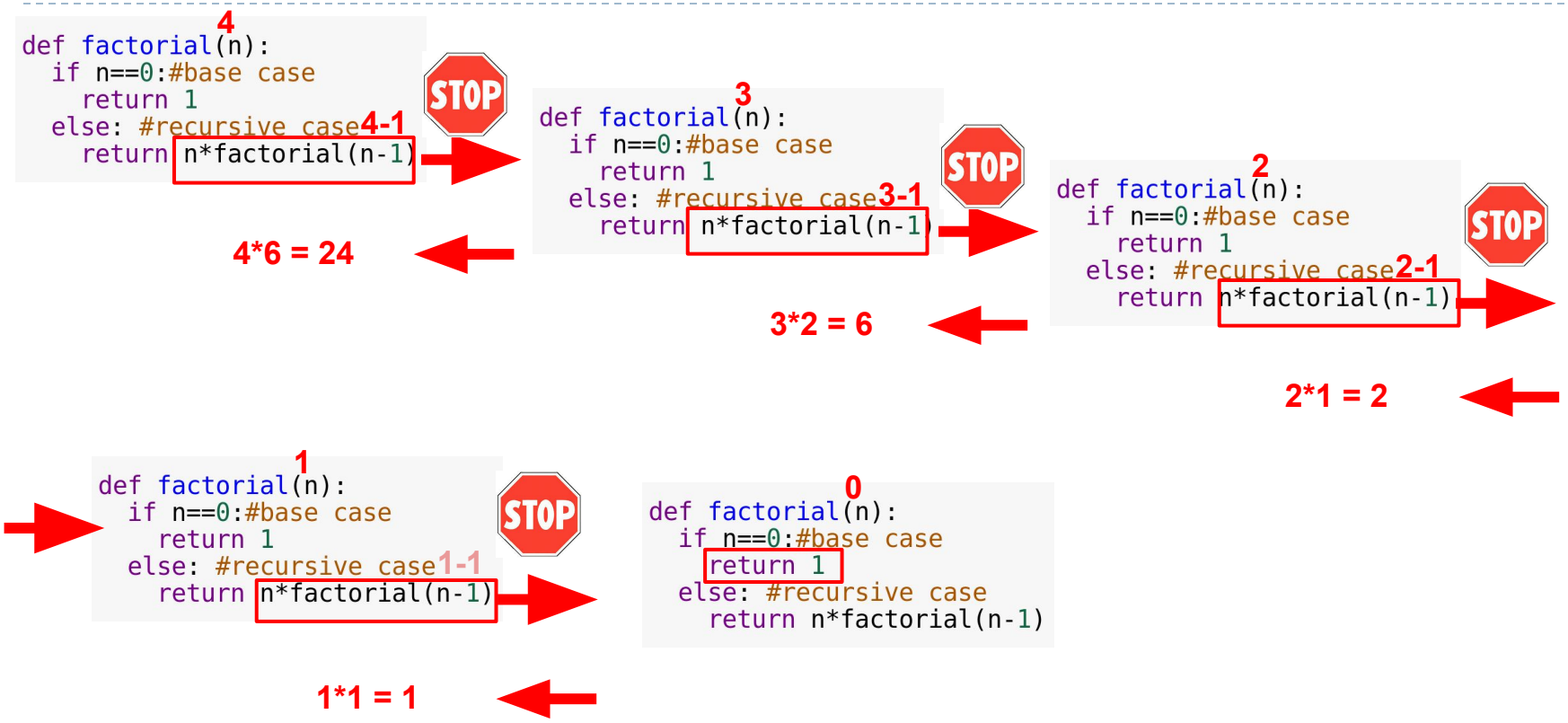
- ¿Qué ocurre cuando desde una función se llama a otra función?



LLamadas a funciones (recursivas)



LLamadas a funciones (recursivas)



Ejemplo: Multiplicación por suma

$$5 \times 3 = 15 = 5 + 5 + 5$$

Ejemplo: Multiplicación por suma

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
def multiply_by_sum(x: int, y: int) -> int :
```

Primero, piensa sobre el caso base

Ejemplo: Multiplicación por suma

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
def multiply_by_sum(x: int, y: int) -> int :
    if y==0:
        return 0
```

Correcto!!!. Ahora piensa sobre el caso recursivo.

Ejemplo: Multiplicación por suma

$$5 \times 3 = 15 = 5 + 5 + 5$$

```
def multiply_by_sum(x, y) :
    if y==0:
        return 0
    else:
        return x+multiply_by_sum(x,y-1)
```

Sí!!!, lo tienes!!!

Índice

1) Qué es recursión?
2) Algunos ejemplos de recursión.
3) **Tipos de recursión.**
4) Iteración y Recursión

Tipos de recursión

1. **Recursión Lineal**: una llamada recursiva podría producir como máximo una nueva llamada recursiva.
2. **Recursión Binaria**: una llamada recursiva debe generar dos nuevas llamadas recursivas.
3. **Multiple recursión**: una llamada recursiva puede generar tres o más llamadas recursivas.

Recursión Lineal

- Ya hemos visto algunos ejemplos: factorial, multiplicación por suma.
- Ahora, estudiaremos algunos ejemplos más:
  - Calcular la potencia de un número
  - Calcular la suma de un array de enteros.
  - Invertir una lista de Python (array)

Ejemplo Recursión Lineal: Potencia

Función potencia: power(x,n)=x^n

$$power(x,n)=\begin{cases} 1 & \text{if } n=0 \\ x*power(x,n-1) & \text{if } n>0 \end{cases}$$

Ejemplo Recursión Lineal: Potencia

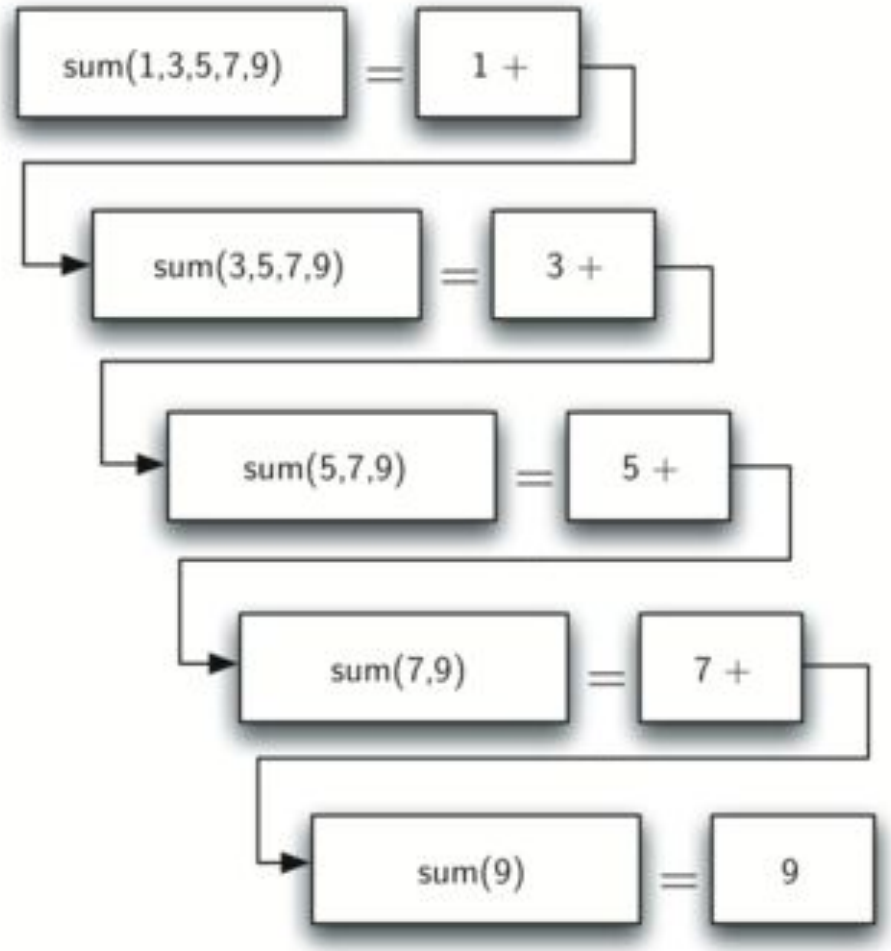
```
def power(a: int, n: int) -> int:
    """a^n, n >=0"""
    if not isinstance(a, int) or n < 0 or not isinstance(n, int):
        return None

    if n == 0:
        return 1
    else:
        return a * power(a, n - 1)
```

Suma una lista de números

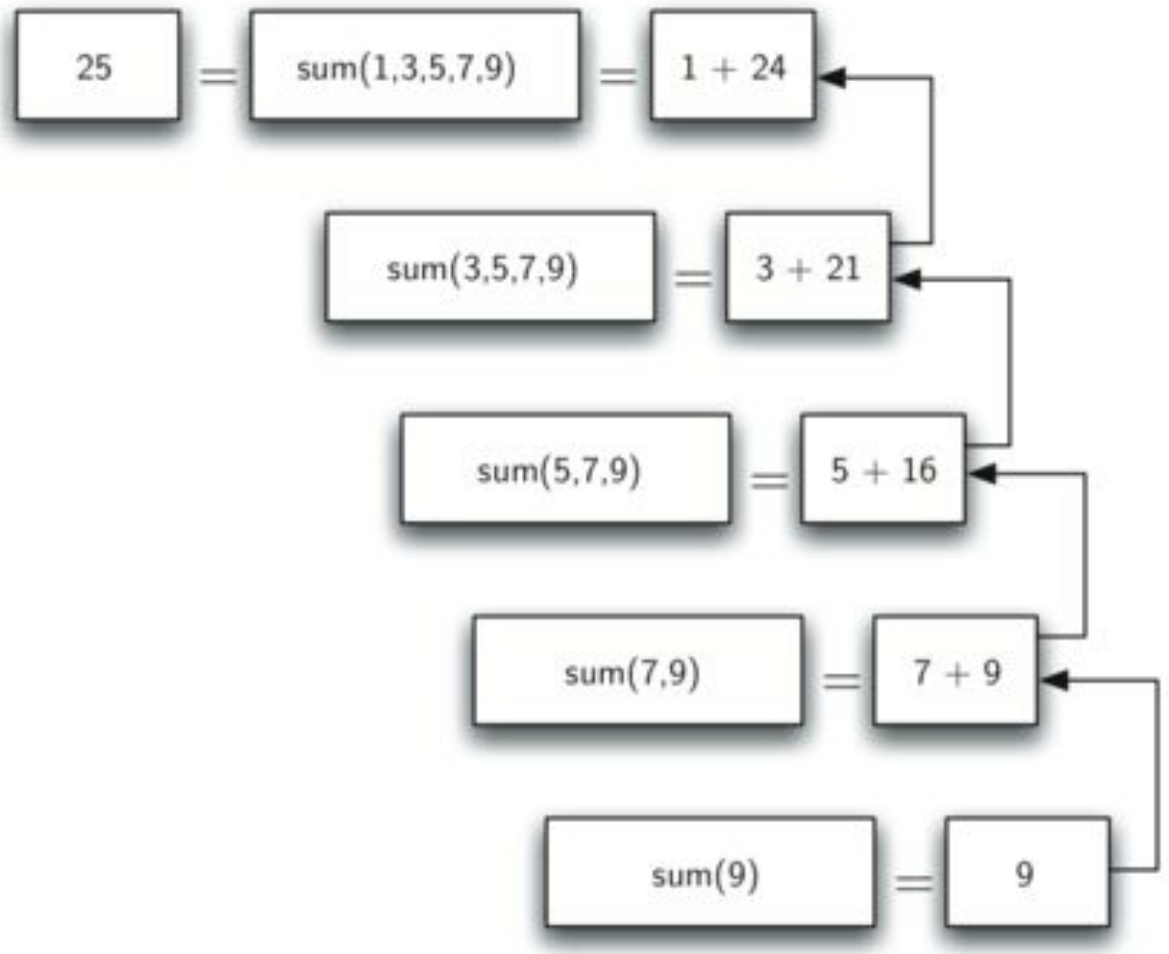
- Dada una lista de enteros [1,3,5,7,9], ¿cómo podemos obtener su suma usando recursión?

Suma una lista de números



Caso base

Suma una lista de números



Suma una lista de números

```python
def sum_list(a: list) -> list:
    """Returns the sum of the elements in the list.
    In this solution, it is not allowed to update the input list,
    but you can use slice"""
    if a is None or not isinstance(a, list):
        return None

    if len(a) == 0:
        return 0
    else:
        return a[0] + sum_list(a[1:])
```

Suma una lista de números

- Ya hemos visto una solución basada en slicing que resuelve el problema usando recursión.
- Sin embargo, desde el punto de vista de complejidad espacial no es eficiente porque con cada llamada recursiva se está creando una nueva lista: a[1:].
- Piensa en otras soluciones:
  - por ejemplo, modificando la lista.
  - sin modificar la lista y sin usar slicing.
- Solución

Invertir una lista

[8,5,3,4,1] -> [1,4,3,5,8]

Puede ser resuelto usando recursión lineal:

- guardando el primer elemento en una variable x.
- invirtiendo recursivamente la sublista data[1:]
- Por último, debes añadir x al final de la sublista invertida

Ejemplo 5: Invertir una lista

[8,5,3,4,1] -> [1,4,3,5,8]

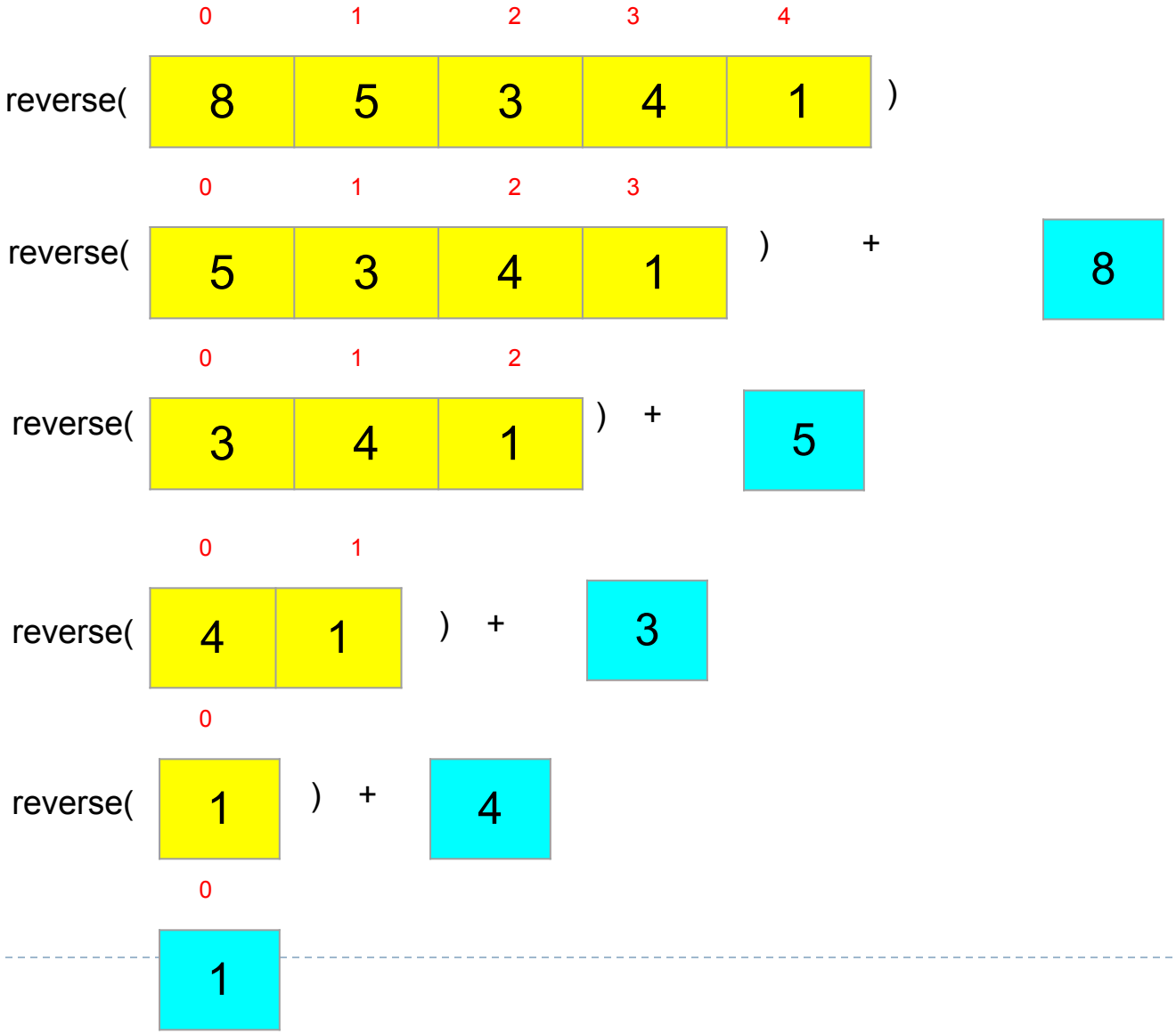
Caso Base: Cuando el array está vacío o sólo tiene un elemento, su inversa es el mismo array.

Invertir una lista

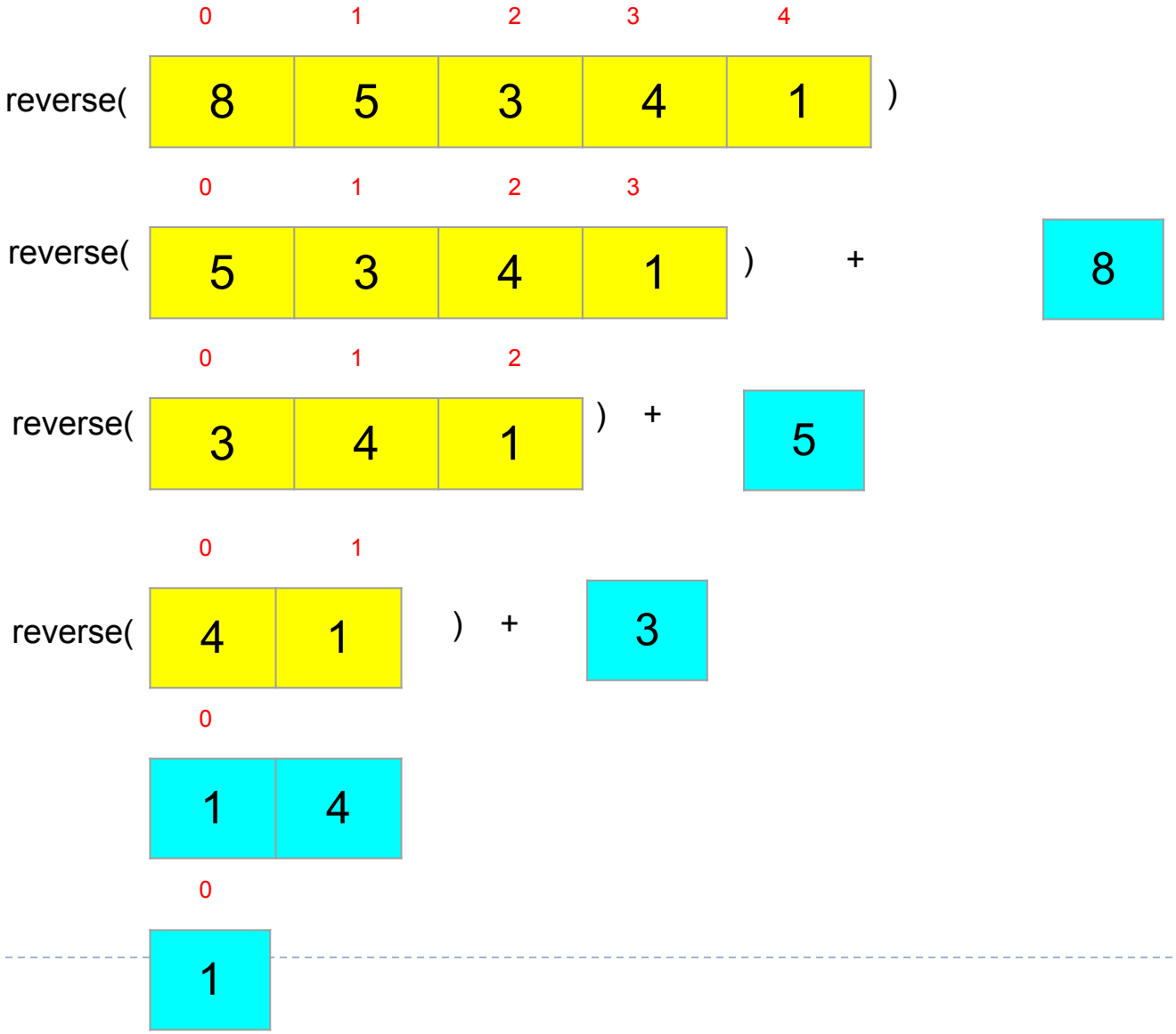
[8,5,3,4,1] -> [1,4,3,5,8]

En el caso recursivo, podemos mover el primer elemento al final del array, e invertir el resto del array, recursivamente.

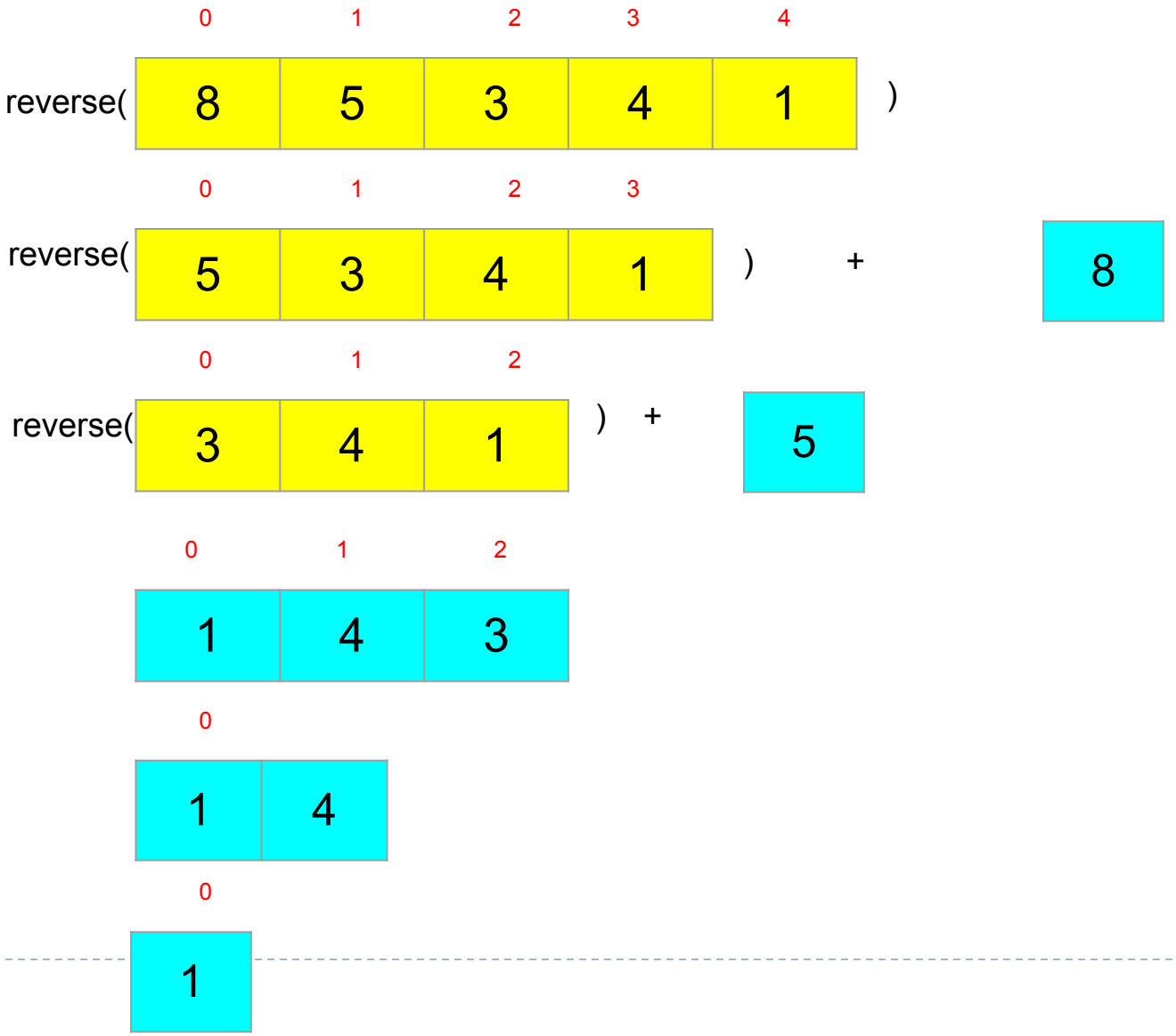
Invertir una lista



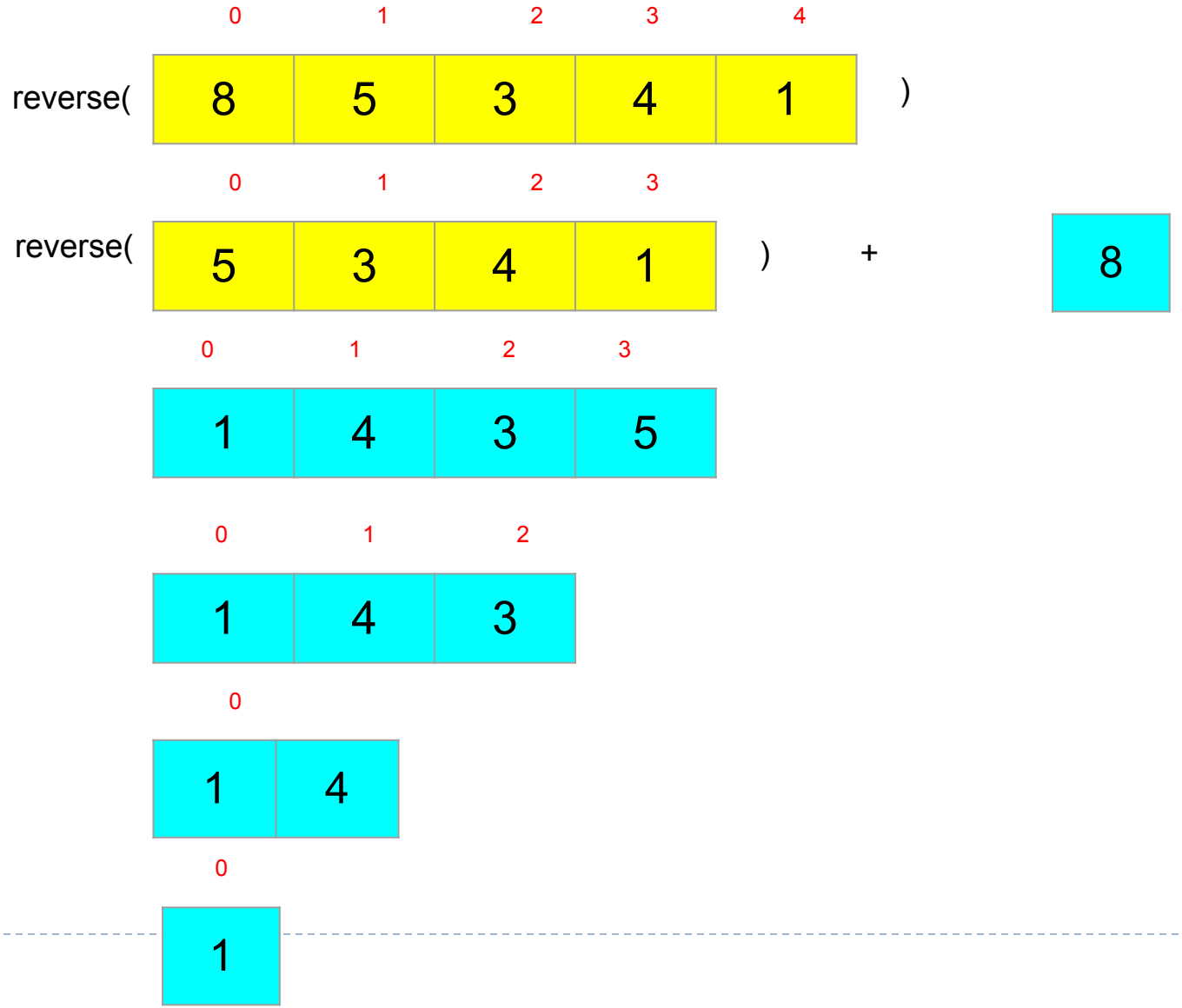
Invertir una lista



Invertir una lista

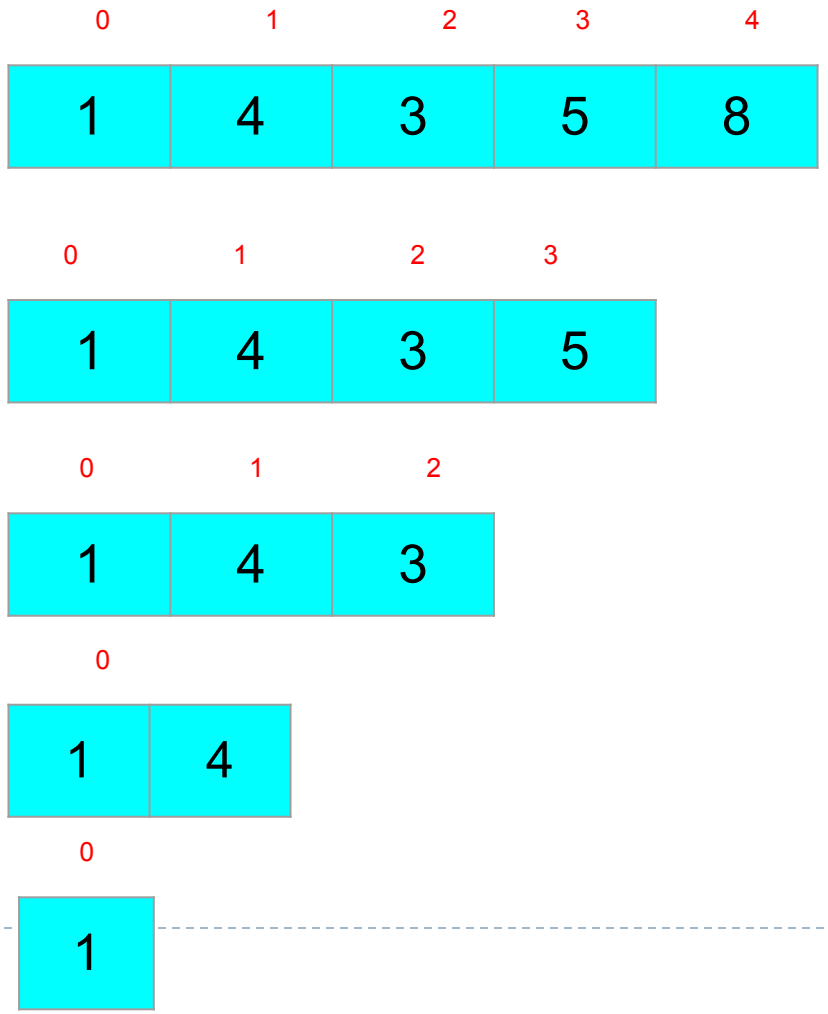


Invertir una lista



Invertir una lista

[8,5,3,4,1] -> [1,4,3,5,8]



Ejemplo: Invertir una lista

```
def reverse1(a: list) -> None:
    if a is None or not isinstance(a, list):
        return
    if len(a) >= 1:
        aux = a.pop(0)
        reverse1(a)
        a.append(aux)
```

Invertir una lista

- Piensa en una solución que no elimine ningún elemento de la lista.
- Sí está permitido intercambiar elementos en el array.

Invertir una lista (enfoque II)

Caso recursivo: intercambiar cambiando el primer y último elemento, e invertir el resto del array recursivamente.

Caso Base: la lista no tiene ningún elemento.

Solución

Ejemplo: Máximo común divisor

- Encontrar el entero mayor d que divida a a y b
- Algoritmo de Euclides algorithm (300 BCE).

$$mcd(a,b) = \begin{cases} a & \text{if } b=0 \\ gpd(b,a\%b) & \text{otherwise} \end{cases}$$

gcd(4032,1272) = gcd(1272, 216) = gcd(216,192) = gcd(192,24) = gcd(24,0) =0

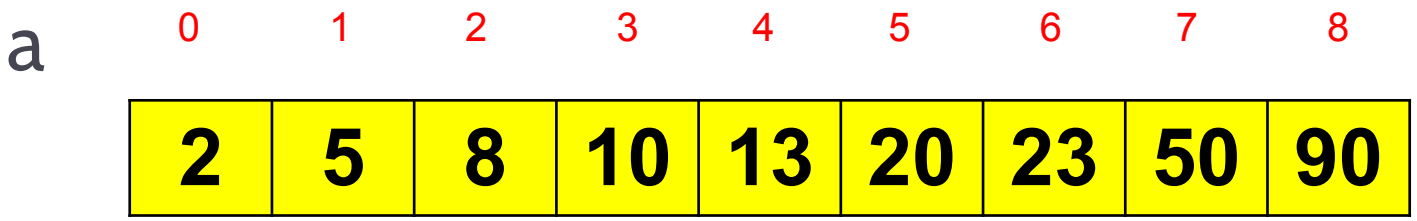
Recursión lineal: Máximo común divisor

```python
def gcd(a: int, b: int) -> int:
    """returns the greatest common divisor of a and b"""
    if not isinstance(a, int) or a < 0 or not isinstance(a, int) or b < 0:
        return None  # a, b must be >=0
    # suppose a > b, if a < b, we change
    if a < b:
        a, b = b, a
    return _gcd(a, b)


def _gcd(a: int, b: int) -> int:
    if b == 0:
        return a
    else:
        return _gcd(b, a % b)
```

Búsqueda binaria

Entrada: un array ordenado de enteros y un número x (por ejemplo, x=23)



a[mid]>x?

1) x=a[mid], Encontrado!!!
2) x<a[mid], buscar desde start hasta mid-1
3) x>a[mid], buscar desde mid+1 hasta end

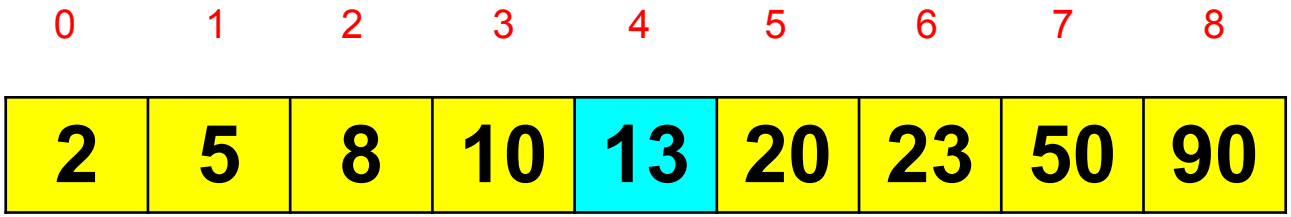
Búsqueda binaria

Función recursiva que recibe una lista ordenada de enteros y un número entero, y devuelve True si el el número existe en la lista, y False en otro caso.
Opciones:

- Puedes usar slicing.
- No puedes usar slicing.
- Un problema más difícil sería devolver el índice donde se encuentra el número, o -1 (si no existe en la lista).

Soluciones

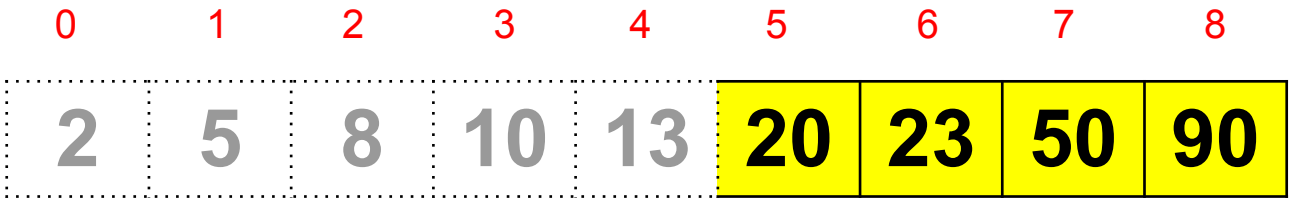
Búsqueda binaria, x = 23



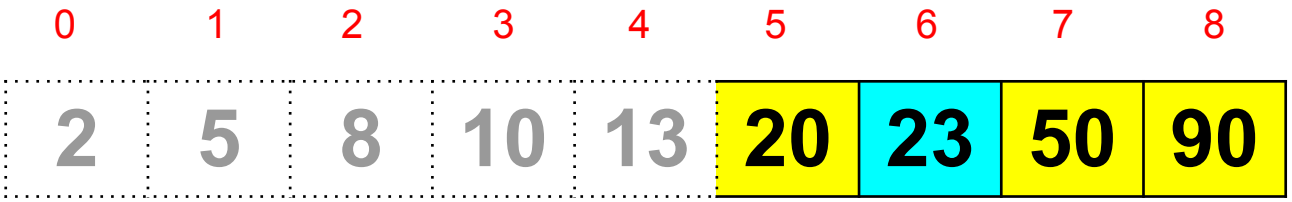
start mid end

a[mid]<23 -> start=mid+1=5, end=8

Búsqueda binaria, x = 23

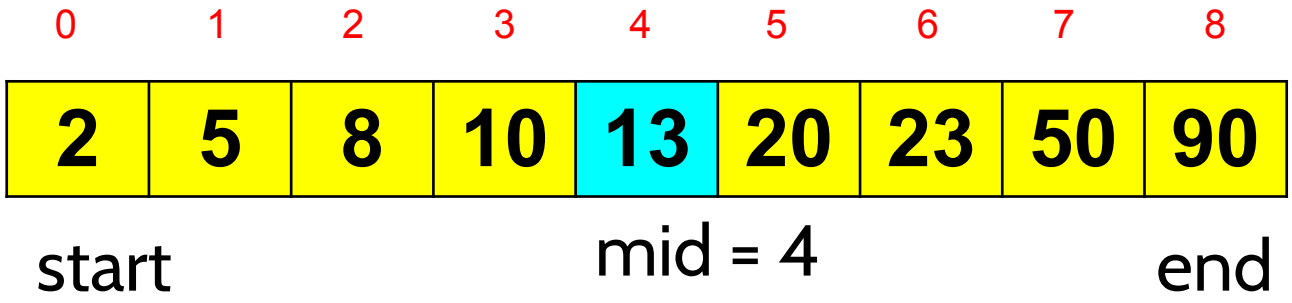


mid = (start + end)//2 = (5 + 8)//2 = 13 // 2= 6



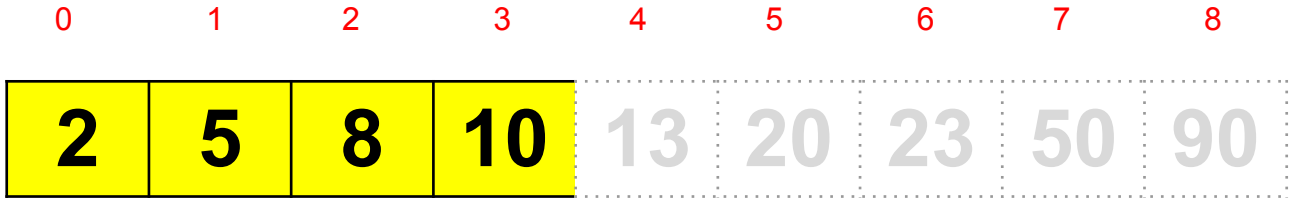
a[mid]=x, Encontrado!!!

Búsqueda binaria, x = 10



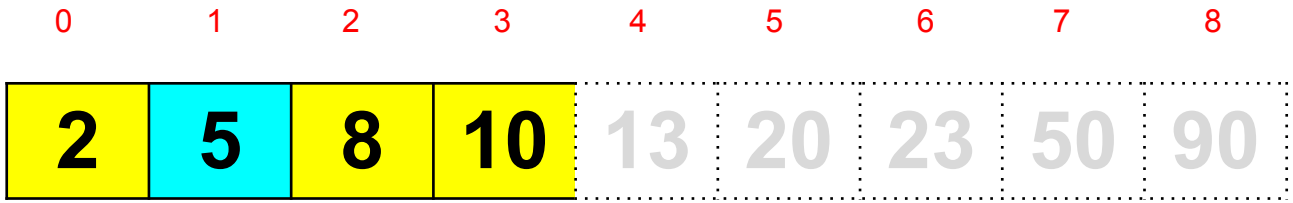
a[mid] > 10 -> start=0 , end=mid-1=3

Búsqueda binaria, x = 10



start end

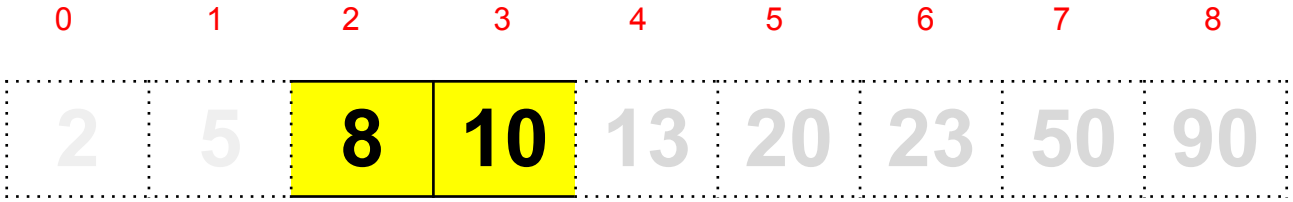
mid = (start + end)//2 = (0+3)//2 = 3 // 2= 1



start mid end

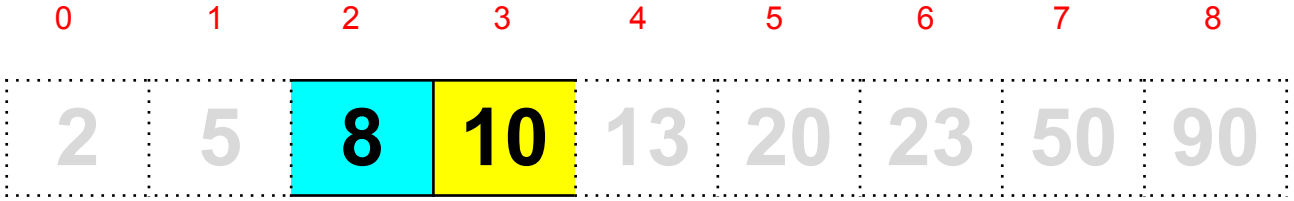
a[mid] < 10 -> start=mid+1=2, end=3

Búsqueda binaria, x = 10



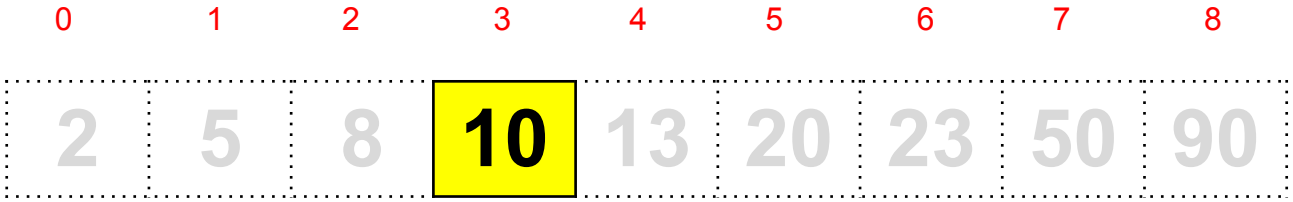
start end

mid = (start + end)//2 = (2+3)//2 = 4 // 2= 2

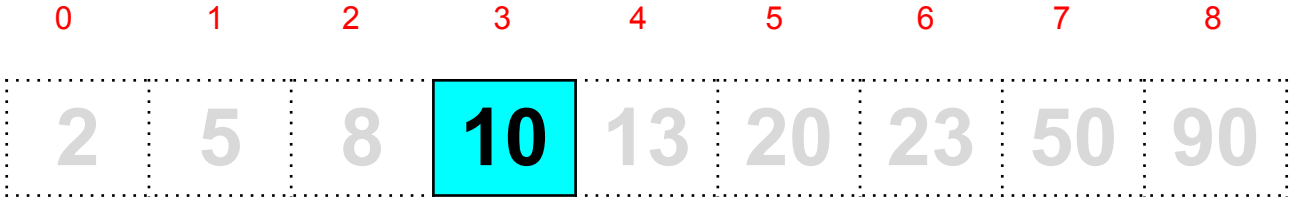


a[mid] < 10 -> start=mid+1=3, end=3

Búsqueda binaria, x = 10

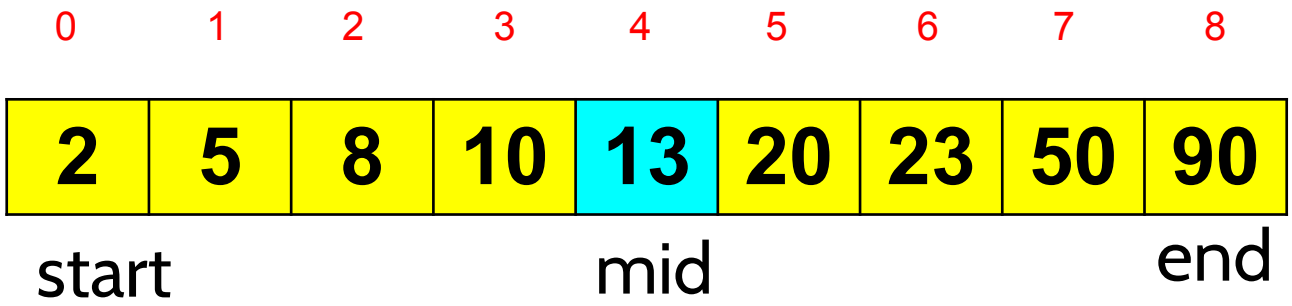


mid = (start + end)//2 = (3+3)//2 = 6 // 2= 3

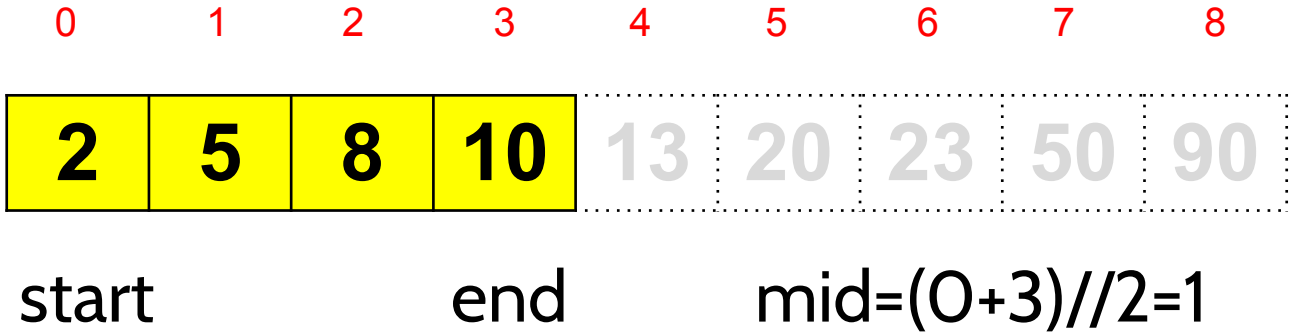


10==a[mid] -> Encontrado!!!

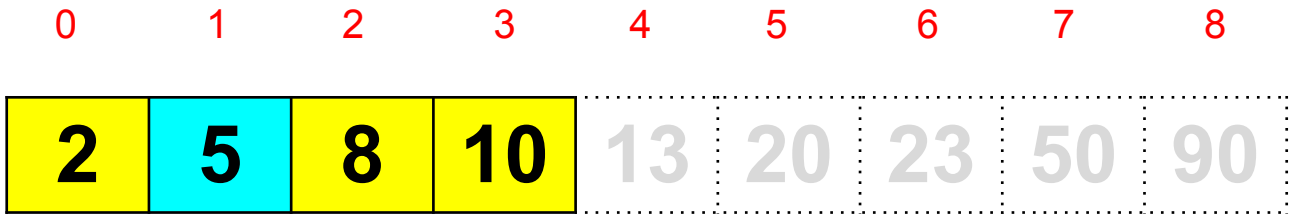
Ejemplo: Búsqueda binaria, x = 7 (no existe)



x<a[mid] -> end=mid-1

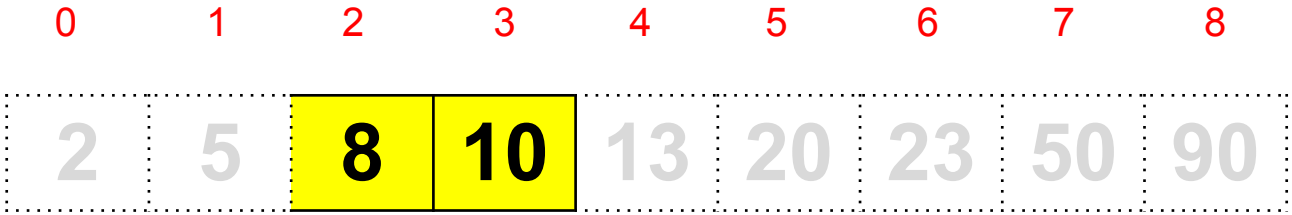


Ejemplo: Búsqueda binaria, x = 7 (no existe)



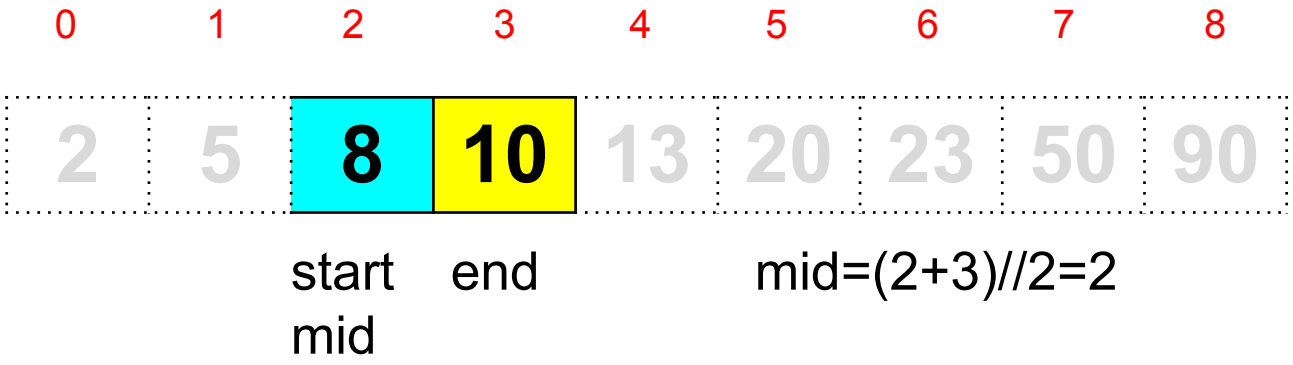
start mid end mid=(0+3)//2=1

x>A[mid] -> start=mid+1



start end mid=(2+3)//2=2

Ejemplo: Búsqueda binaria, x = 7 (no existe)



x<A[mid] -> end=mid-1=1, start=2

end<start!!!, significa que hemos revisado todas las particiones posibles sin encontrar el elemento. Debemos devolver False.

Recursión Binaria

- En alguno de los casos recursivos, se realizan dos llamadas recursivas.
- Estudiará dos ejemplos:
  - Números Fibonacci
  - Suma una lista de números usando recursión binaria.

Recursión Binaria: Números Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$Fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fib(n-1) + Fib(n-2) & \text{if } n>1 \end{cases}$$

Recursión Binaria: Números Fibonacci

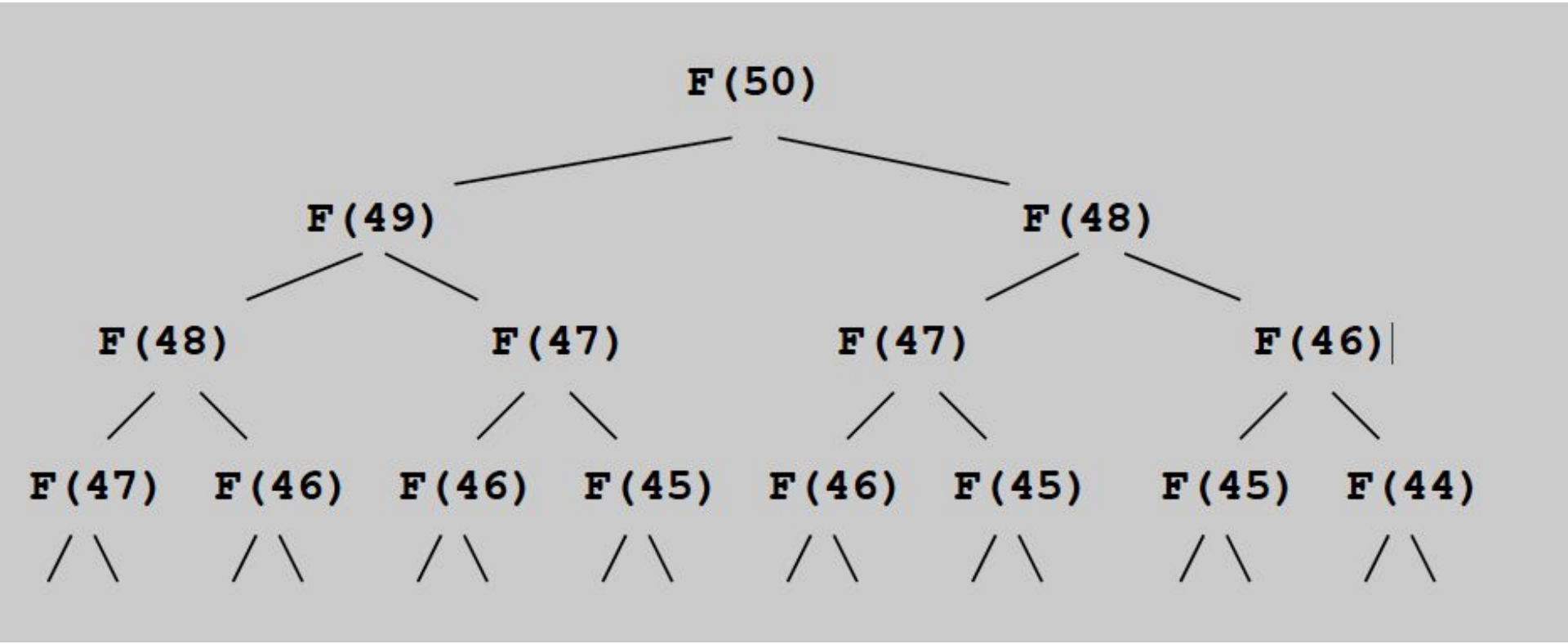
```
def fibo(n: int) -> int:
    if n < 0 or not isinstance(n, int):
        return None

    if n <= 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

¿Es una forma eficiente de calcular fibo(50)?

Recursión Binaria: Números Fibonacci

No!!! Este código es espectacularmente ineficiente: $O(2^n)$



Una función más eficiente para Fibonnaci

```
def fibo2(n: int) -> (int, int):
    """This implementation is more efficient than the previous one.
    It returns a pair (x, y), where x is the Fibonacci number for n-1, and y for n"""
    if n < 0 or not isinstance(n, int):
        return None

    if n <= 1:
        # returns the fibonacci for 0 and for 1
        return 0, n
    else:
        # a is the fibonacci number for n-2, b is the Fibonacci number for n-1
        a, b = fibo2(n-1)
        # Now, we have to return the Fibonacci number for n-1, and the Fibonacci number for n,
        # which is a + b
        return b, a + b
```

Recursión Binaria: sumar una lista de números

¿Cómo calcular la suma de los números de una lista usando recursión binaria?

Idea!!!: dividir en dos mitades, calcular la suma de la primera parte, calcular la suma de la segunda parte, y sumar estos resultados

Recursión Binaria: sumar una lista de números

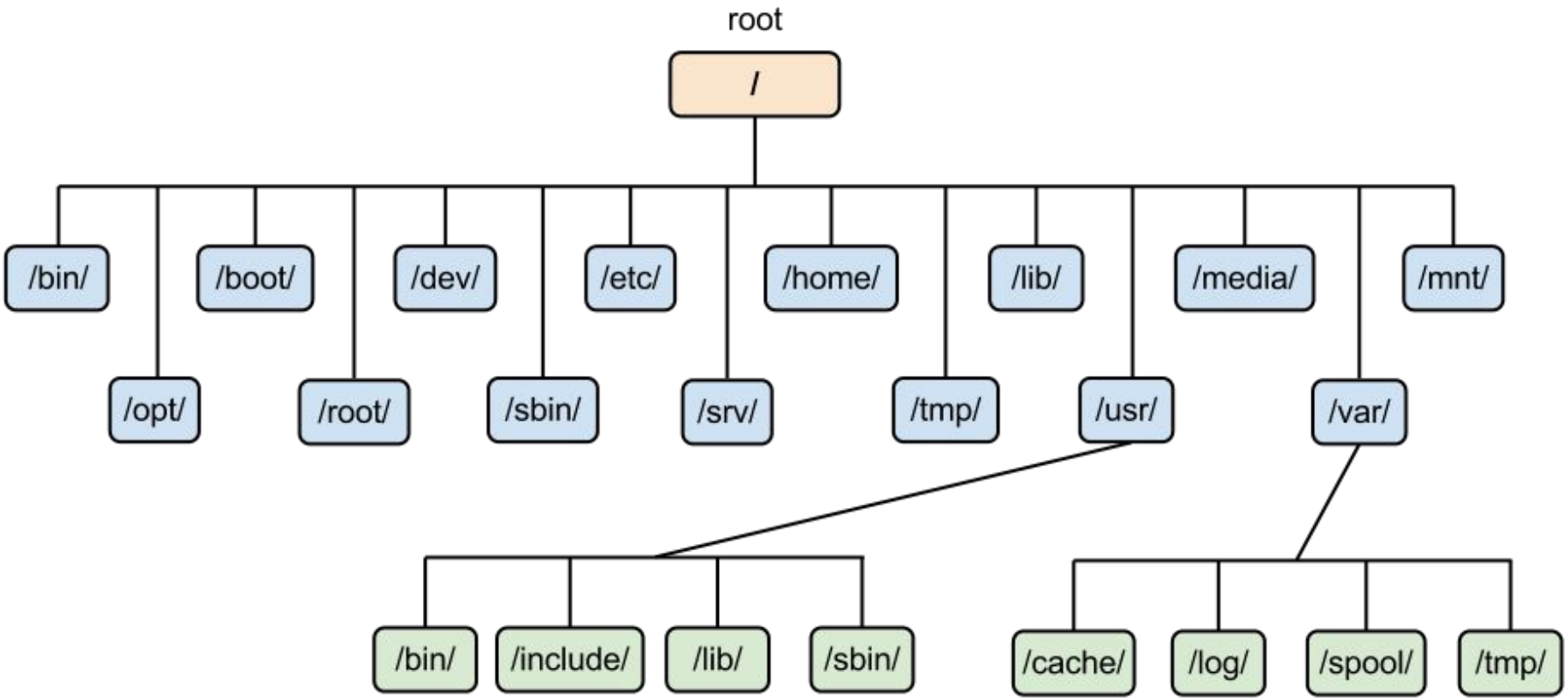
```
def binary_sum(a: list) -> int:
    """returns the sum of the elements in a by using
    binary recursion"""
    if a is None or not isinstance(a, list):
        return None
    if len(a) == 0:
        return 0
    else:
        m = len(a) // 2
        return binary_sum(a[0:m]) + a[m] + binary_sum(a[m+1:])
```

Recursión Múltiple

- En al menos uno de los casos recursivos, se realizará tres o más llamadas recursivas.

- Ejemplo: explorar el sistema de archivos puede resolverse usando recursión múltiple.

Recursión Múltiple: explorar sistema de archivos



Recursión Múltiple: explorar sistema de archivos

¿Cómo calcular el espacio en disco de un determinado directorio?. Te dejamos este pseudocódigo, para que trates de implementarlo en Python.

```
Algorithm DiskUsage(path):
    Input: A string designating a path to a file-system entry
    Output: The cumulative disk space used by that entry and any nested entries
    total = size(path)                    {immediate disk space used by the entry}
    if path represents a directory then
        for each child entry stored within directory path do
            total = total + DiskUsage(child)          {recursive call}
    return total
```

Solución

Índice

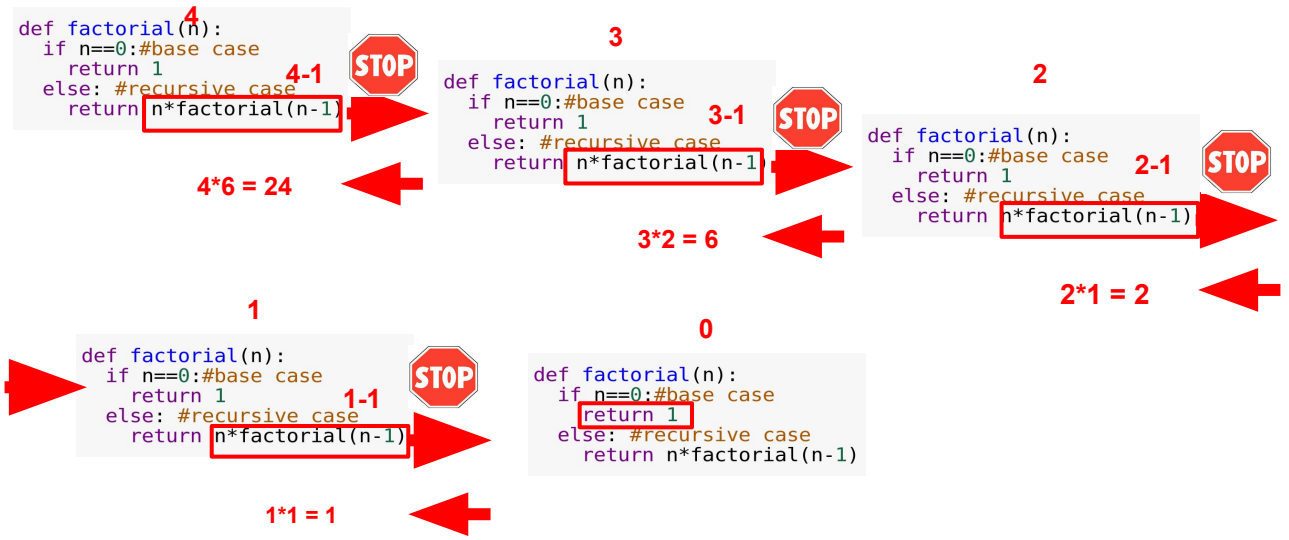
1) ¿Qué es recursión?
2) Algunos ejemplos de recursión.
3) Tipos de recursión.
4) **Iteración y Recursión**

Iteración y Recursión

- Un bucle también es un proceso repetitivo.
- Un método recursivo es una solución matemáticamente más elegante (y requiere menos líneas de código) que un bucle.
- A menudo, cuando implementamos fórmulas matemáticas (que suelen definirse con recursión), es mucho más fácil conseguir una solución recursiva que iterativa.
    - Calcular el tamaño o altura de un árbol (lo veremos en el tema 5).

Iteración y Recursión

- La recursión implica un mayor tiempo de ejecución y gasto de memoria principal, porque todas los resultados parciales (junto con los argumentos y variables locales de cada llamada recursiva) son almacenados en memoria en tiempo de ejecución.



Iteración y Recursión

- Si se encadenan muchas llamadas recursivas, es posible que se produzca un desbordamiento de memoria. Por ejemplo, calcula fibo(n) para n>50

Iteración y Recursión

- **Todo algoritmo recursivo puede expresarse como iterativo y viceversa**.
- Sin embargo, según las condiciones del problema a resolver podrá ser preferible utilizar la solución recursiva o la iterativa.

To iterate is human, to recurse, divine