Introduction to Recursion

Marcelo Ponce

March 2021

Department of Computer and Mathematical Sciences - UTSC

Today's lecture

In this first lecture we will discuss the following topics:

- Recursion: informal concept and (more) formal definition
- Examples
- Recursion in Programming
- Recursion in Data Structures
- Summary/Overview

The goal for today's lecture is to introduce you to the topic of *Recursion*, and review its main applications in programming and data structures.
Please stop me if you have a question.

Examples and codes from slides, available at

`https://github.com/mponce0/UTSC-CS_lectures`

Recursion

Recursion

Recursion is the process of defining something in terms of itself

E.g. a *recursive function* is a function that **calls itself** until a *base condition* is true, and then its execution stops.

Recursion

Recursion is the process of defining something in terms of itself

E.g. a *recursive function* is a function that **calls itself** until a *base condition* is true, and then its execution stops.

More formally, a class of objects or methods exhibits *recursive* behavior when it can be defined by two properties:

- **A base rule**: A simple base case (or cases) – a terminating scenario that does not use recursion to produce an answer.
- **A recursive step** – a set of rules that reduces all successive cases toward the base case.

Recursion

Recursion is the process of defining something in terms of itself

E.g. a *recursive function* is a function that **calls itself** until a *base condition* is true, and then its execution stops.

More formally, a class of objects or methods exhibits *recursive* behavior when it can be defined by two properties:

- **A base rule**: A simple base case (or cases) – a terminating scenario that does not use recursion to produce an answer.
- **A recursive step** – a set of rules that reduces all successive cases toward the base case.

Sourdough fermentation

Credit: *Janus Sandsgaard* (wikipedia)

Examples

- Factorial:

$$
\boldsymbol{n! \equiv \begin{cases} 0! = 1! = 1 & n = 0,1 \\ 1 \times 2 \times \ldots \times (n-1) \times n = \prod_{1}^{n} i & n > 1 \end{cases}}
$$

Examples

- Factorial:

$$n! \equiv \begin{cases} 0! = 1! = 1 & n = 0, 1 \\ 1 \times 2 \times \ldots \times (n-1) \times n = \prod_{1}^{n} i & n > 1 \end{cases}$$

$$n! \equiv \begin{cases} 1 & n = 0 \\ (n-1)! \times n & n > 0 \end{cases}$$

Examples

- Factorial:

$$n! \equiv \begin{cases} 0! = 1! = 1 & n = 0, 1 \\ 1 \times 2 \times \ldots \times (n-1) \times n = \prod_{1}^{n} i & n > 1 \end{cases}$$

$$n! \equiv \begin{cases} 1 & n = 0 \\ (n-1)! \times n & n > 0 \end{cases}$$

```
def factorialRecursive(num):
  '''Function to compute factorial using a recursive implementation'''

  # base condition: if n=0 or 1 --> n!=1
  if num < 2:
    return 1
  else:
    return num * factorialRecursive(num-1)
```

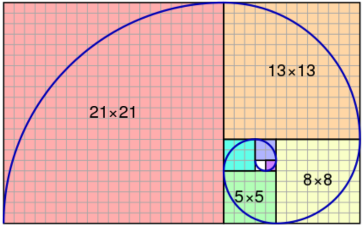
Examples

- Fibonacci numbers

Examples

- Fibonacci numbers

$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \quad n \geq 2 \end{cases}$$



Credit *Jahobr* (wikipedia)

Examples

- Fibonacci numbers

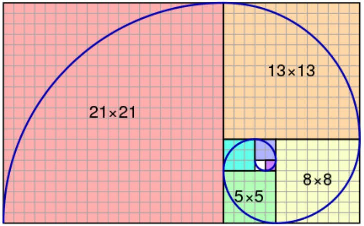
$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \quad n \geq 2 \end{cases}$$

```python
def Fibonacci(n):
  '''Recursive definition of Fibonacci numbers'''

  if n >= 2:
    # recursive rule
    return (Fibonacci(n-1)+Fibonacci(n-2))
  elif ( n==0 or n==1 ):
    # base rule
    return n
  else:
    print("Error: Fibonacci nbrs defined for 
        positive integers!")


def FibonacciSeq(n):
  '''Function to generate Fibonacci series'''

  for i in range(0,n+1):
    print(Fibonacci(i))
```



Credit *Jahobr* (wikipedia)

Examples

- Fibonacci numbers

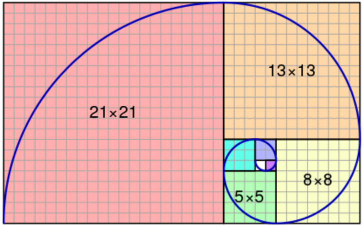
$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \quad n \geq 2 \end{cases}$$

```python
def Fibonacci(n):
    '''Recursive definition of Fibonacci numbers'''

    if n >= 2:
        # recursive rule
        return (Fibonacci(n-1)+Fibonacci(n-2))
    elif ( n==0 or n==1 ):
        # base rule
        return n
    else:
        print("Error: Fibonacci nbrs defined for 
            positive integers!")


def FibonacciSeq(n):
    '''Function to generate Fibonacci series'''

    for i in range(0,n+1):
        print(Fibonacci(i))
```



Credit *Jahobr* (wikipedia)

- Fibonacci nbrs – multiple applications in CS: GCD, merge-sort, Fibonacci trees, PRN,...

Examples

- Fibonacci numbers

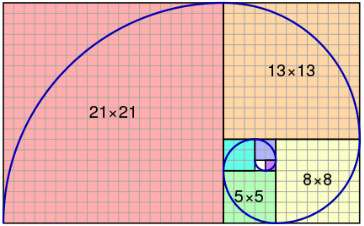
$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \quad n \geq 2 \end{cases}$$

```python
def Fibonacci(n):
  '''Recursive definition of Fibonacci numbers'''

  if n >= 2:
    # recursive rule
    return (Fibonacci(n-1)+Fibonacci(n-2))
  elif ( n==0 or n==1 ):
    # base rule
    return n
  else:
    print("Error: Fibonacci nbrs defined for 
        positive integers!")


def FibonacciSeq(n):
  '''Function to generate Fibonacci series'''

  for i in range(0,n+1):
    print(Fibonacci(i))
```



Credit *Jahobr* (wikipedia)

- Fibonacci nbrs – multiple applications in CS: GCD, merge-sort, Fibonacci trees, PRN,...
- Mathematics
  - Fractals
  - Proof by induction

Recursive Algorithms

The Three “Laws” of Recursion

- A recursive algorithm must have a *base case –rule–*.
- A recursive algorithm must change its *state* and “move” toward the base case – **converge**.
- A recursive algorithm must call itself, *recursively*.

Recursion vs Iteration

In many cases, a recursive implementation can be done using an iterative approach.

Recursion vs Iteration

In many cases, a recursive implementation can be done using an iterative approach.

Let's go back to our factorial example,

$$n! \equiv \begin{cases} 1 & n = 0 \\ (n-1)! \times n & n > 0 \end{cases}$$

In many cases, a recursive implementation can be done using an iterative approach.

Let's go back to our factorial example,

$$n! \equiv \begin{cases} 1 & n = 0 \\ (n-1)! \times n & n > 0 \end{cases} \qquad n! \equiv \begin{cases} 1 & n = 0 = 1 \\ \displaystyle\prod_{i=1}^{n} i & n > 1 \end{cases}$$

Recursion vs Iteration

In many cases, a recursive implementation can be done using an iterative approach.

Let's go back to our factorial example,

$$n! \equiv \begin{cases} 1 & n = 0 \\ (n-1)! \times n & n > 0 \end{cases}$$

$$n! \equiv \begin{cases} 1 & n = 0 = 1 \\ \prod_{i=1}^{n} i & n > 1 \end{cases}$$

```python
def factorialRecursive(num):
  '''Function to compute factorial using a
       recursive implementation'''

  # base condition: if n=0 or 1 --> n!=1
  if num < 2:
    return 1
  else:
    return num * factorialRecursive(num-1)
```

Recursion vs Iteration

In many cases, a recursive implementation can be done using an iterative approach.

Let's go back to our factorial example,

$$n! \equiv \begin{cases} 1 & n = 0 \\ (n-1)! \times n & n > 0 \end{cases}$$

$$n! \equiv \begin{cases} 1 & n = 0 = 1 \\ \prod_{i=1}^{n} i & n > 1 \end{cases}$$

```python
def factorialRecursive(num):
  '''Function to compute factorial using a
      recursive implementation'''

  # base condition: if n=0 or 1 --> n!=1
  if num < 2:
    return 1
  else:
    return num * factorialRecursive(num-1)
```

```python
def factorialIteractive(num):
  '''Function to compute factorial
      using the iterative product
      implementation'''

  fact = 1

  # iteration to compute product
  # i.e. n! = 1*2*3*...*(n-1)*n
  for n in range(2, num + 1):
    fact = fact * n

  return fact
```

Recursion vs Iteration: Performance

These two implementations do not only differ on the programming styles...

Recursion vs Iteration: Performance

These two implementations do not only differ on the programming styles...

Performance

Usually recursive implementations perform worth than iterative ones.

Recursion vs Iteration: Performance

These two implementations do not only differ on the programming styles...

Performance

Usually recursive implementations perform worth than iterative ones.

```
factorialRecursive(3)           # 1st call with 3
3 * factorialRecursive(2)       # 2nd call with 2
3 * 2 * factorialRecursive(1) # 3rd call with 1
3 * 2 * 1           # return from 3rd call as num=1
3 * 2               # return from 2nd call
6                   # return from 1st call
```

Recursion vs Iteration: Performance

These two implementations do not only differ on the programming styles...

Performance

Usually recursive implementations perform worth than iterative ones.

```
factorialRecursive(3)          # 1st call with 3
3 * factorialRecursive(2)      # 2nd call with 2
3 * 2 * factorialRecursive(1) # 3rd call with 1
3 * 2 * 1          # return from 3rd call as num=1
3 * 2              # return from 2nd call
6                  # return from 1st call
```

Implem.	Fn.calls	cum.Time
Iterative	17	26 msec
Recursive	10119	40 msec
math.factorial	17	28 msec
numpy	12682	262 msec

Recursion vs Iteration: Stack

Recursion vs Iteration: Stack

Stack Function Calls

Recursion vs Iteration: Stack

Stack Function Calls

A *stack* is a data structure that operates on a **Last In/First Out** (LIFO) basis.

An item is "pushed" onto a stack to add to it, and an item is "popped" off the stack to remove it.

Using a stack is a method of ordering certain operations for execution.

Recursion vs Iteration: Stack

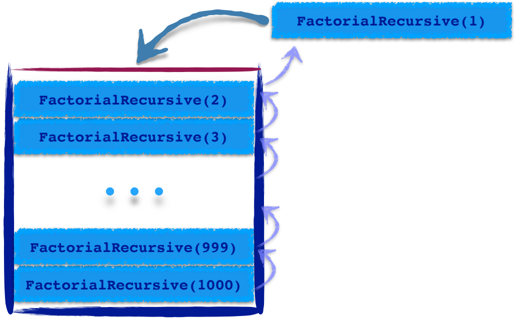
Stack Function Calls

A *stack* is a data structure that operates on a **Last In/First Out** (LIFO) basis.

An item is "pushed" onto a stack to add to it, and an item is "popped" off the stack to remove it.

Using a stack is a method of ordering certain operations for execution.

A recursive function, calls itself until a "base condition" is met and then its execution stops.
While the condition is not met, we will keep placing *execution contexts* on top of the stack.
In some cases, this could lead to a **stack overflow**: *run out of "space" to hold items in the stack.*



Python specific limits on Recursion

- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in **RecursionError**.

Python specific limits on Recursion

- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in **RecursionError**.

```
>>> from factorials import *

>>> factorialRecursive(3)
6

>>> factorialRecursive(999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File code/factorials.py", line 28, in factorialRecursive
    return num * factorialRecursive(num-1)
   ...
RecursionError: maximum recursion depth exceeded in comparison

>>> factorialRecursive(998)
4027900501272209945382406745976015873066815457564711036474447...
```

For more details, see the GitHub repository for this lecture.

Quick Sort

- It's an in-place sorting algorithm with worst-case time complexity $\sim \mathcal{O}(n^2)$
- It's a *divide-and-conquer* algorithm
- It works by selecting a **'pivot'** element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted *recursively*.

```
('Original Array: ', [4, 2, 7, 3, 1, 6])
([1, 2, 3], 4, [7, 6])
([], 1, [2, 3])
([], 2, [3])
([6], 7, [])
('Sorted Array: ', [1, 2, 3, 4, 6, 7])
```

```python
def QuickSort(arr, verbose=False):

    elements = len(arr)

    if elements < 2:  # Base case
        return arr

    current_position = 0  #Position of the partitioning element

    for i in range(1, elements):  #Partitioning loop
        if arr[i] <= arr[0]:
            current_position += 1
            temp = arr[i]
            arr[i] = arr[current_position]
            arr[current_position] = temp

    temp = arr[0]
    arr[0] = arr[current_position]
    arr[current_position] = temp  #Brings pivot to it's appropriate position

    if verbose:
        print(arr[0:current_position],arr[current_position],
              arr[current_position+1:elements])

    left = QuickSort(arr[0:current_position],verbose) #Sorts the elements to the l
    right = QuickSort(arr[current_position+1:elements],verbose) #Sorts the element

    arr = left + [arr[current_position]] + right #Merging everything together

    return arr
```

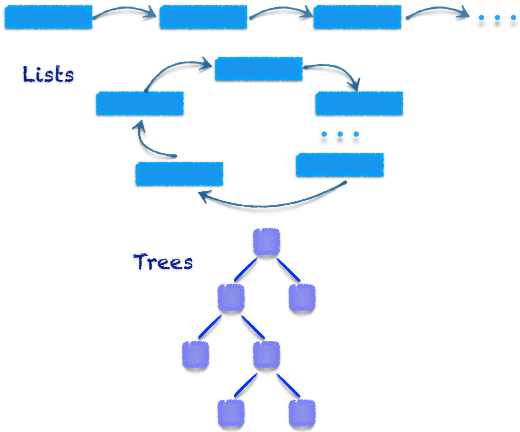
Recursive Data Structures

Recursive Data Structures

So far, we have been talking about recursive algorithms... but similar recursion concepts apply to data structures.

Recursive Data Structures

So far, we have been talking about recursive algorithms... but similar recursion concepts apply to data structures.



For instance,

- Lists, sets, dictionaries
- Nested lists
- Trees
- File-system: folders contain other folders can contain other folders, until finally at the bottom of the recursion are plain (non-folder) files.

Recursive lists

```
def addList(lst, newElement, head=True):
  """function to add elements to a list
      recursively"""
  # elements can be added at the
      beginning or the end of the list
  if head:
    lst = [newElement] + lst
  else:
    lst = lst + [newElement]
  return lst

def nestedList(lst, newElement):
  """function to generate nested lists of
       new elements"""
  lst = list([lst, newElement])
  return lst
```

Recursive lists

```python
def addList(lst, newElement, head=True):
  """function to add elements to a list
      recursively"""
  # elements can be added at the
      beginning or the end of the list
  if head:
    lst = [newElement] + lst
  else:
    lst = lst + [newElement]
  return lst

def nestedList(lst, newElement):
  """function to generate nested lists of
       new elements"""
  lst = list([lst, newElement])
  return lst
```

```python
mylist=[0]
mynestedlist=[0]
for i in range(1,7):
  # create a list by appending elements
  mylist = addList(mylist,i)
  # create a nested list
  mynestedlist = nestedList(
      mynestedlist,i)

# display resulting lists
print("incremental list: ", mylist)
print("Nested list: ", mynestedlist)

# combine list
print("append list+nested list: ",
    addList(mylist,mynestedlist))
print("nest list+nested list: ",
    nestedList(mylist,mynestedlist))
```

Recursive lists

```python
def addList(lst, newElement, head=True):
  """function to add elements to a list
      recursively"""
  # elements can be added at the
      beginning or the end of the list
  if head:
    lst = [newElement] + lst
  else:
    lst = lst + [newElement]
  return lst

def nestedList(lst, newElement):
  """function to generate nested lists of
      new elements"""
  lst = list([lst, newElement])
  return lst
```

```python
mylist=[0]
mynestedlist=[0]
for i in range(1,7):
  # create a list by appending elements
  mylist = addList(mylist,i)
  # create a nested list
  mynestedlist = nestedList(
      mynestedlist,i)

# display resulting lists
print("incremental list: ", mylist)
print("Nested list: ", mynestedlist)

# combine list
print("append list+nested list: ",
    addList(mylist,mynestedlist))
print("nest list+nested list: ",
    nestedList(mylist,mynestedlist))
```

```
incremental list:  [6, 5, 4, 3, 2, 1, 0]
Nested list:  [[[[[[[0], 1], 2], 3], 4], 5], 6]
append list+nested list:  [[[[[[[[0], 1], 2], 3], 4], 5], 6], 6, 5, 4, 3, 2, 1, 0]
nest list+nested list:  [[6, 5, 4, 3, 2, 1, 0], [[[[[[[0], 1], 2], 3], 4], 5], 6]]
```

Trees

```
# binary tree
class bTree:
    def __init__(self):
        self.left = None
        self.right = None
        self.data = None

# example
root = bTree()
root.data = "root"
root.left = bTree()
root.left.data = "left"
root.right = bTree()
root.right.data = "right"
```

```
# arbitrary childs
class aTree:
    def __init__(self, data):
        self.children = []
        self.data = data

# example
left = aTree("left")
middle = aTree("middle")
right = aTree("right")
root = aTree("root")
root.children = [left, middle, right]
```

```
# tree using dictionaries
tree = {
    "a": ["b", "c"],
    "b": ["d", "e"],
    "c": [None, "f"],
    "d": [None, None],
    "e": [None, None],
    "f": [None, None],
}
```

Closing Remarks

Reentrant Code

- Recursion is a special case of a more general situation known as *reentrancy*.
- Reentrant code can be safely re-entered, meaning that it can be *called again* even while a call to it is underway.
- Direct recursion is one way that reentrancy can happen.
- Mutual recursion between two or more functions is another way this can happen – A calls B, which calls A again...
- Direct mutual recursion is virtually always intentional and designed by the programmer. But unexpected mutual recursion can lead to bugs.
- Reentrancy is also related to *concurrency*: in a concurrent program, a method may be called at the same time by different parts of the program that are running concurrently.

Summary

Recursion

- Elegant and clean programming approach
- Powerful due to its simplicity and direct "translation"
- In some cases it can be a direct implementation from the problem
- Recursion $\leftrightarrow$ Iteration
- Be aware of corner/edge cases ("base rule") or infinite recursions, sometimes logic can be complex
  - implement test cases
- Be aware of limitations and pitfalls (i.e. performance and stack utilization)
- Relevant also in data structures

Examples and codes from slides, available at

`https://github.com/mponce0/UTSC-CS_lectures`

Q&A

Questions?

Q&A

Questions?

What does the following function do?

```
def to_string(n, b, convTab='0123456789abcdef'):
    # base case
    if n < b:
        return convTab[n]

    # // -> integer division ; % -> reminder of the division
    return to_string(n // b, b, convTab) + convTab[n % b]
```

Questions?

What does the following function do?

```python
def to_string(n, b, convTab='0123456789abcdef'):
    # base case
    if n < b:
        return convTab[n]

    # // -> integer division ; % -> reminder of the division
    return to_string(n // b, b, convTab) + convTab[n % b]
```

```python
print(to_string(1453, 16))  # => 5Ad
```