

UE5 Fundamentals of Algorithms

Lecture 2: Recursion

Ecole Centrale de Lyon, Bachelor of Science in Data Science for Responsible Business

Romain Vuillemot



Outline

- What is recursion?
- Recursive/iterative approach
- Recursive data structures
- Tail vs non-tail recursion

What is recursion?

*A function is considered **recursive** when it invokes itself. This self-invocation typically involves handling a straightforward case, known as the **base case**, and implementing instructions that lead towards reaching this base case.*

Why recursion?

- Main reason: some problems are more easily implementable recursively.
- Other reasons: Iterative code is difficult to parallelize
- Code can be more elegant and concise than the iterative version, simpler to implement.
- Similarity to proof by induction.

Example 1: factorial (iterative)

Reminder: Reminder: here is the factorial function in the iterative version

```
In [13]: def factorial_iter(n):  
         r = 1  
         for i in range(1, n + 1):  
             r *= i  
         return r
```

```
In [14]: assert factorial_iter(0) == 1  
         assert factorial_iter(1) == 1  
         assert factorial_iter(5) == 120  
         assert factorial_iter(10) == 3628800
```

- The function `fact_iter` is called once
- Uses iterative operator (loop, while, etc.)

Example 1: factorial (recursive)

```
In [15]: def factorial_rec(n):  
         if n == 0:  
             return 1  
         else:  
             return n * factorial_rec(n - 1)
```

```
In [16]: assert factorial_iter(0) == factorial_rec(0)  
         assert factorial_iter(1) == factorial_rec(1)  
         assert factorial_iter(5) == factorial_rec(5)  
         assert factorial_iter(10) == factorial_rec(10)
```

- The function `factorial_rec` is called multiple times
- Identical results to an *iterative* version of the function

Example: Sum of digits

```
In [ ]: def recursive_sum(arr):  
        if not arr:  
            return 0  
        else:  
            return arr[0] + recursive_sum(arr[1:])  
  
        def iterative_sum(arr):  
            total = 0  
            for num in arr:  
                total += num  
            return total
```

Example 2: Fibonnacci (iterative)

```
In [27]: def fibo_iter(n):  
         arr = [0, 1]  
         for i in range(2, n+1):  
             arr.append(arr[i-1] + arr[i-2])  
         return arr[n]
```

```
In [28]: fibo_iter(4)
```

```
Out[28]: 3
```

Example 2: Fibonnacci (iterative)

```
In [27]: def fibo_iter(n):  
         arr = [0, 1]  
         for i in range(2, n+1):  
             arr.append(arr[i-1] + arr[i-2])  
         return arr[n]
```

```
In [28]: fibo_iter(4)
```

```
Out[28]: 3
```

Recursive version?

Example 2: Fibonnacci (iterative)

```
In [27]: def fibo_iter(n):  
         arr = [0, 1]  
         for i in range(2, n+1):  
             arr.append(arr[i-1] + arr[i-2])  
         return arr[n]
```

```
In [28]: fibo_iter(4)
```

```
Out[28]: 3
```

Recursive version?

```
In [29]: def fibo_rec(n):  
         if n <= 1:  
             return n  
         else:  
             return fibo_rec(n-1) + fibo_rec(n-2)
```

```
In [26]: assert fibo_rec(10) == fibo_iter(10)
```

Recursion in real life (plants)

https://en.wikipedia.org/wiki/Barnsley_fern

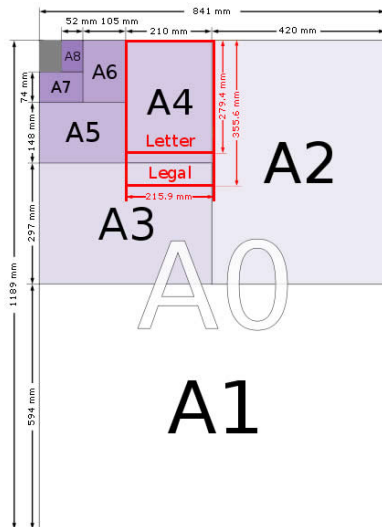
https://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension



Recursion in real life (paper folding)

https://en.wikipedia.org/wiki/Paper_size

(uses the ISO 216 standard's scaling factor of approximately 1.1892071 ($2^{1/4}$) to calculate the dimensions of various ISO paper sizes based on their relation to A0)



Recursion in real life (paper folding) in Pseudo Code

1. Constant Ratio:

- $\text{ratio} = \sqrt{2}$

2. Creating A_i from A_{i-1} :

- function createPaper(A_{i-1}):
 - fold A_{i-1} along its length to create A_i

3. Recurrence Relation:

- function calculateDimensions(A_{i-1}):
 - $\text{length_}A_i = \text{width_}A_{i-1}$
 - $\text{width_}A_i = \text{length_}A_{i-1} / 2$

Recursion in real life (paper folding) in Python

```
In [23]: def generate_iso_paper_sizes(n):  
    iso_sizes = {"A0": (841, 1189)}  
    current_size = "A0"  
  
    for i in range(1, n + 1):  
        width, height = iso_sizes[current_size]  
        next_size = f"A{i}"  
        iso_sizes[next_size] = (height / 2, width)  
        current_size = next_size  
  
    return iso_sizes  
  
    # Generate and print the first 5 ISO paper sizes  
    iso_paper_sizes = generate_iso_paper_sizes(5)  
    for size, dimensions in iso_paper_sizes.items():  
        print(f"{size}: Width = {dimensions[0]} mm, Height = {dimensions[1]} mm")
```

```
A0: Width = 841 mm, Height = 1189 mm  
A1: Width = 594.5 mm, Height = 841 mm  
A2: Width = 420.5 mm, Height = 594.5 mm  
A3: Width = 297.25 mm, Height = 420.5 mm  
A4: Width = 210.25 mm, Height = 297.25 mm  
A5: Width = 148.625 mm, Height = 210.25 mm
```

Recursive data structures

Some **data types** are inherently recursive, meaning that a subset of this data type has the same data type:

- Lists
- Strings (arrays)
- Binary trees
- Linked lists
- Custom (e.g., using objects)

Tail vs non-tail recursion

Recursion can be categorized into two types: **tail recursion** and **non-tail recursion**.

- **tail recursion**, the recursive call is the last operation before returning a result. This makes it efficient in terms of memory usage and stack overflow risk, as there are no pending calculations.
- **non-tail recursion** involves additional operations after the recursive call, potentially leading to a stack of calls and increased memory consumption.

The choice between these types depends on the problem and programming language. Tail recursion often leads to more efficient and readable code, but sometimes non-tail recursion is necessary, impacting variable lifespan and the result.

⚠ Impacts results and variables span of life

Tail vs non-tail recursion (cont.)

```
In [31]: def print_desc(n):  
         if n > 0:  
             print(" " * n, " n= " , n , " before recursive call")  
             ecrire_desc (n - 1)
```

```
In [32]: print_desc(3)
```

```
    n= 3  before recursive call  
    n= 2  avant l'appel récursif  
    n= 1  avant l'appel récursif
```


Tail vs non-tail recursion (cont.)

```
In [31]: def print_desc(n):  
         if n > 0:  
             print(" " * n, " n= " , n , " before recursive call")  
             ecrire_desc (n - 1)
```

```
In [32]: print_desc(3)
```

```
    n= 3  before recursive call  
    n= 2  avant l'appel récursif  
    n= 1  avant l'appel récursif
```

```
In [33]: def print_asc(n) :  
         if n > 0:  
             ecrire_asc(n - 1)  
             print(" " * n, " n= " , n , " after recursive call")
```

```
In [34]: print_asc(3)
```

```
    n= 1  après l'appel récursif  
    n= 2  après l'appel récursif  
    n= 3  after recursive call
```

Disadvantages of Recursion



- While recursion can lead to concise and elegant solutions, it's crucial to handle base cases properly to avoid infinite loops or excessive function calls.
- Recursive functions require careful consideration of termination conditions to prevent stack overflow errors or excessive memory usage.
- Can become memory-intensive if written poorly.
- One must wait for the base case to start getting results (no intermediate result), results cannot be obtained progressively (if non-tail recursion).

Recursion calls

The screenshot displays a Python 3.6 IDE with a recursive function `recherche_dichotomique_recursive` and its call stack. The function is defined as follows:

```
1 def recherche_dichotomique_recursive( element, liste_triee):
2     if a == b :
3         return a
4     if b == -1 :
5         b = len(liste_triee)-1
6     m = (a+b)//2
7     if liste_triee[m] == element :
8         return m
9     elif liste_triee[m] > element :
10        return recherche_dichotomique_recursive(element,
11        else :
12            return recherche_dichotomique_recursive(element,
13
14 recherche_dichotomique_recursive(1, [1, 2, 3])
```

The call stack shows the following frames:

- Global frame: `recherche_dichotomique_recursive`
- `recherche_dichotomique_recursive` frame: `element` is 1, `liste_triee` is [1, 2, 3], `a` is 0, `b` is 2, `m` is 1.
- `recherche_dichotomique_recursive` frame: `element` is 1, `liste_triee` is [1, 2, 3], `a` is 0, `b` is 0.

The `Objects` panel shows the function `recherche_dichotomique_recursive` with default arguments `a` (0) and `b` (-1). It also shows a list object `[1, 2, 3]` with indices 0, 1, and 2.

Visualize recursive calls [pythontutor](https://pythontutor.com/)

In case of too many function calls, a **maximum recursion depth exceeded error** is triggered. You can anticipate this error by not exceeding the limit `sys.getrecursionlimit()` or by changing it using `sys.setrecursionlimit()`.

```
In [30]: import sys
sys.getrecursionlimit()
```

```
Out[30]: 3000
```

Transformation Schemes

There are some recursive to iterative transformation schemes:

1. Write a recursive algorithm.
2. Test, validate, and prove it (in critical applications).
3. Apply transformation techniques to obtain an iterative version.

Note that these techniques are not 100% automated.

```
f_rec(X) =  
  if p(X) then a(X)  
  else  
    b(X)  
    f_rec(new(X))  
  end if;  
  
f_iter(X) =  
  if p(X) = false then  
    do  
      b(X);  
      X := new(X);  
    until p(X) = true;  
  end if;  
  a(X);
```

