Grado en Ciencia e Ingeniería de Datos, 2018-2019

# Unit 4. Recursion

## Algorithms and Data Structures (ADS)

Author: Isabel Segura-Bedmar

# Index

- **What is recursion?**
- Some examples of recursion
- Types of recursion
- Iteration versus Recursion

# What is recursion?

- A way to achieve repetition.
- A method calls itself.
- Closely related to mathematical induction.
- Some data structures can have a recursive structure (nodes or trees)

# What is recursion?

- Every recursive method has two parts:
  - **BASE CASE(S)**: case(s) so simple that they can be solved directly.
  - **RECURSIVE CASE(S)**: more complex and make use of recursion to:
    - Break the problem to smaller subproblems and
    - Combine into a solution to the larger problem.

# What is recursion?

The three laws of recursion:

1. A recursive algorithm must have at least one **base case**.
2. A recursive algorithm must call itself, recursively.
3. A recursive algorithm must move toward the base case.

# Index

# Example 1: Factorial function

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

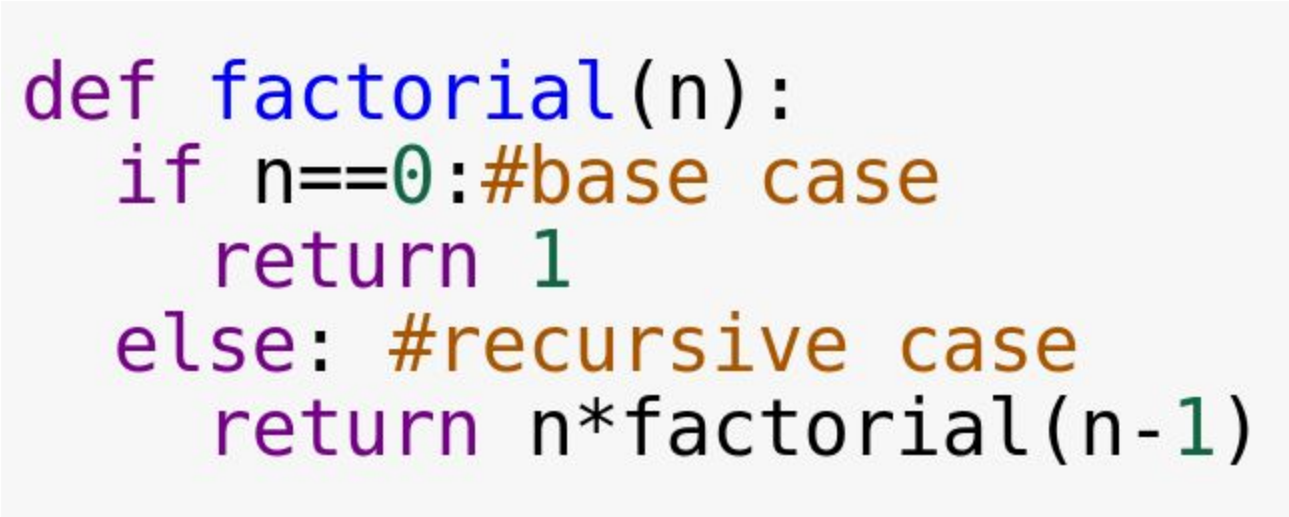
4!=4.3.2.1=24

4!=4.(3.2.1)=4.3!

**Recursive definition**

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

# Example 1: Implementation of factorial function

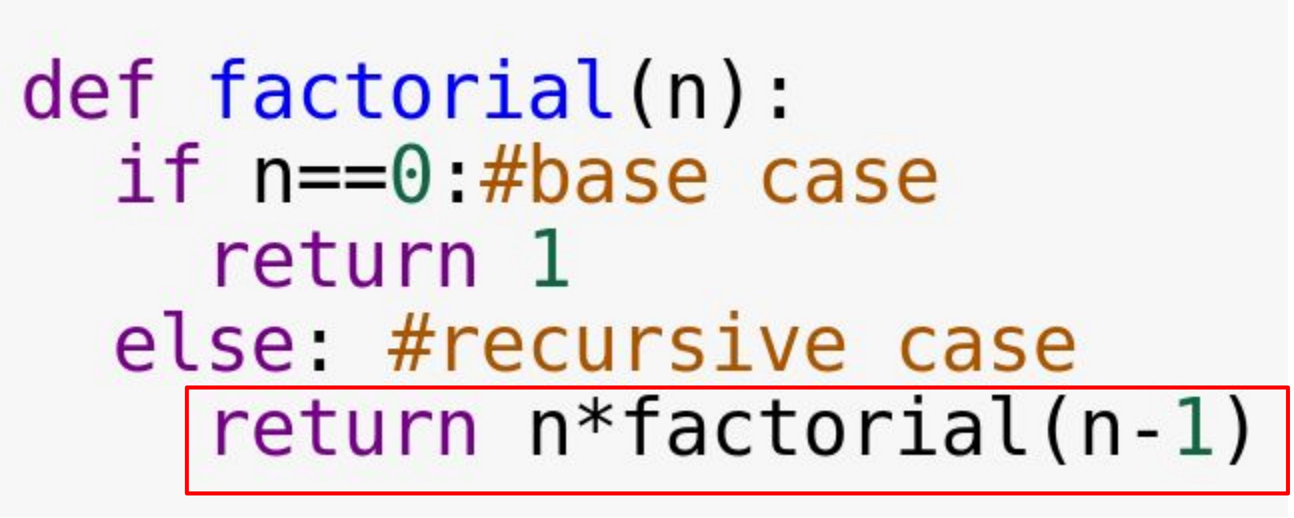


```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

# Example 1: Tracing factorial

*First Call*

factorial(4) → 4 * factorial(3)



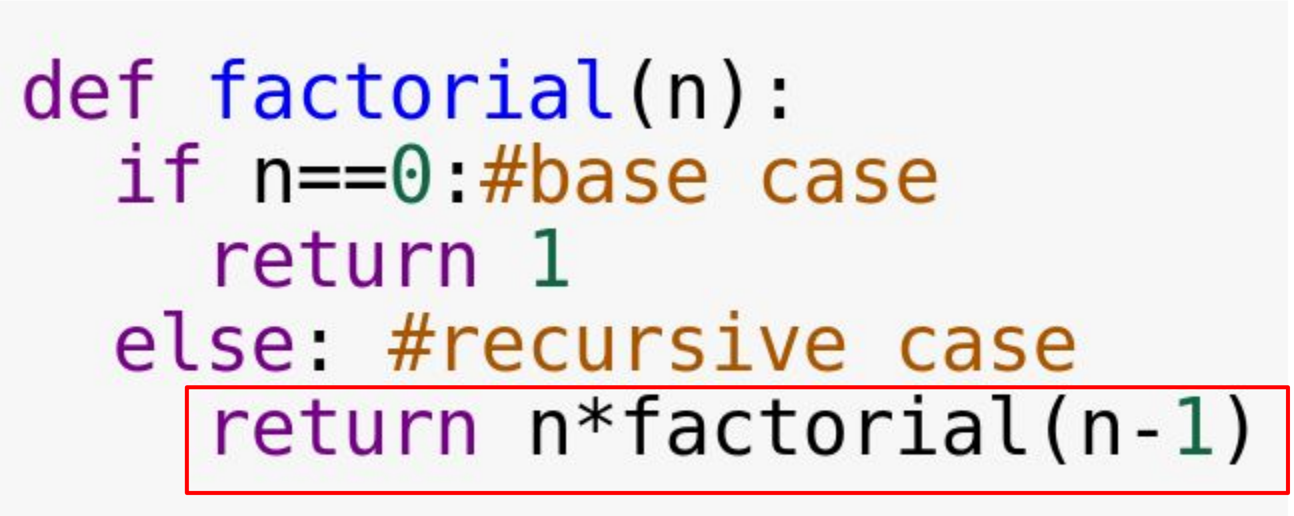
```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

# Example 1: Tracing 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)
```

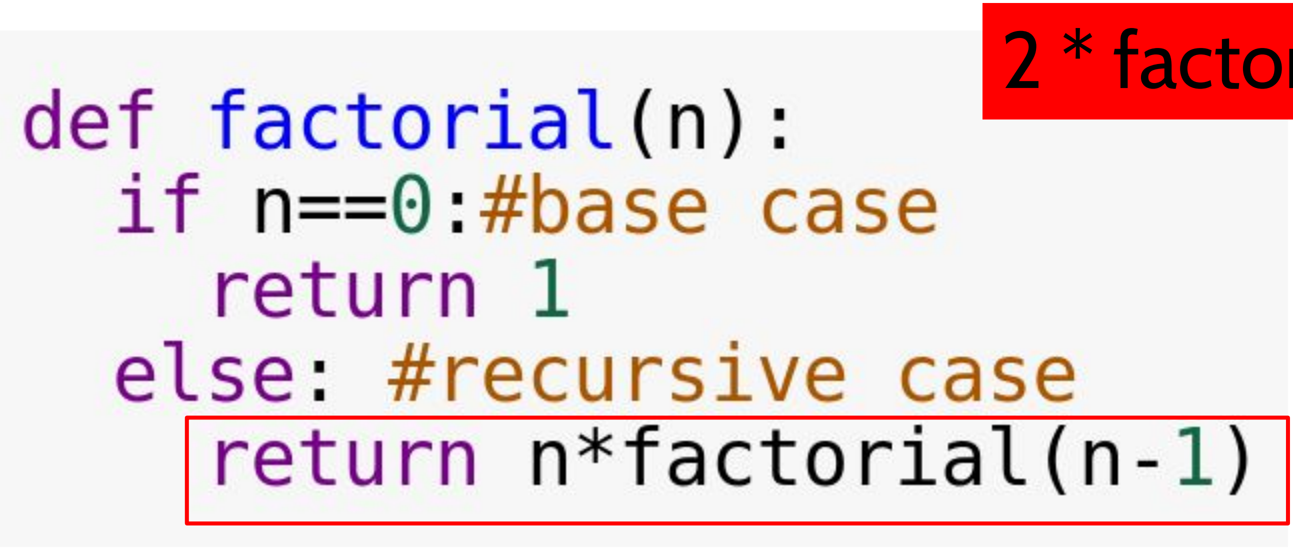
# Example 1: Tracing factorial

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)
```

# Example 1: Tracing factorial

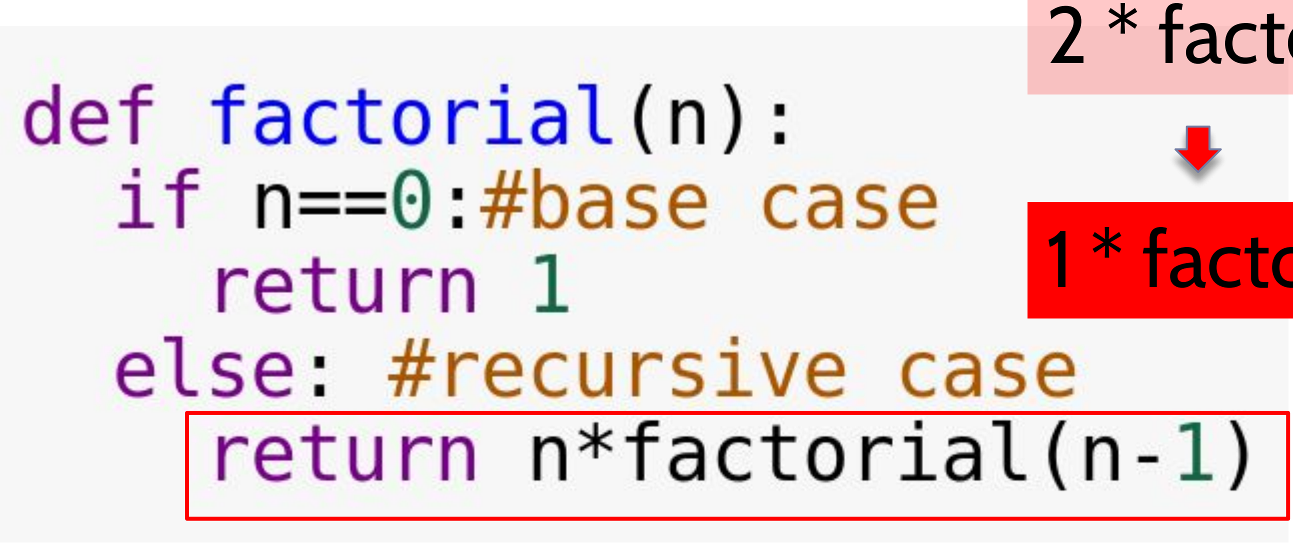
factorial(4)

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)
```

# Example 1: Tracing factorial

factorial(4)

4 * factorial(3)

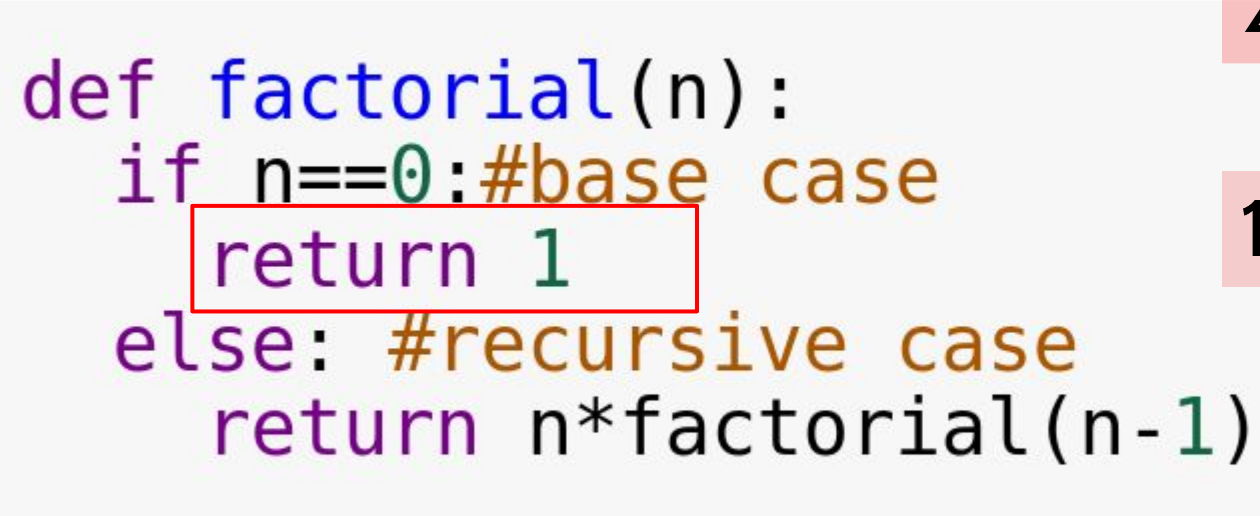
3 *factorial(2)

2 * factorial(1)

1 * factorial(0)

1

Base case



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

# Example 1: Tracing factorial

factorial(4)

4 * factorial(3)

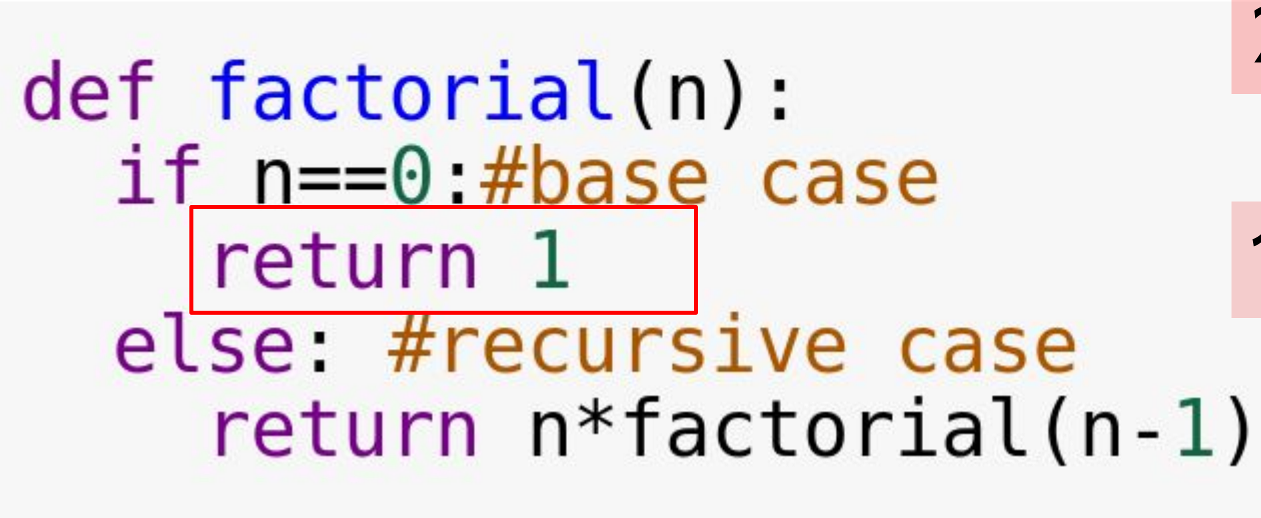
3 *factorial(2)

2 * factorial(1)

1 * factorial(0)

1

1

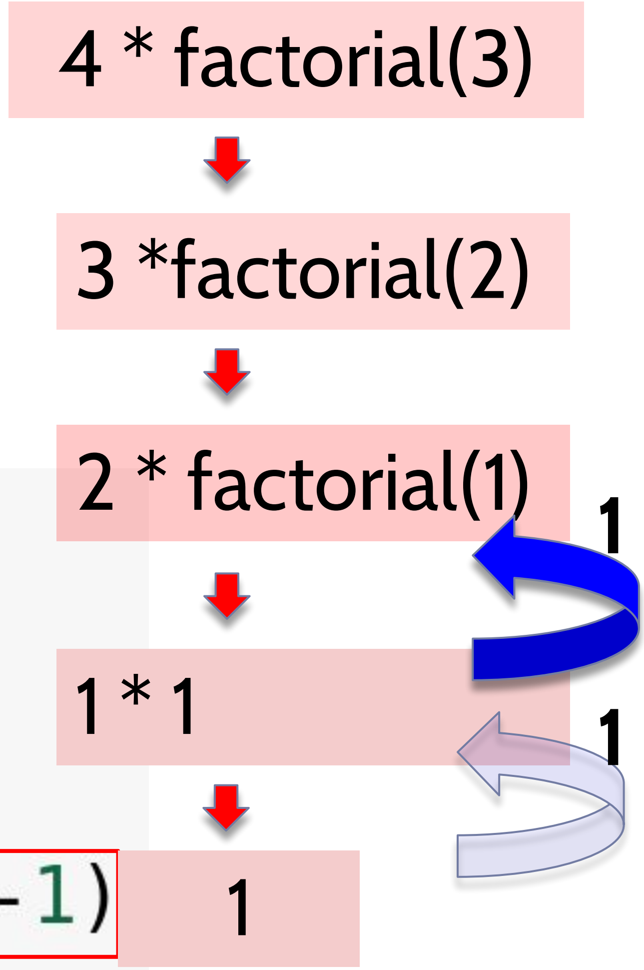


```
def factorial(n):
  if n==0:#base case
    return 1
  else: #recursive case
    return n*factorial(n-1)
```

# Example 1: Tracing factorial

factorial(4)

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



4 * factorial(3)

3 *factorial(2)

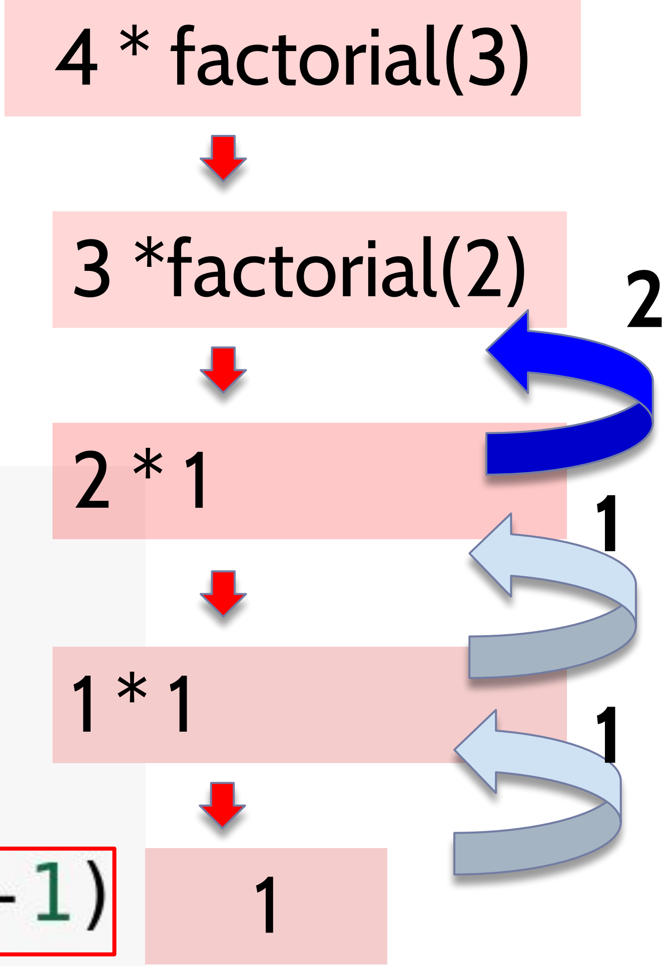
2 * factorial(1)

1 * 1

1

# Example 1: Tracing factorial

factorial(4)



4 * factorial(3)

3 *factorial(2)

2 * 1

1 * 1

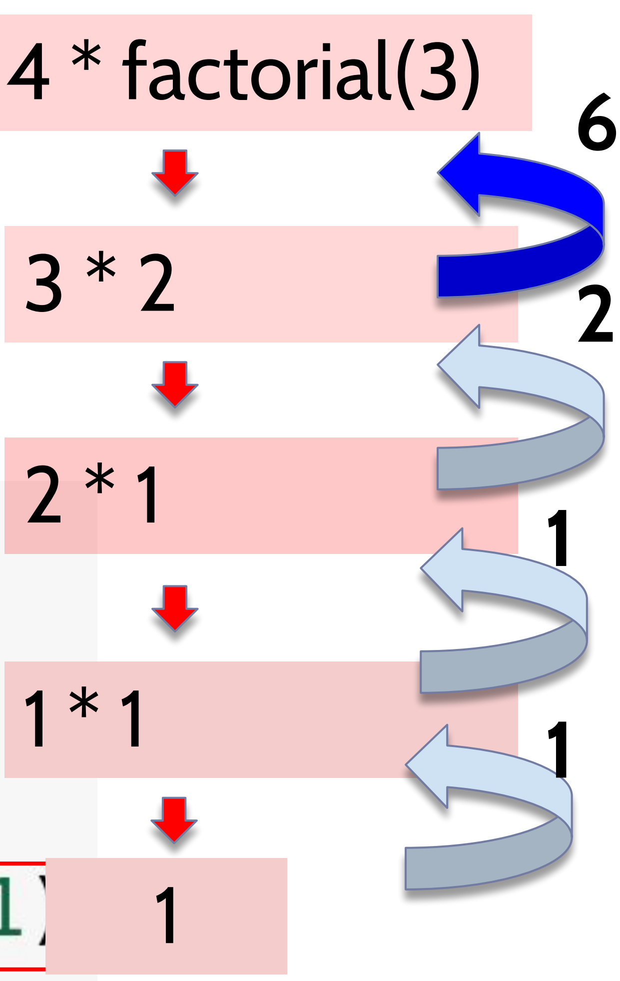
1

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

# Example 1: Tracing factorial

factorial(4)

*First Call*



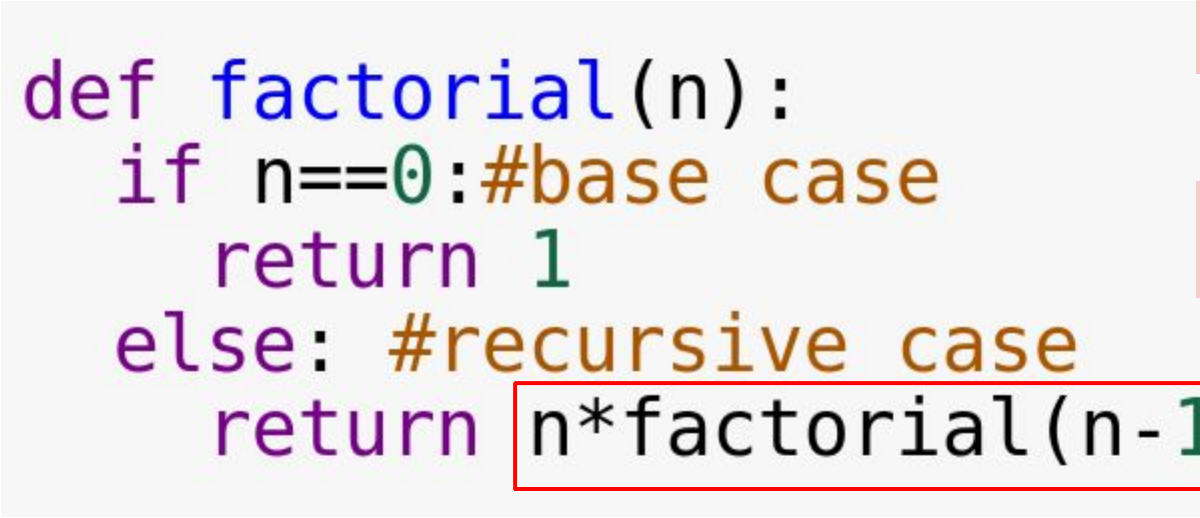
4 * factorial(3)

3 * 2

2 * 1

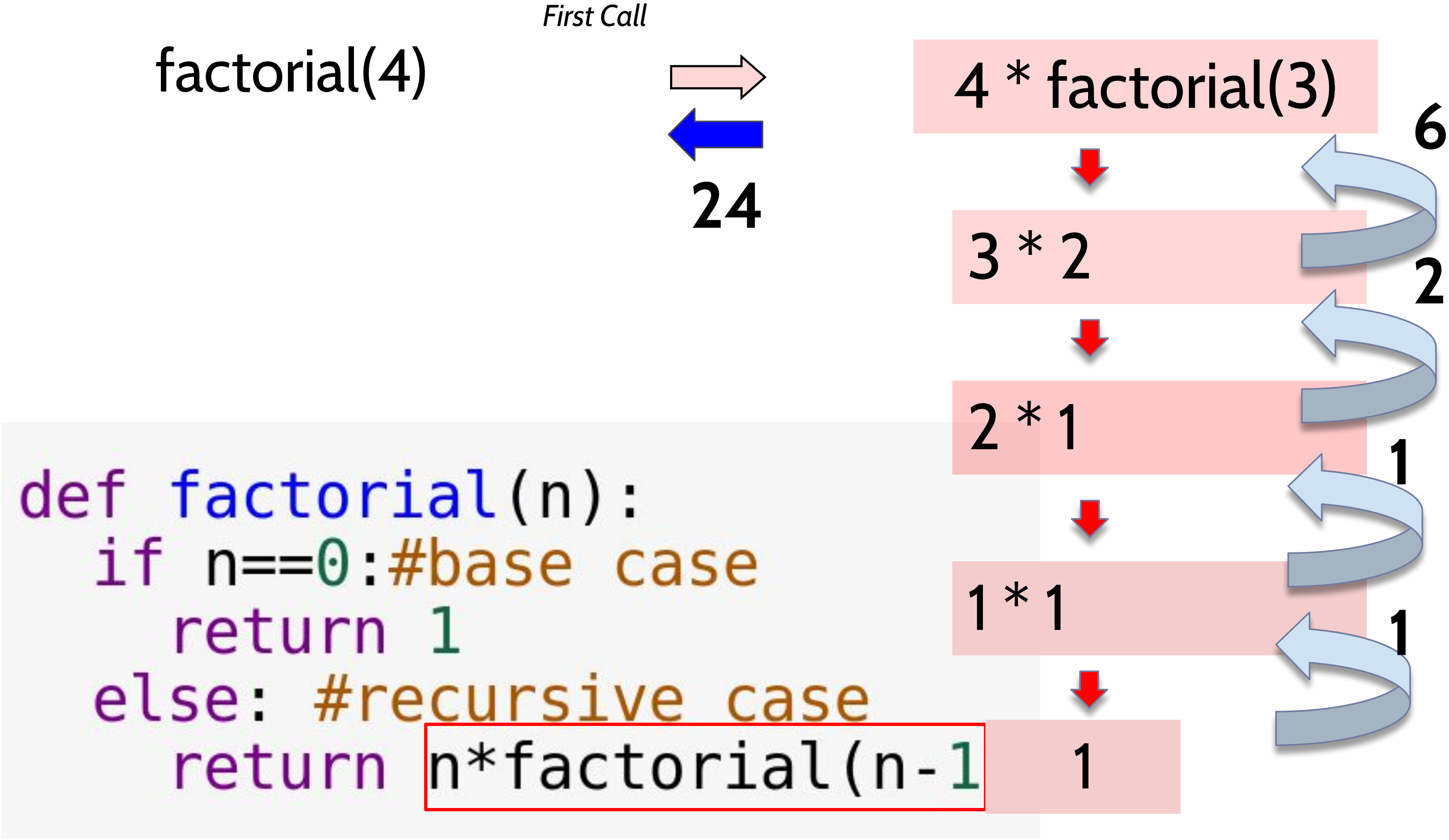
1 * 1

1



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

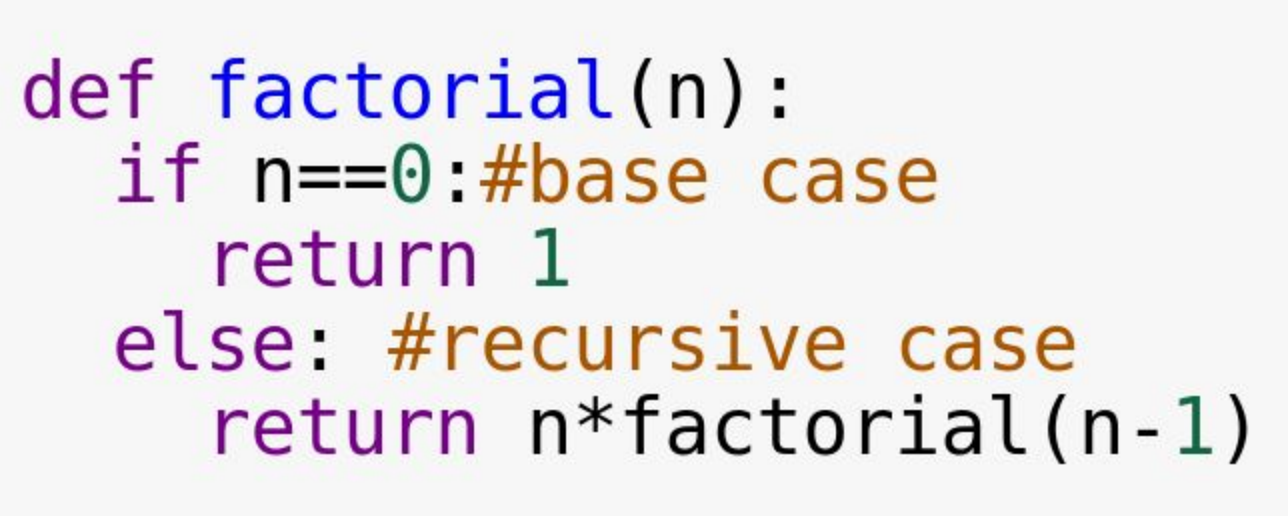
# Example 1: Tracing factorial



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

# Example 1: Analysis of factorial function

Big-O function for factorial function?



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

There is n+1 calls (each of which accounts for O(1) operations).
Therefore, factorial is O(n)

# Index

- What is recursion?
- Some examples of recursion
  - Factorial
  - **Multiplication by addition**
  - Binary search
- Types of recursion
- Iteration versus Recursion

# Example 2: Multiply 2 numbers using addition

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

# Example 2: Multiply 2 numbers using addition

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

```
def multiplyRec(x, y) :
```

First, think about the base case(s)???

# Example 2: Multiply 2 numbers using addition

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

```
def multiplyRec(x, y) :
    if y==0:
        return 0
```

Right!!!. Now, think about the recursive case(s)

# Example 2: Multiply 2 numbers using addition

5 x 3 = 15 = 5 + 5 + 5

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

Yes, you got it!!!

# Index

- What is recursion?
- Some examples of recursion
  - Factorial
  - Multiplication by addition
  - **Binary search**
- Types of recursion
- Iteration versus Recursion

# Example 3: Binary search

Input: a <u>sorted</u> array of integers and a number

x = 23

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | **2** | **5** | **8** | **10** | **13** | **20** | **23** | **50** | **90** |

start

mid = (start + end) / 2

end

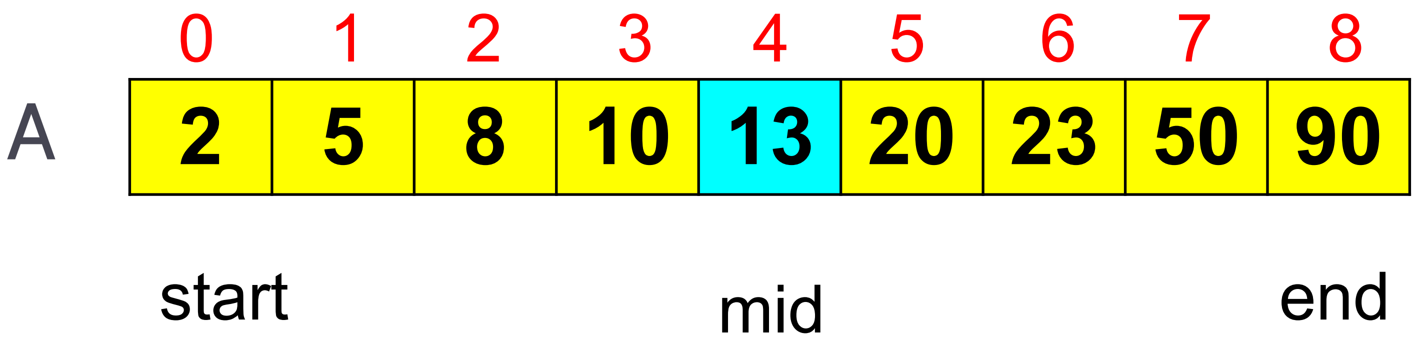
A[mid]>x?

1) x=A[mid], Found!!!
2) x<A[mid], search from start to mid-1
3) x>A[mid], search from mid+1 to start

# Example 3: Binary search

Input: a sorted array of integers and a number

x = 23



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | **2** | **5** | **8** | **10** | **13** | **20** | **23** | **50** | **90** |

start

mid

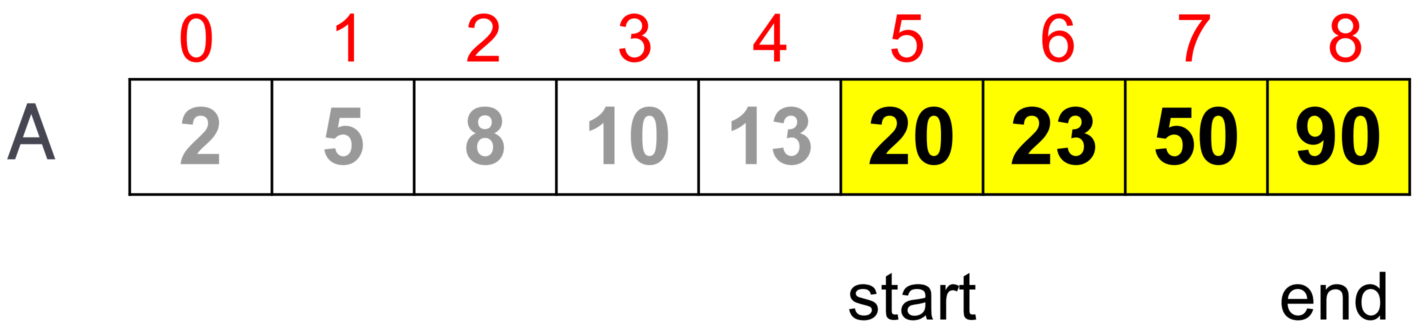
end

A[mid]<23 -> search from 5 to 8

# Example 3: Binary search

Input: a <u>sorted</u> array of integers and a number

x = 23



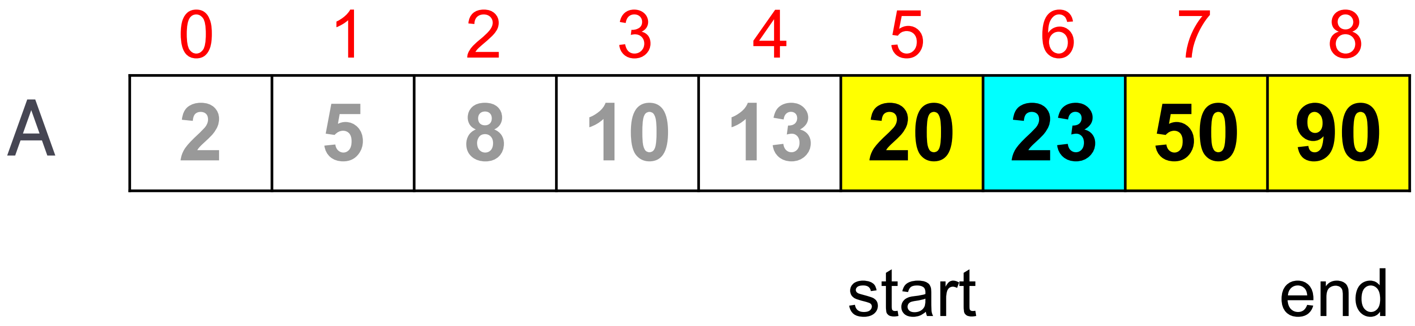
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | 2 | 5 | 8 | 10 | 13 | **20** | **23** | **50** | **90** |
|   |   |   |   |   |   | start |   |   | end |

A[mid]<23 -> search from 5 to 8

# Example 3: Binary search

Input: a sorted array of integers and a number

x = 23



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | 2 | 5 | 8 | 10 | 13 | **20** | **23** | **50** | **90** |
|   |   |   |   |   |   | start |   |   | end |

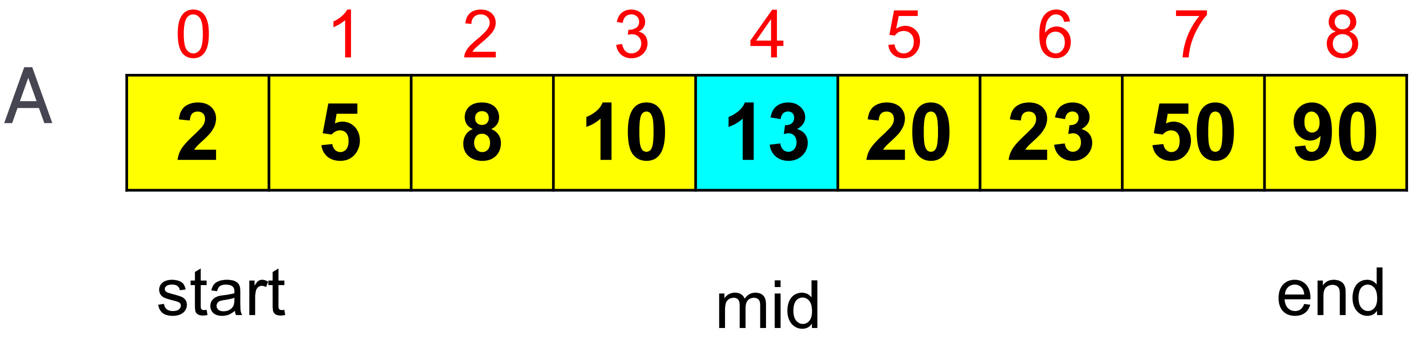
mid = (start + end) = 5 + 8 = 13 / 2= 6

A[mid]==23 -> Found it!!!

# Example 3: Binary search

Input: a sorted array of integers and a number

x = 7 (which does not exist in the list)



A

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| **2** | **5** | **8** | **10** | **13** | **20** | **23** | **50** | **90** |

start

mid

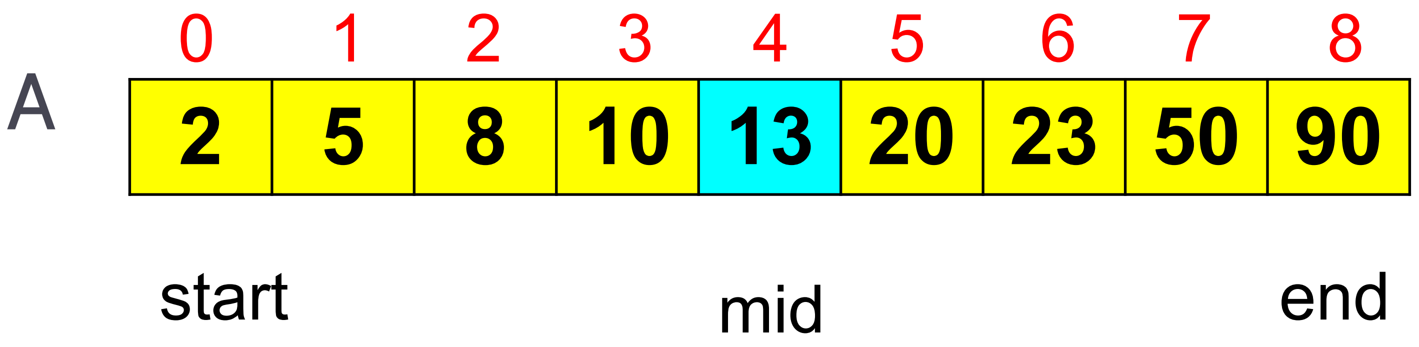
end

What happens if the array does not contain the target?

# Example 3: Binary search

Input: a <u>sorted</u> array of integers and a number

x = 7



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | **2** | **5** | **8** | **10** | **13** | **20** | **23** | **50** | **90** |

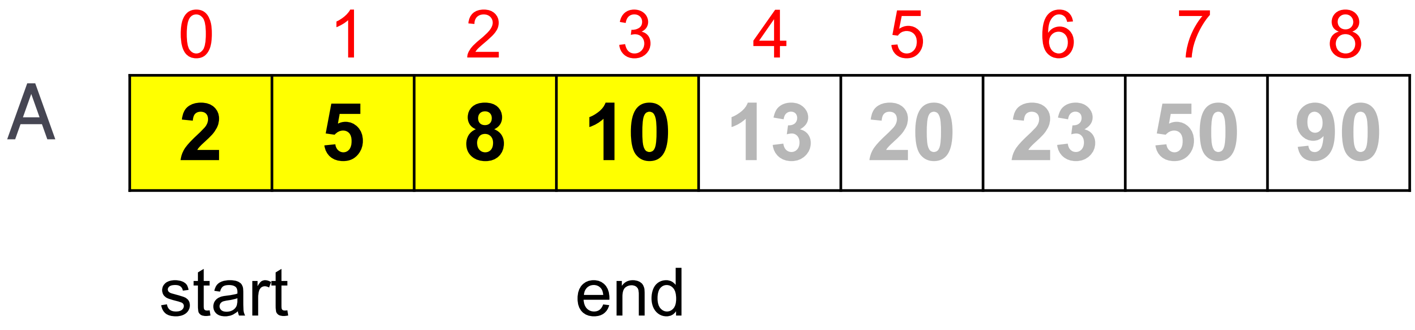
start mid end

A[mid]>7 -> search from 0 to 3

# Example 3: Binary search

Input: a <u>sorted</u> array of integers and a number

x = 7



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | **2** | **5** | **8** | **10** | 13 | 20 | 23 | 50 | 90 |

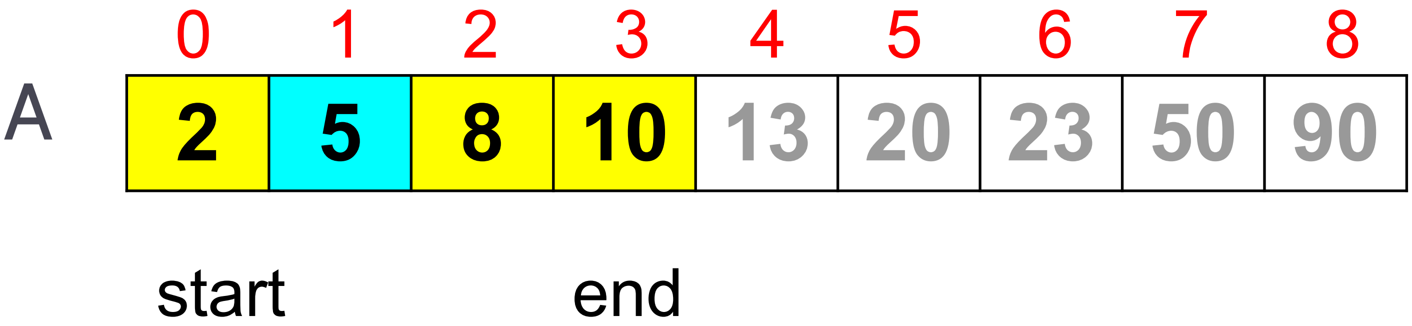
start end

mid=(0+3)/2=1

# Example 3: Binary search

Input: a sorted array of integers and a number

x = 7



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | **2** | **5** | **8** | **10** | 13 | 20 | 23 | 50 | 90 |

start

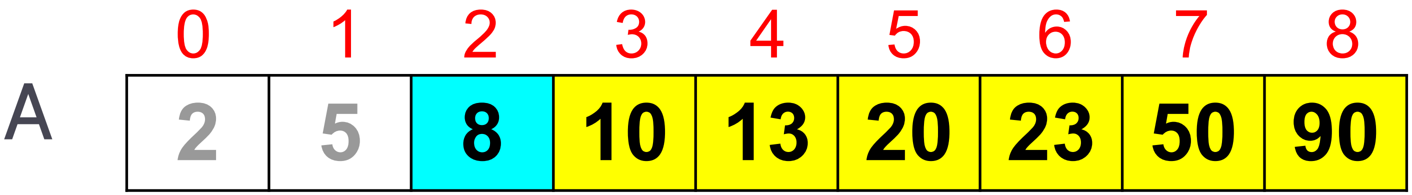
end

A[mid]<7 -> search 2 to 3

# Example 3: Binary search

Input: a <u>sorted</u> array of integers and a number

x = 7



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | 2 | 5 | **8** | **10** | **13** | **20** | **23** | **50** | **90** |

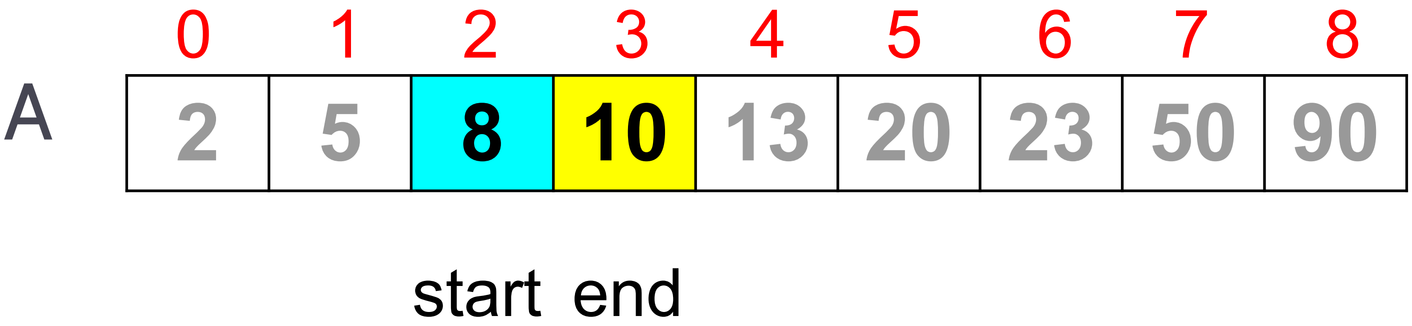
start end

mid=(2+3)/2=2

# Example 3: Binary search

Input: a sorted array of integers and a number

x = 7



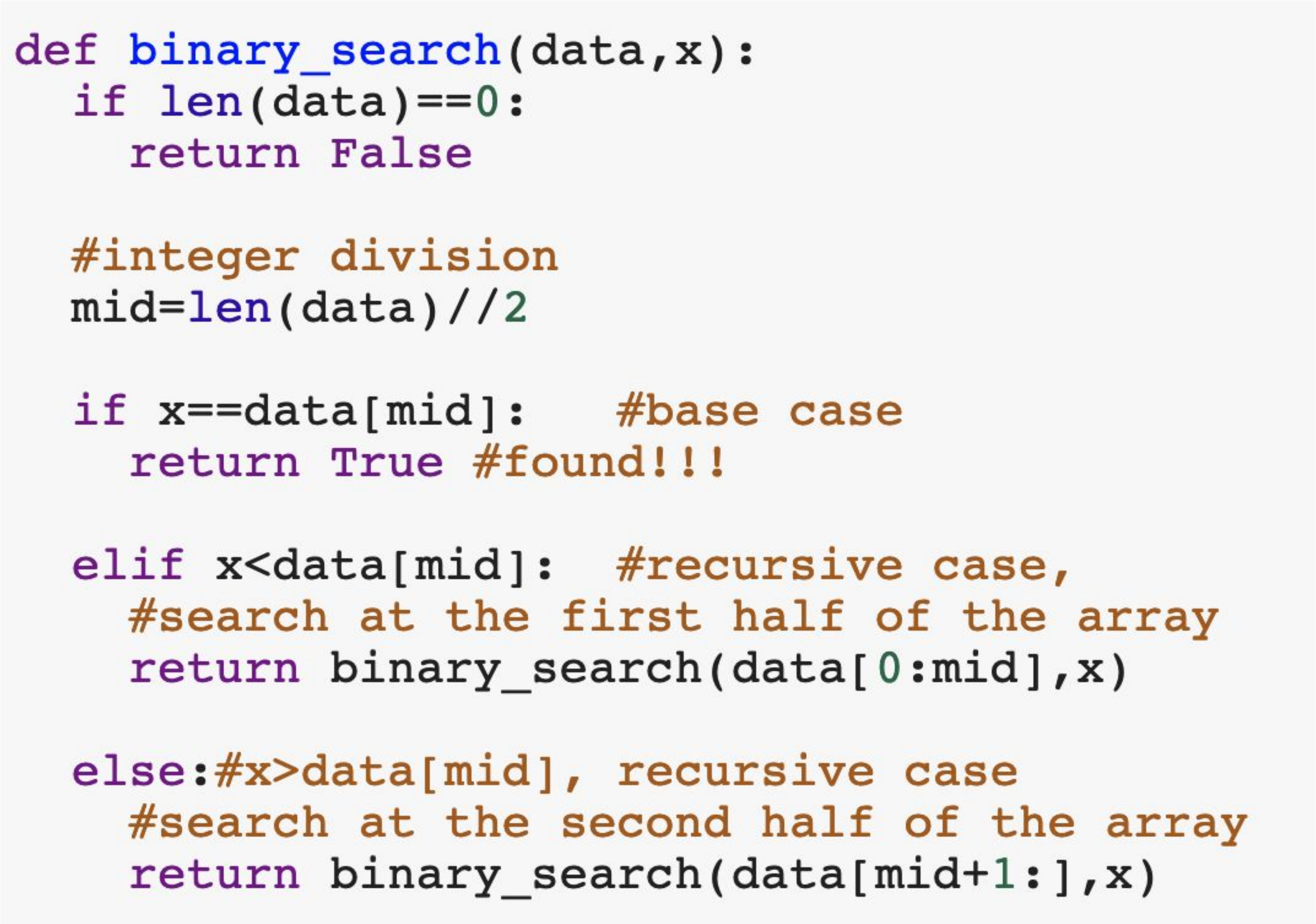
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | 2 | 5 | **8** | **10** | 13 | 20 | 23 | 50 | 90 |

start end

A[mid]=8>7 -> start=2 , end=mid-1=2-1=1

start>end!!! : the array does not contain it!!!

# Example 3: Implementation of Binary search



```
def binary_search(data,x):
  if len(data)==0:
    return False

  #integer division
  mid=len(data)//2

  if x==data[mid]:   #base case
    return True #found!!!

  elif x<data[mid]:  #recursive case,
    #search at the first half of the array
    return binary_search(data[0:mid],x)

  else:#x>data[mid], recursive case
    #search at the second half of the array
    return binary_search(data[mid+1:],x)
```

# Example 3: Analysis of Binary search

- Initially, the number of candidates is n;
- after the first call in a binary search, it is at most n/2;
- after the second call, it is at most n/4;

  ….
- after the jth call, the number of candidate entries remaining is at most $n/2^j$ .
- In the worst case, the function stops when there are not more candidate entries

# Example 3: Analysis of Binary search

The maximum number of possible recursive calls is the smallest integer r such that

$$\frac{n}{2^r} < 1.$$

$$r = \lfloor \log n \rfloor + 1$$

O(log n)

# Index

# Types of recursion

1. **Linear recursion**: a recursive call may make at most one new recursive call.
2. **Binary recursion**: a recursive call may make two new recursive calls.
3. **Multiple recursion**: a recursive call may make three or more recursive calls.

# Index

# Types of recursion: Linear recursion

- We already see some examples: factorial, binary search, etc.
- Now, we will study more examples:
  - Computing the sum of a sequence of integers.
  - Reversing an array
  - Computing powers

# Types of recursion: Linear recursion

Sum a list of numbers:

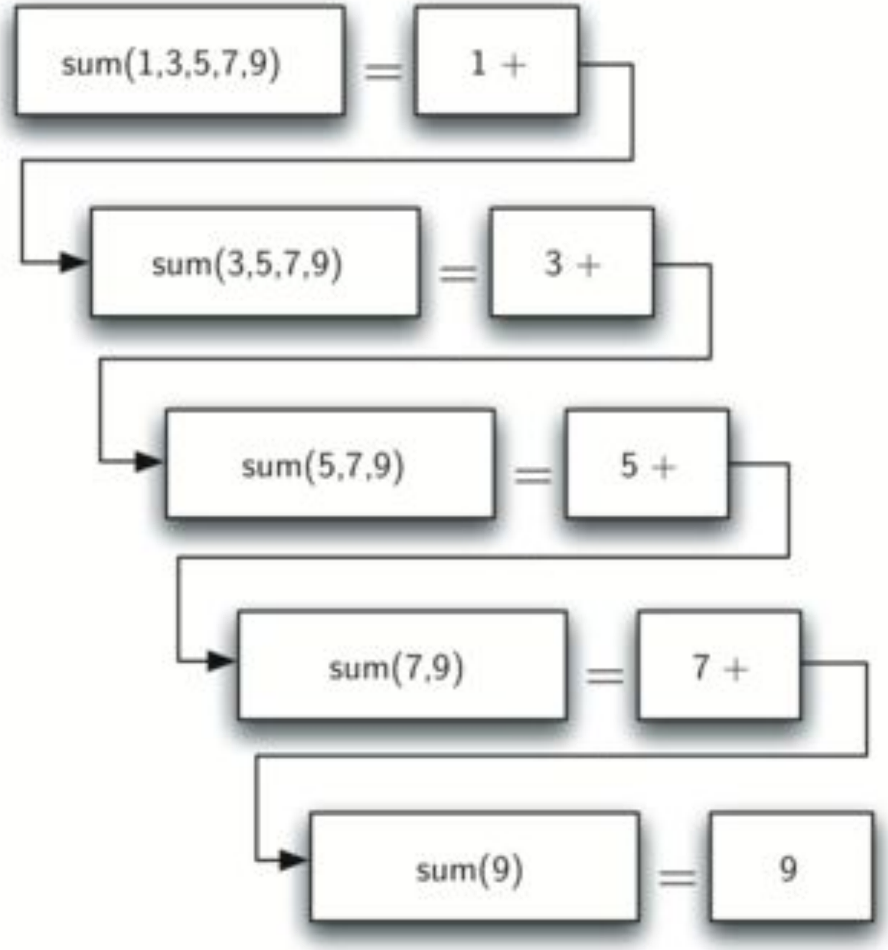
```
def sumArray(data):
  result=0
  for x in data:
     result += x
  return result

print(sumArray([3,5,8,0]))
```

Iterative solution

# Types of recursion: Linear recursion

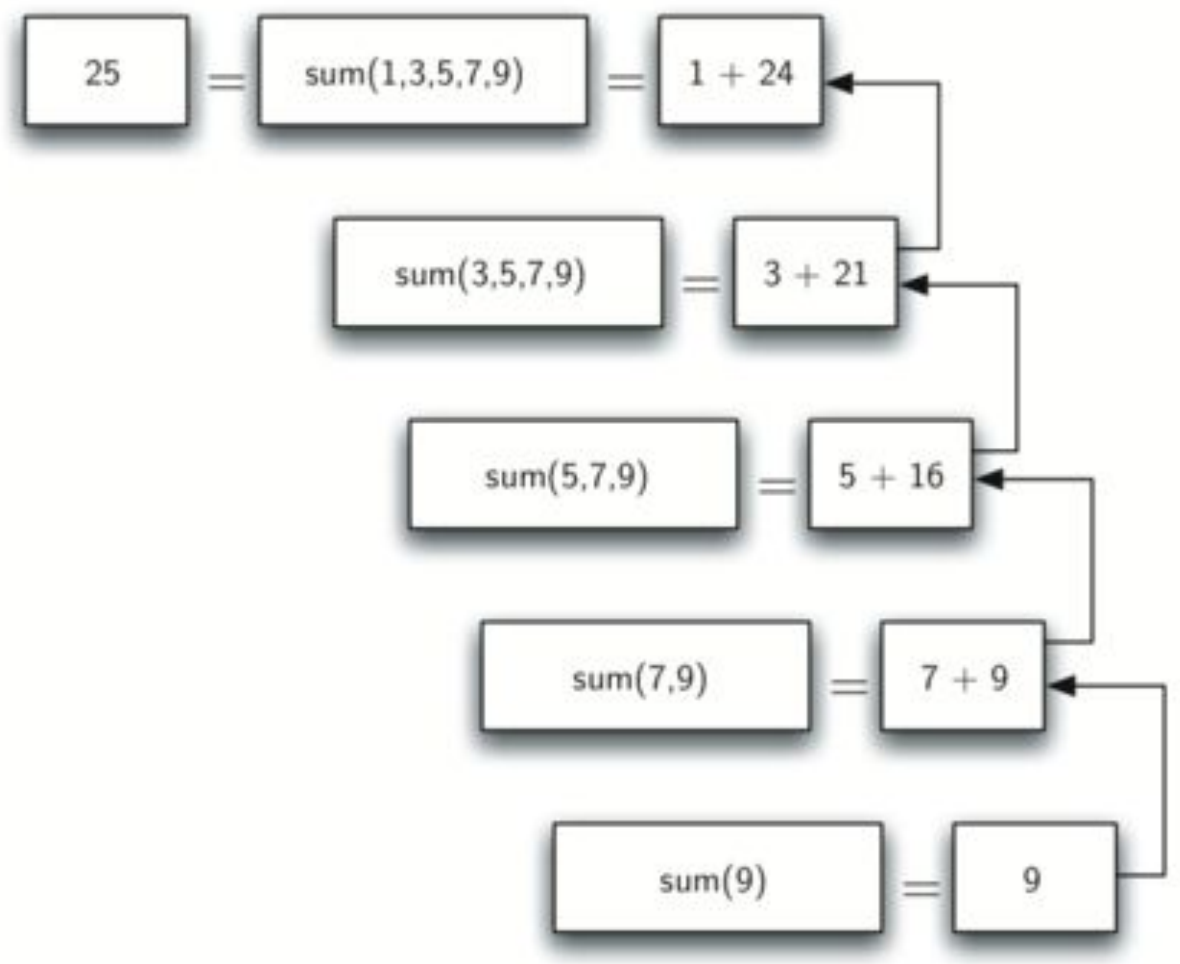
Given a sequence of numbers, [1,3,5,7,9], how can we obtain its sum?

# Types of recursion: Linear recursion



Base case

# Types of recursion: Linear recursion



# Types of recursion: Linear recursion

Example of linear recursion: Sum a list of numbers

```
def sumArrayRec(data):
  if len(data)==0:
    return 0
  else:
    return data[0] + sumArrayRec(data[1:])
```

# Types of recursion: Linear recursion

```
def sumArrayRec(data):
  if len(data)==0:
    return 0
  else:
    return data[0] + sumArrayRec(data[1:])
```

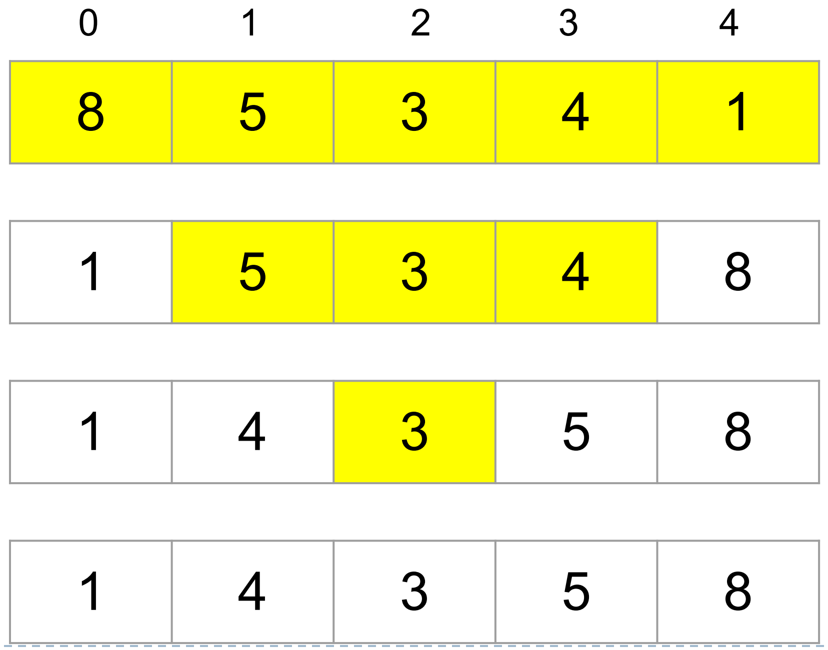
**Time complexity:** for an input of size n, it makes n+1 calls.

↓

O(n)

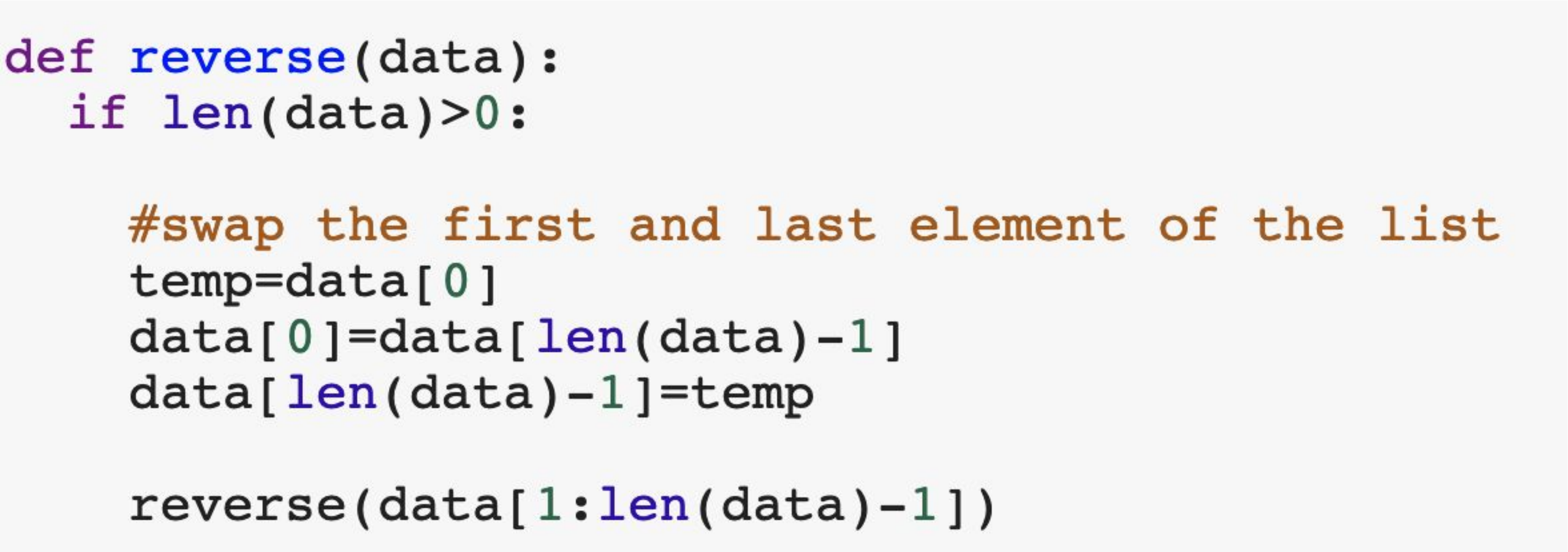
# Types of recursion: Linear recursion

- **Reversing an array**: [8,5,3,4,1] -> [1,4,3,5,8]
- Can be solved by using linear recursion: swapping first and last elements, and recursively reversing the remaining ones.



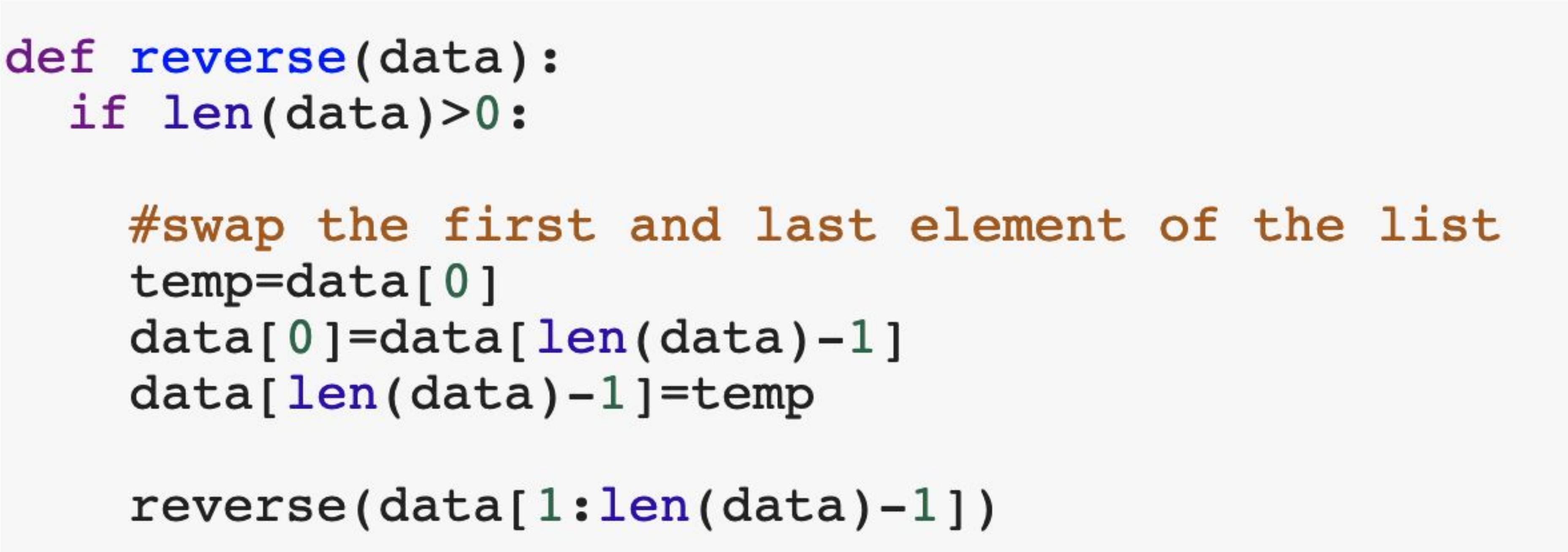
| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 8 | 5 | 3 | 4 | 1 |
| 1 | 5 | 3 | 4 | 8 |
| 1 | 4 | 3 | 5 | 8 |
| 1 | 4 | 3 | 5 | 8 |

# Types of recursion: Linear recursion



```
def reverse(data):
  if len(data)>0:

    #swap the first and last element of the list
    temp=data[0]
    data[0]=data[len(data)-1]
    data[len(data)-1]=temp

    reverse(data[1:len(data)-1])
```

# Types of recursion: Linear recursion



```
def reverse(data):
  if len(data)>0:

    #swap the first and last element of the list
    temp=data[0]
    data[0]=data[len(data)-1]
    data[len(data)-1]=temp

    reverse(data[1:len(data)-1])
```

**Time complexity:** for an input of size n, it makes 1+n/2 recursive calls.

O(n)

# Types of recursion: Linear recursion

Power function: power(x,n)=$x^n$

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

# Types of recursion: Linear recursion

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

**Time complexity:** O(n)

# Types of recursion: Linear recursion

- Find largest integer d that evenly divides into p and q.
- Euclid's algorithm (300 BCE).

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

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

# Types of recursion: Linear recursion

```
def gcd(a,b):
  #suppose, a,b>=0, a>b
  if b==0:
    return a
  else:
    return gcd(b,a%b)
```

# Index

- What is recursion?
- Some examples of recursion
- **Types of recursion**
  - Linear recursion
  - **Binary recursion**
  - Multiple recursion
- Iteration versus Recursion

# Types of recursion: Binary recursion

- Makes two recursive calls.
- We will study two examples:
    - Fibonacci numbers
    - Sum of a list of numbers using binary recursion.

# Types of recursion: Binary recursion

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

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

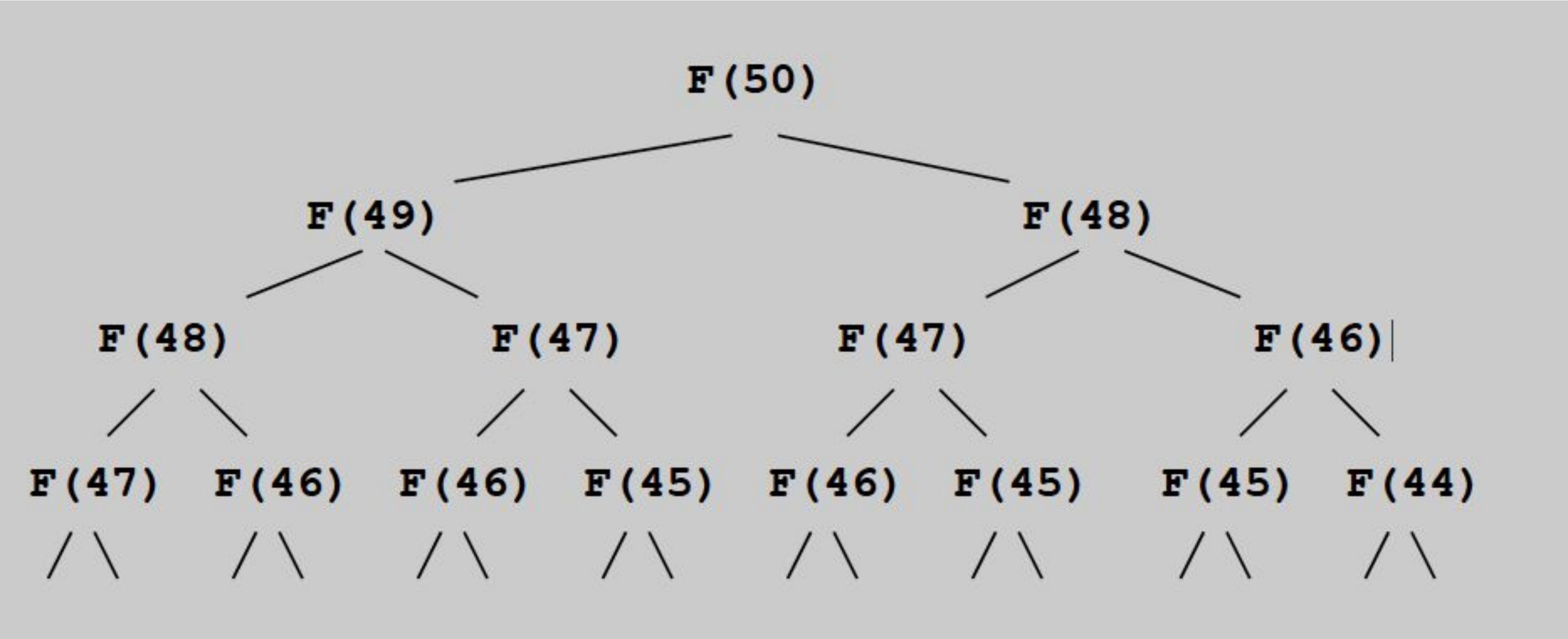
# Types of recursion: Binary recursion

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

Is this an efficient way to compute F(50)?

# Types of recursion: Binary recursion

No, no, no! This code is spectacularly inefficient: O($2^n$)



# Types of recursion: Binary recursion

A more efficient way to calculate fibonacci numbers:

```
def fibo2(n):
   """Return pair of fibonnacci numbers
   F(n), F(n-1)"""
   if n==1:
      return (1,0)
   else:
      (a,b)=fibo2(n-1)
      return (a+b,a)

print(fibo2(50))
```

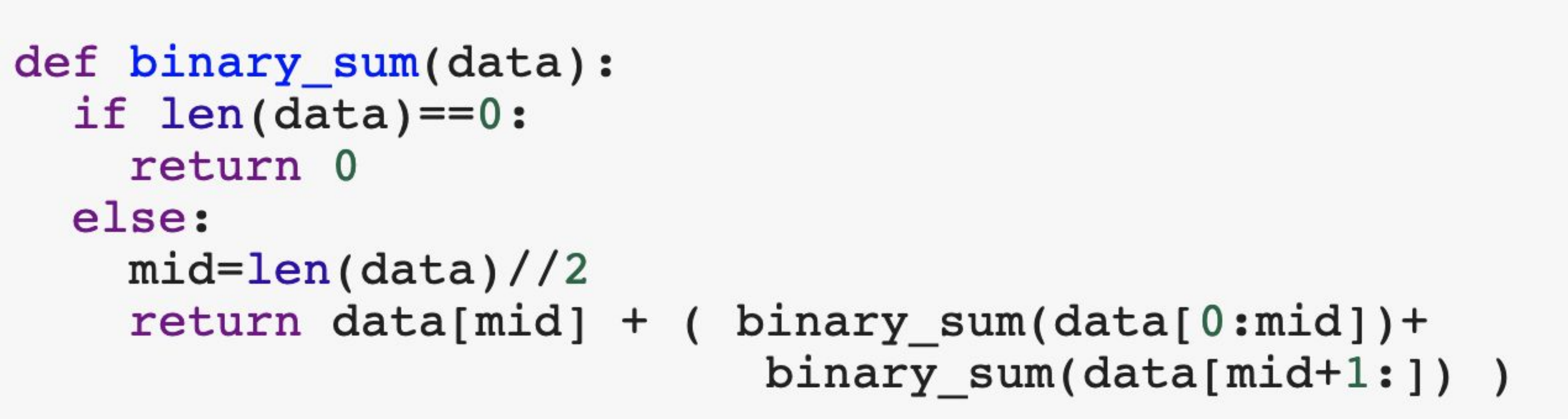
Time complexity: O(n)

# Types of recursion: Binary recursion

- How to compute the sum of an sequence of numbers using binary recursion?

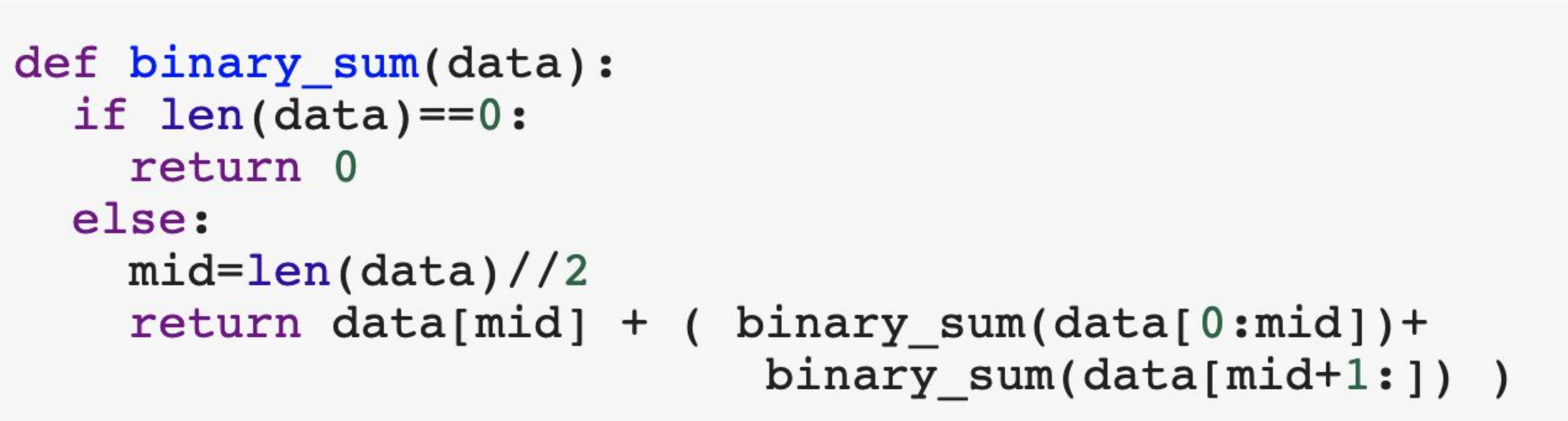
Idea!!!: divide into two halves, compute the sum of the first half, compute the sum of the second half, and add these sums

# Types of recursion: Binary recursion



```
def binary_sum(data):
  if len(data)==0:
    return 0
  else:
    mid=len(data)//2
    return data[mid] + ( binary_sum(data[0:mid])+
                          binary_sum(data[mid+1:]) )
```

# Types of recursion: Binary recursion



```
def binary_sum(data):
  if len(data)==0:
    return 0
  else:
    mid=len(data)//2
    return data[mid] + ( binary_sum(data[0:mid])+
                          binary_sum(data[mid+1:]) )
```

**Time complexity:** for an input of size n, there are 2n-1 recursive calls

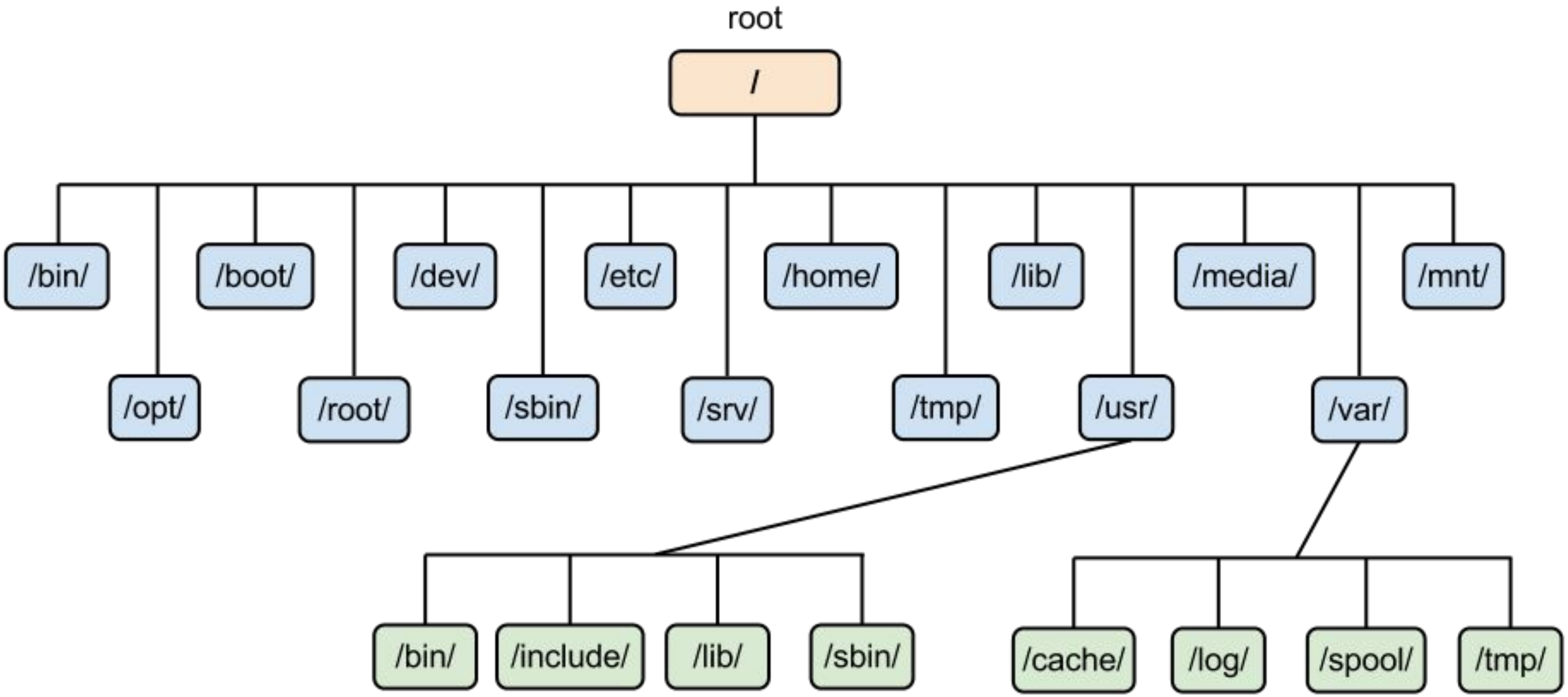
# Index

- What is recursion?
- Some examples of recursion
- **Types of recursion**
  - Linear recursion
  - Binary recursion
  - **Multiple recursion**
- Iteration versus Recursion

# Types of recursion: Multiple recursion

- Makes three or more recursive calls.
- Exploring the file system can be solved using multiple recursion

# Types of recursion: Multiple recursion



Exploring file system

# Types of recursion: Multiple recursion

How to compute the disk space usage of a given directory (path)?

```
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
```

Implement it yourself!!!

# Index

- What is recursion?
- Some examples of recursion
- **Types of recursion**
  - Linear recursion
  - Binary recursion
  - Multiple recursion
- **Iteration versus Recursion**

# Iteration vs Recursion

- A loop is also a repetitive process.
- A recursive method is more mathematically elegant than using a loop. **Recursion is** easy and neat approach (powerful programming paradigm).
- **Recursive methods** have worse time-complexity than loops (because each function call requires multiple memory to store the internal address of the method)
- All recursive methods can be solved using a iterative solution.
- Not all problem can be solved using recursive.

*To iterate is human, to recurse, divine*