

A gamified introduction to Python Programming

Lecture 9

About recursion and decorators

Matthieu DE MARI – Singapore University of Technology and Design



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN



Outline (Chapter 9)

- What is **recursion** in programming?
- How can **recursion** be used as an alternative to looping?
- What is the **LRU cache decorator** and its purpose?
- How to measure the **performance** of functions?
- What is a **decorator** anyway?
- Some practice on recursion.

Recursion: definition

Definition (**recursion**)

Recursion is a common mathematical and programming concept. It means that a **function can be defined by calling itself**.

- An example of a recursive function is the **factorial function** with value n (denoted $n!$). You typically know it as

$$n! = 1 \times 2 \times \cdots \times n$$

- The factorial function value can also be defined in a recursive manner, as follows.

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

- In other words, we can define the value of the **factorial function with value n** , by reusing the factorial function with value $(n - 1)$.

Recursion vs. **for** loop

The **factorial function** with value n (denoted $n!$) can be defined as

$$n! = 1 \times 2 \times \cdots \times n$$

- Following this definition, we could compute the value of the factorial using a simple **for** loop.
- And that would work just fine.

```
1 def factorial_fun_for(n):  
2     result = 1  
3     for i in range(1, n + 1):  
4         result *= i  
5     return result
```

```
1 print(factorial_fun_for(1))
```

1

```
1 print(factorial_fun_for(3))
```

6

```
1 print(factorial_fun_for(5))
```

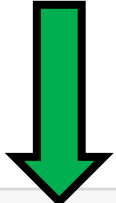
120

Recursion vs. **for** loop

But we could also use the recursive definition for the factorial function, defined as

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

- And it works as well, without any **for** loop!
- (Some people would even argue that this second function is in fact simpler!)



```
1 def factorial_fun_rec(n):
2     if n == 1:
3         value = 1
4     else:
5         value = n*factorial_fun_rec(n - 1)
6     return value
```

```
1 print(factorial_fun_rec(1))
```

1

```
1 print(factorial_fun_rec(3))
```

6

```
1 print(factorial_fun_rec(5))
```

120



Recursion vs. **for** loop

But we could also use the recursive definition for the factorial function, defined as

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

A recursive function should have:

- A **base case**, here **n = 1**, which leads to a direct result and allows the recursion to stop.
- A **recursive case** that allows for the recursion to happen.

```
1 def factorial_fun_rec(n):  
2     if n == 1:  
3         value = 1  
4     else:  
5         value = n*factorial_fun_rec(n - 1)  
6     return value
```

```
1 print(factorial_fun_rec(1))
```

1

```
1 print(factorial_fun_rec(3))
```

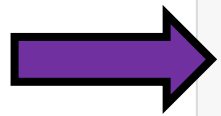
6

```
1 print(factorial_fun_rec(5))
```

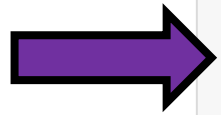
120

Recursion vs. **for** loop

What happens in practice? Multiple **concurrent runs** of a function.



```
1 def factorial_fun_rec(n):  
2     print("Function called, with n = {}".format(n))  
3     if n == 1:  
4         value = 1  
5     else:  
6         value = n*factorial_fun_rec(n - 1)  
7     print("Function completed, with n = {}, and return value = {}".format(n, value))  
8     return value
```



```
1 print(factorial_fun_rec(1))
```

```
Function called, with n = 1  
Function completed, with n = 1, and return value = 1  
1
```

```
1 print(factorial_fun_rec(3))
```



```
Function called, with n = 3  
Function called, with n = 2  
Function called, with n = 1  
Function completed, with n = 1, and return value = 1  
Function completed, with n = 2, and return value = 2  
Function completed, with n = 3, and return value = 6  
6
```

f(3)

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

executes f with
argument 2

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

executes f with
argument 1

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

executes f with
argument 0

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

Forward pass for
the recursion

At this stage, there are four zones of
memory in the **call stack**, each due to
one call to function f.

Backward pass for the recursion

f(3)

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

executes f with
argument 2

f(2)

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

executes f with
argument 1

f(1)

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

executes f with
argument 0

f(0)

```
if( n == 0):  
    return 0  
else:  
    return n + f(n-1)
```

The 1st call to f is now able to evaluate $n + f(n - 1)$ and returns 6 to the initial statement that executed it.

The 2nd call to f is now able to evaluate $n + f(n - 1)$ and returns 3 to the 1st call.

The 3rd call to f is now able to evaluate $n + f(n - 1)$ and returns 1 to the 2nd call.

The 4th call to f triggers the **base case**, which returns 0 to the third call.

At this stage, there are four zones of memory in the **call stack**, each due to one call to function f.

Recursion vs. **for** loop

Recursion is an interesting trick for computing the value of a function, which would normally require an **iterative for loop**.

- In some cases, recursion might be **faster** than an iterative for loop.
- Sometimes it is not.

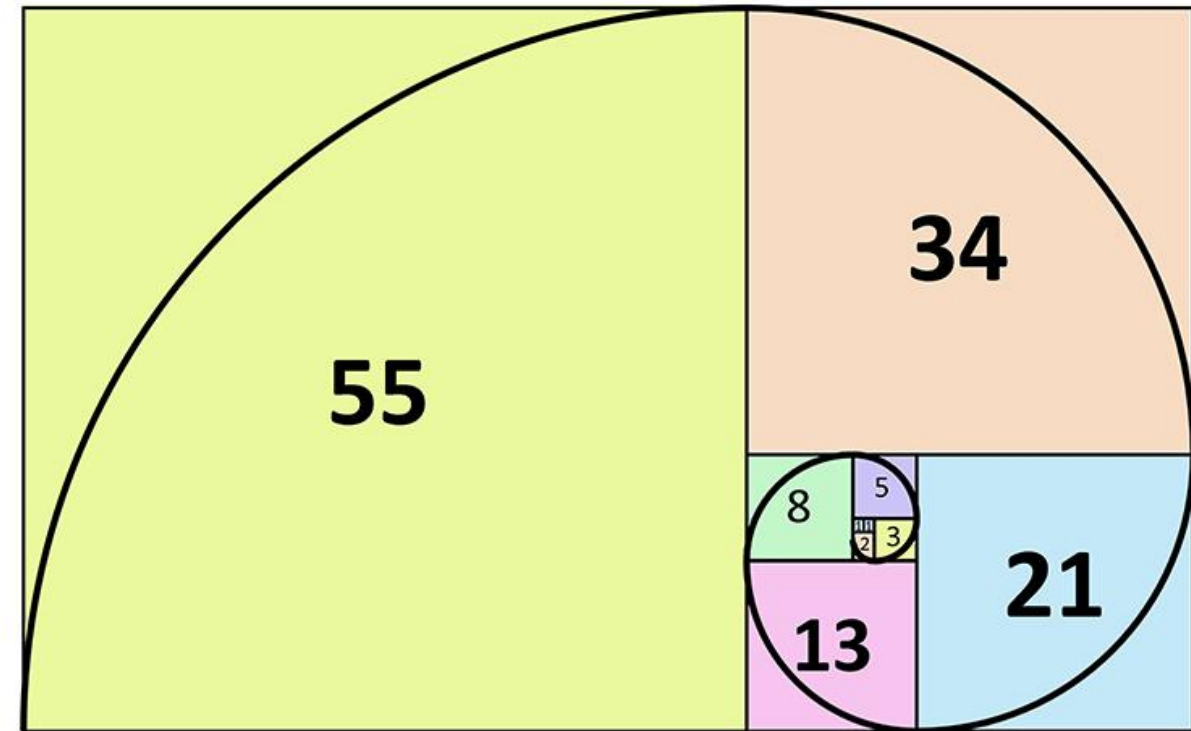
→ **Something we will discuss once we investigate computational complexity (a.k.a. the science of designing the best code for a task)**

Practice activities: recursion vs. **for** loops

Let us practice a bit

**Activity – Fibonacci
sequence.ipynb**

The Fibonacci sequence F is a mathematical curiosity.



Practice activities: recursion vs. **for** loops

Let us practice a bit

Activity – Fibonacci sequence.ipynb

The Fibonacci sequence F is a mathematical curiosity.

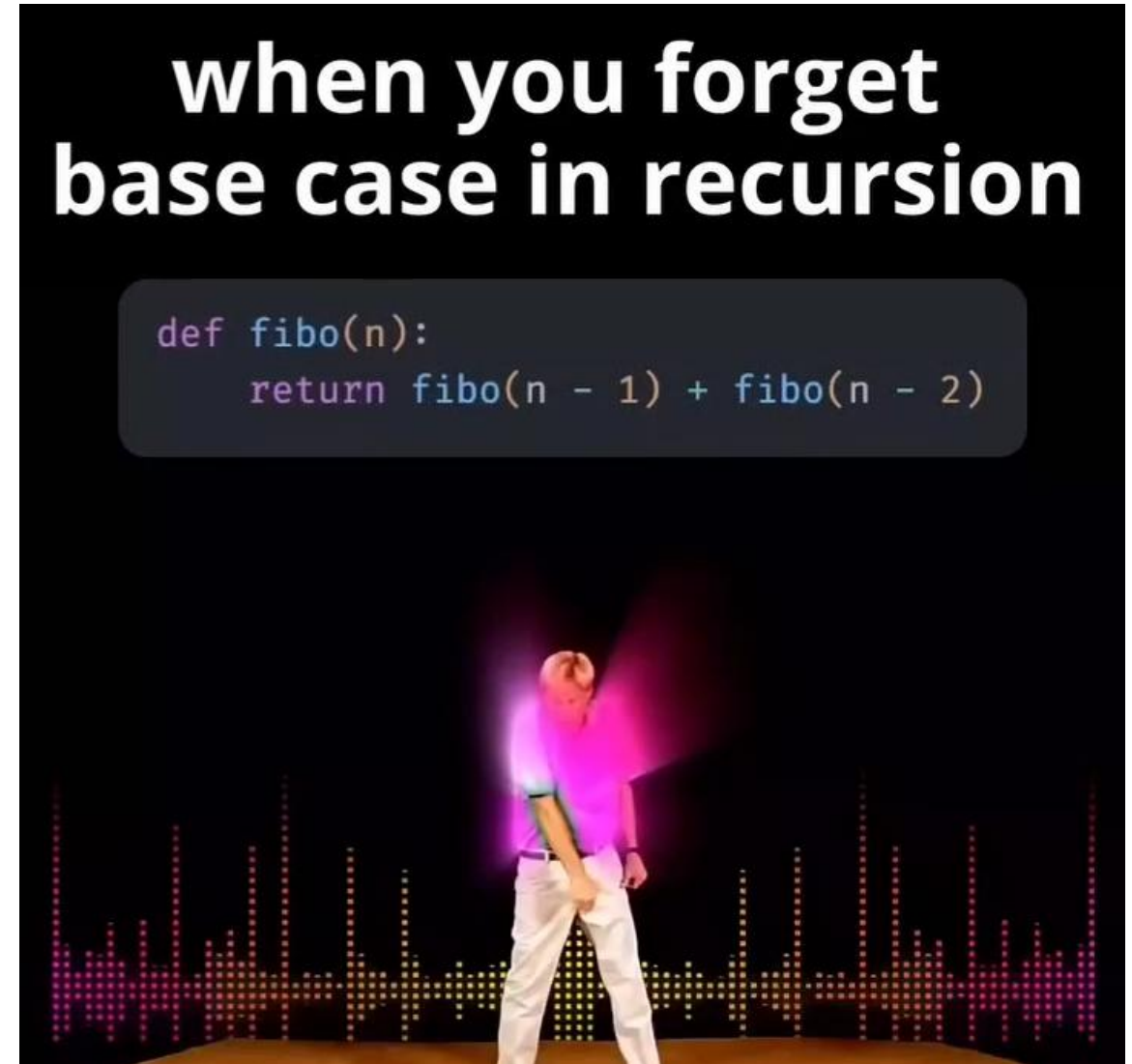
- The first and second elements of the sequence are equal to 1, i.e. $F(1) = 1$ and $F(2) = 1$.
- The n -th Fibonacci element, with $n \geq 3$, is defined as the **sum of its previous two elements**, i.e. $F(n) = F(n - 1) + F(n - 2)$.
- **Task:** Write **two functions** that compute and return the n -th Fibonacci element, for any value of $n \geq 1$. The first function should use a **for** loop, the second should use **recursion**.

Some typical mistakes with recursion

Typical beginner mistakes with recursion include:

- Forgetting the base case,
- Writing a recursive design that does not end up being resolved by the base case at some point. (This means the forward pass never reaches an end and you are stuck with an infinite loop!)

Careful with your function design!



Completely out-of-scope

For those of you who like pain and want to see stuff from Term 3.

What makes a good function?

Definition (a “good” function?): We can define a “good” function as

- 1. A function that is able to operate on any of its “normal” test cases.**
(In the case of a function whose job is to find average value of a list, a normal case consists of a list containing one or more integers)
- 2. A function that is able to cover for special cases.**
(What happens to our function if we use an empty list?
Cases where a division by zero occurs? Etc.
Should it return a None in these cases? An empty list? Etc.
Should it refuse to operate and produce an error message?)
- 3. A function that produces results as fast as possible!**
(Measure it!)

iPython Magic Commands

Definition (iPython Magic Commands):

In Jupyter Notebooks, **magic commands** are special commands that are not a part of the Python language itself but are provided by the iPython/Notebook environment. **Magic commands** are prefixed with percentage signs (**%** or **%%**) and are used to perform various operations.

For instance, we can use the **%timeit** magic command to measure how much time our functions take to run, and compare different designs!

```
def do_thing():  
    # Summing all numbers between 0 and 1,000,000  
    total = 0  
    for i in range(1000001):  
        total += i  
    return total
```

```
%timeit do_thing()
```

27.9 ms ± 4.24 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Here, we have three possible designs for a function, which calculates the sum of all integer numbers between 0 and n .

- All three designs are valid and produce the correct result if used with $n > 0$.

But... Which design is the best then?

```
def do_thing(n):  
    # Summing all numbers between 0 and n  
    total = 0  
    for i in range(n+1):  
        total += i  
    return total
```

```
%timeit do_thing(1000000)
```

29.3 ms ± 678 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
def do_thing_v2(n):  
    # Summing all numbers between 0 and n  
    return sum(range(n+1))
```

```
%timeit do_thing_v2(1000000)
```

21.7 ms ± 448 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
def do_thing_v3(n):  
    # Summing all numbers between 0 and n  
    return n*(n+1)/2
```

```
%timeit do_thing_v3(1000000)
```

73.5 ns ± 1.83 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)

Which Fibonacci design was the best then?

The recursive design, while simpler in appearance, seems to be much slower...

Why is that?

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

```
def fib_for(n):  
    val1 = 1  
    val2 = 1  
    for i in range(n-2):  
        val1, val2 = val2, val1 + val2  
    return val2
```

```
print(fib_rec(10), fib_for(10))
```

```
55 55
```

```
%timeit fib_rec(30)
```

```
68.2 ms ± 647 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit fib_for(30)
```

```
662 ns ± 12.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Which Fibonacci design was the best then?

The recursive design, while simpler in appearance, seems to be much slower...

Why is that?

The issue is that we recalculate the same Fibonacci numbers many times when using the recursion, which is very time consuming!

```
def fib_rec(n):  
    print("Calculating Fib with n = {}".format(n))  
    if n <= 2:  
        return 1  
    else:  
        return fib_rec(n-1) + fib_rec(n-2)
```

```
fib_rec(10)
```

```
Calculating Fib with n = 10  
Calculating Fib with n = 9  
Calculating Fib with n = 8  
Calculating Fib with n = 7  
Calculating Fib with n = 6  
Calculating Fib with n = 5  
Calculating Fib with n = 4  
Calculating Fib with n = 3  
Calculating Fib with n = 2  
Calculating Fib with n = 1  
Calculating Fib with n = 2  
Calculating Fib with n = 3  
Calculating Fib with n = 2  
Calculating Fib with n = 1  
Calculating Fib with n = 4  
Calculating Fib with n = 3  
Calculating Fib with n = 2  
Calculating Fib with n = 1
```

The LRU cache decorator (Memoization)

Definition (**Memoization**):

Memoization is an optimization technique used to speed up the execution of a (recursive) function. It can be particularly useful for functions with recursive or repetitive calls, which are re-computing the same result multiple time.

The idea is to remember/cache the results of expensive function calls and return the memorized result when the same inputs occur again.



The LRU cache decorator (Memoization)

Definition (**Memoization**):

Memoization is an optimization technique used to speed up the execution of a (recursive) function. It can be particularly useful for functions with recursive or repetitive calls, which are re-computing the same result multiple time.

The idea is to remember/cache the results of expensive function calls and return the memorized result when the same inputs occur again.

Memoization is easily done by using the **LRU Cache decorator** (`@lru_cache`) from the functools library!

(Note: functools is a standard Python library, which contains many **decorators** and functions you can use to pimp your functions! Read more, here: <https://docs.python.org/3/library/functools.html>)

LRU cache makes the recursion much faster!

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

```
from functools import lru_cache  
  
@lru_cache(maxsize = None) # None means the cache can grow indefinitely  
def fib_rec_v2(n):  
    if n <= 2:  
        return 1  
    else:  
        return fib_rec_v2(n-1) + fib_rec_v2(n-2)
```

```
%timeit fib_rec(30) # No LRU cache (slow)
```

69.2 ms ± 1.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
%timeit fib_rec_v2(30) # With LRU cache (damn faster!)
```

44.7 ns ± 1.02 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)

```
%timeit fib_for(30) # For loop is not even close!
```

768 ns ± 142 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)

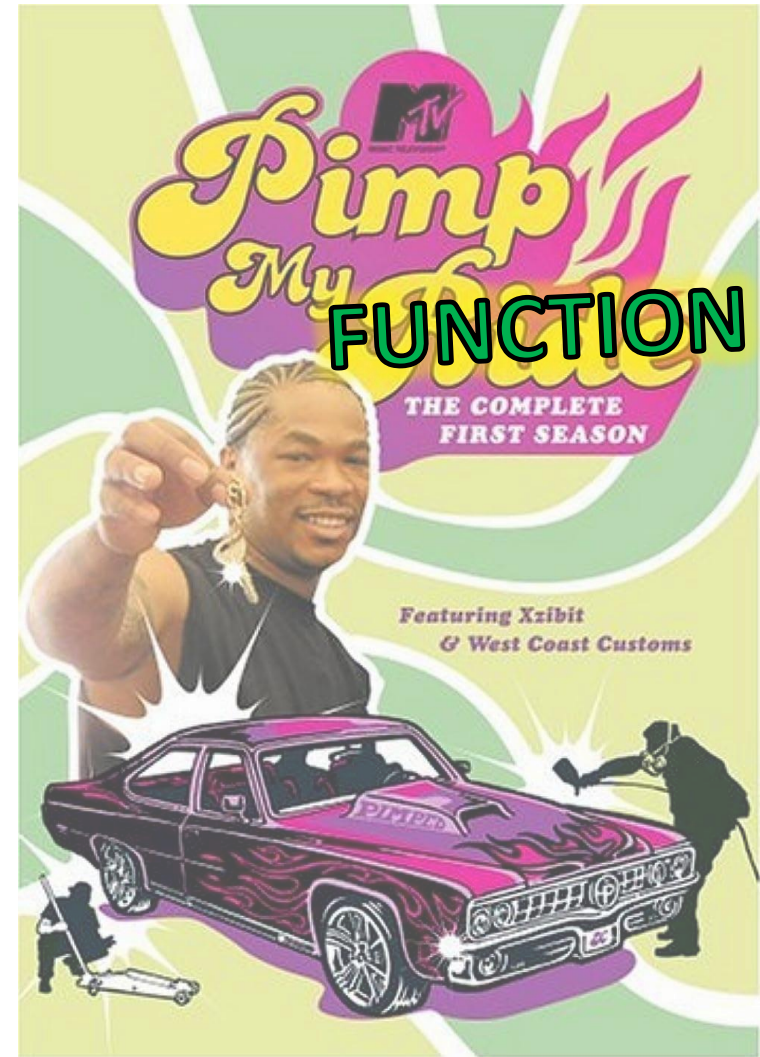
A quick word about decorators

Definition (**Decorators**):

Decorators are a powerful and flexible way to modify or extend the behavior of functions without changing their actual code.

Decorators allow you to wrap another function and execute code before and/or after the wrapped function runs.

They are often used for tasks such as logging, timing, memoization, access control, and more.



A quick word about decorators

Decorators require

- A **decorator** function, which takes a function **func** as a parameter and contains another def statement for the **wrapper** function, inside.
- Within this **wrapper** function, operations can occur before and/or after calls made using the function call **func()**.
- Later on, use the **@** symbol and the name of your **decorator** on a **third** “decorated” function.

- When running third function, the **decorator/wrapper** will execute its operations before and/or after the function call!

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper
```

```
@my_decorator  
def say_hello():  
    print("Hello!")
```

```
say_hello()
```

```
Something is happening before the function is called.  
Hello!  
Something is happening after the function is called.
```


Using decorators to measure the time taken by your function runs!

- For instance, we can use the `time()` function from the `time` library, which gives us a way to measure time...
- And create a time measuring decorator to measure how much time our functions take to run!

Extremely useful to assess if one function design is better than another! (%timeit?)

```
from time import time

def my_decorator(func):
    def wrapper():
        function_name = func.__name__
        print("Started measuring time used by function {}".format(function_name))
        start = time()
        func()
        stop = time()
        time_taken = round(stop - start, 5)
        print("And... Done. Function took {} seconds to run!".format(time_taken))
    return wrapper

@my_decorator
def do_thing():
    # Summing all numbers between 0 and 1,000,000
    total = 0
    for i in range(1000001):
        total += 1
    return total

do_thing()
```

```
Started measuring time used by function do_thing().
And... Done. Function took 0.02268 seconds to run!
```

slido

Please download and install the Slido app on all computers you use



What will be printed?

① Start presenting to display the poll results on this slide.



What will happen if a recursive function in Python does not have a base case?

① Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use



What to put in blank A and blank B to create a recursive function that sums elements of a list of integers?

① Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use



What will be printed here?

① Start presenting to display the poll results on this slide.



Which of these two functions will be the fastest to compute the 50th Fibonacci number?

① Start presenting to display the poll results on this slide.

Just a few additional questions
(For those looking for a challenge)

Some more practice for today: on for loops and recursion.

Easy challenge: What is, in fact, the fastest way to calculate the n -th Fibonacci number?

(Recursion, Recursion with memoization, for loop, or... Something else?)

Difficult challenge: In the Extra Challenge folder of today's lecture, an extra "difficult" challenge for you to try.

Finished everything and looking for a real challenge? (Finish the rest first!)

Instructions

Write a function **f()** that receives a list of integers, which are always sorted in ascending order, without any duplicates.

For instance [1, 2, 4, 6] is a valid list. The list [1, 4, 2, 6] is not a valid list (not sorted in ascending order), and [1, 2, 2, 4, 6] is not (has duplicates).

The function should return, as a result, a list of all the possible sublists of elements, in ascending order.

You may use subfunctions to break down the operations in the function **f()**, if it helps you to make the code cleaner.

Test case (only one is given, but not the only one?) ¶

If the list given to **f()** is [1, 2, 4, 6], then it should return, as a result the following list of list, with the elements in that order:

```
[[], [1], [2], [4], [6], [1, 2], [1, 4], [1, 6], [2, 4], [2, 6], [4, 6], [1, 2, 4], [1, 2, 6], [1, 4, 6], [2, 4, 6], [1, 2, 4, 6]]
```