Practice more

As usuall, you can find exercises in these yellow frames. None of them are mandatory.
The first exercise is about integrating C++ into your program.
More familiar topics start from Exercise 3.

1 Compute Fibonacci numbers in Python, with Numba, and in C++

To see how compiled languages (such as C++) can be more efficient than interpreted ones (such as Python), we will calculate Fibonacci numbers using Python, C++ and Numba that speeds up your Python code and see how it compares.

1.1 Python

For pure python, use a recursive version for computing the Fibonacci number:

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

1.2 C++

We use `ctypes` to communicate with C++ code from Python. This is a built-in module in Python that is written to communicate with code written in the language `c`, but with a simple modification it also works for C++. Note that there are a lot of ways to do this in Python, some other notable modules are `Cython` and `SWIG` (https://realpython.com/python-bindings-overview/).

The example of script written in C and the way of integrating it to python is given below.

1.3 Files description

The files in `VA_files.zip` are:

- `person.cpp` C++ code for the module `person` with a class `Person`.
- `Makefile` The file that gives instructions to the `make` command how the C++ code in `person.cpp` should be compiled.
- `person.py` Python code for the module `person`.
- `VA_C.py` A test Python program that uses the module `person`.

We now briefly describe the contents of these files, and their function in the test program `VA_C.py`.

person.cpp

```
1  #include <cstdlib>
2  // Person class
3
4  class Person{
5          public:
6                  Person(int);
7                  int get();
8                  void set(int);
9          private:
10                 int age;
11         };
12
13 Person::Person(int n){
14         age = n;
15         }
16
17 int Person::get(){
18         return age;
19         }
20
21 void Person::set(int n){
22         age = n;
23         }
```

- `#include <cstdlib>` on line 1 means that definitions in `cstdlib` are included.
- Comments in C++ are made with `//` (the rest of the line is a comment) like on line 2 or with `/* comment */` which can span over several lines.
- `Person` (line 4–11) is a class with three public methods (constructor, `get`, and `set` on lines 6–8) and a private value (the persons `age` on line 10).
- Data types must be declared in C++ which is not necessary in Python. In this code we only have `int`, which stands for integer. Other examples are `double` (floating point number) and `char` (character).
  - `Person(int);` means that the constructor takes a variable of type `int` as an argument.
  - `int get();` means that the method `get` does not have any argument and returns an `int`
  - `void set(int);` means that the method `set` requires an argument of type `int` and returns nothing, `void`.
- Note that expressions are finalized with a `;`. C++ does not take into account white space/indentation like Python, and instead uses `{}` to organize code blocks.

In the end of `person.cpp` there is also,

person.cpp (cont.)

```
1  extern "C"{
2          Person* Person_new(int n) {return new Person(n);}
3          int Person_get(Person* person) {return person->get();}
4          void Person_set(Person* person, int n) {person->set(n);}
5          void Person_delete(Person* person){
6                  if (person){
7                          delete person;
8                          person = nullptr;
```

```
9                           }
10                      }
11              }
```

This code, which is in `c`, is a bridge code so that the Python module `ctypes` can communicate with the C++ code. `Person_delete` is a destructor, which removes an object.

As mentioned before, C++ code has to be compiled every time it has been modified (otherwise the compiled binary will be of an older version of the code).

To compile the code in `person.cpp`, the following two commands are to be executed in terminal (can be done in Linux machines):

```
$ g++ -std=c++11 -c -fPIC person.cpp -o person.o
$ g++ -std=c++11 -shared -o libperson.so  person.o
```

These command create a so-called "shared object" `libperson.so` which Python can import. The compiler on the Linux machines is `gcc`, and `g++` the C++ compiler.

Since every time you have edited the file `person.cpp`, you have to run the two commands above you can instead use the command `make` which gets its commands from a file called `Makefile`.

Makefile

```
# Makefile for VA
all:
        g++ -std=c++11 -c -fPIC person.cpp -o person.o
        g++ -std=c++11 -shared -o libperson.so  person.o
clean:
        rm -fr *.o *.so __pycache__
```

Comments in makefiles start with a #-sign, and then the rest of the line is a comment, like on line 1. It is important to have a <tab> on the rows that start with `g++` and `rm`. To run the two `g++` commands, all one have to do in the terminal is to type `make` (or `make all`). This is because `all:` is the first block of commands. The second block of commands is named `clean:` which runs the command on the last line (`rm -fr *.o *.so __pycache__`, which deleted files `*.o`, `*.so` and the directory `__pycache__`); it is called by running `make clean` in the terminal.

You can add other blocks of consecutive commands in the make file, just have a unique identifier like `all:` and `clean:` above. A more advanced build system of the same type is `CMake` that in a similar way can compile on all major platforms (like Windows, macOS, Linux, ...)

To read more about make files you can for example read https://opensource.com/article/18/8/what-how-makefile and https://www.gnu.org/software/make/manual/make.html, and about `Cmake` on https://cmake.org/.

The actual Python module (that you will import with `import person` in other Python programs) is

person.py

```
1
2  """ Python interface to the C++ Person class """
3  import ctypes
```

```
4  lib = ctypes.cdll.LoadLibrary('./libperson.so')
5
6  class Person(object):
7          def __init__(self, age):
8                  lib.Person_new.argtypes = [ctypes.c_int]
9                  lib.Person_new.restype = ctypes.c_void_p
10                 lib.Person_get.argtypes = [ctypes.c_void_p]
11                 lib.Person_get.restype = ctypes.c_int
12                 lib.Person_set.argtypes = [ctypes.c_void_p,ctypes.c_int]
13                 lib.Person_delete.argtypes = [ctypes.c_void_p]
14                 self.obj = lib.Person_new(age)
15
16         def get(self):
17                 return lib.Person_get(self.obj)
18
19         def set(self, age):
20                 lib.Person_set(self.obj, age)
21
22         def __del__(self):
23                 return lib.Person_delete(self.obj)
```

- On line 2 the module `ctypes` is imported, which handles the bridging to the `C` and C++ code.
- On line 3 the shared object `libperson.so`, from the compiled C++ code, is imported.
- From line 5 we have the class defintion of an `Person` in Python. In the constructor argument types (`argypes`) and return types (`restype`) are defined for the different functions in the `extern "C"` part of `person.cpp`. The types are here `ctypes.c_int` (`int`) and `ctypes.c_void_p` (`void`).
- On lines 15 and 18 the Python methods for `get()` and `set(val)` are defined.

The last file in `VA_files.zip` is a test code in Python that uses the module `person`.

VA_C.py

```
1  #!/usr/bin/env python
2
3  from person import Person
4
5  def main():
6          f = Person(5)
7          print(f.get())
8          f.set(7)
9          print(f.get())
10
11 if __name__ == '__main__':
12         main()
```

- The first line is a so-called *shebang* (https://en.wikipedia.org/wiki/Shebang_(Unix)) that defines what program should be run if we execute the script directly by `$ ./VA_C.py` (may be necessary to make the script executable by `$ chmod 755 VA_C.py` first). One can also execute the script with `$ python VA_C.py`.
- On line 3 `Person` is imported from the module `person`.
- On lines 6–7 an object `f` of type `Person` is created with a value of 5, which is printed.

- On lines 8–9 the value of `f` is changed to 7, which is printed.

1.4 Numba

While Python in general is slow, it is also easy to read. If a piece of code cannot be understood, it cannot be maintained easily and will quickly become obsolete. Especially for scientific uses, being able to peer-review code and easily try different things is essential, meaning readability is of particular value. To get the best of both worlds and try to speed up Python without needing to write C++-code (sometimes called "the two language problem"), special libraries exist. One such library is *Numba*, which you will use to speed up your Python code and compare it to native Python and C++.

Numba is a so-called "Just-In-Time" (JIT) compiler. Essentially, it compiles a piece of code at run time (as opposed to C++, which is compiled before execution), meaning the first time code is run, it is converted to fast machine code. This is particularly useful on loops, where many executions of the same piece of code are conducted. The one-time compilation will greatly accelerate performance of the following executions.

In order to be able to accelerate code beyond Python-speeds, Numba needs to make a few assumptions about the code, e.g. regarding data types (like in C++, as opposed to generic Python that uses duck typing). We consider an example:

```
1  # Example of a dynamically written function
2  def maxOfList():
3      result = 'Nothing yet'
4      lst = [1, 8, 3, 9, 14]
5      for x in lst:
6          if result=='Nothing yet':
7              result = x
8          elif x > result:
9              result = x
10     return result
```

This code would run perfectly fine in Python, but cannot run with Numba. It has to do with the fact that the variable `result` is of different types in different parts of th code; first a string (`str`) on line 3, then an integer (`int`) on lines 7 and 9. Hence, the comparison on line 8, `>`, is not well defined (might be comparing an integer to a string). Numba analyses the whole function when it is compiling it to make it faster, and we have here an ambiguity that is not acceptable for Numba. Numba is not as strict as C++, which does not handle any conversions at all, but much more strict than standard Python. For the interested, you have here some examples of cases where you need to put extra care for Numba to work and give faster code https://numba.readthedocs.io/en/stable/. However, your task using Numba is so simple you will not have to worry about this.

To use Numba, you first need to install the Numba-library. This is easily done using `pip` on the Linux machines, similarly to installing `matplotlib` or similar libraries. To install, simply open a command window and type:

```
\$ pip install numba
```

To apply Numba to a Python-function, you will use a *function decorator*. You can read more about these online, but essentially these are functions that modify another function in some way (in our case: speed it up through compilation).

The syntax is adding `@decoratorName` above a function definition. So to JIT-compile a function using Numba, you simply add

```
@njit
```

above your function definition. Note that you must have imported Numba to your program:

```
from numba import njit
```

Example:

```
@njit
def someFunction(x):
    ...
```

Numba will then JIT-compile the decorated function the first time it is called.

1.5 Exercise to compare Python, C++ and Numba

Exercise 1: Fibonacci numbers by three methods

There are three ways of computing Fibonacci numbers described above: in Python, in C++, with Numba. Measure how much time the three methods take to complete, for some different Fibonacci numbers.

Make timings for `fib_py(n)`, `fib_numba(n)` and `f=Person(n);f.fib()` (the C++ code) for $n = 30, \ldots, 45$. Use for example `time.perf_counter`. Save the results to a figure (.png) (x-axis is n and y-axis is seconds). Use, for example, `matplotlib.pyplot`.

Hint 1: One way to implement this in C++ is to create one public and one private method in the `Person` class. Do not forget to modify the `C` bridging code, and also the Python module file `person.py`.

Create an "equivalent" C++ method as the Python method `fib_py` above, so that the following works in a Python script

```
f=Person(n)
f.fib()
```

Hint 2: Class person is given as an example. You can write script for Fib numbers in C without using class person at all.

Exercise 2: fib(47)

What is the result for Fibonacci with $n = 47$? If it is negative, expalin why.

2 Recursion

Example 1: The exchange problem

In how many ways can you exchange a Euros in 1, 5, 10, 50 and 100 coins/notes?
(e.g. 90 Euros $50 + 4 \times 10$, 9×10, $8 \times 10 + 10 \times 1$ etc.)

To solve the problem recursively, we have to find smaller problems of the same type.

There are two ways to form smaller problems: find the solution for a smaller amount and find a solution with fewer coins.

We can divide the exchange attempts into two groups:

- those who *do not use* a coin of certain kind, e.g. the first and
- those who *use* the first kind of coin

The solution to the problem can now be formulated:

The number of ways to exchange a Euros when using n different kinds of coins is

1. the number of ways to exchange a Euros using all but the first kind of coin *plus*
2. the number of ways to exchange $a - d$ Euros using all kinds of coins where d is the value of the first type of coin.

The first of these problem is smaller than the original problem because it uses fewer types of coins and the second is smaller because it exchanges a smaller amount.

Assume that the coin types are represented in a list `coins`.

Initial function

```
1 def exchange(a, coins):
2     return exchange(a, coins[1:]) + \
3            exchange(a-coins[0], coins)
4
```

Which cases can we use to terminate the recursion?

- $a = 0$ which means a successful attempt. Count!
- $a < 0$ which means a failed attempt Do not count!
- the list `coins` is empty which means a failed attempt. Do not count!

Final version

```
def exchange(a, coins):
    if a == 0:
        return 1
    elif (a < 0) or len(coins) == 0:
        return 0
    else:
        return exchange(a, coins[1:]) + \
               exchange(a-coins[0], coins)
```

Exercise 3: Exchange

Modify the program `exchange` to print the exchanges being made.

Exercise 4: Recursion depth

Investigate how large n your computer can handle in the the recursive function `fac`!

Exercise 5: Exchange with memoization.

If you tested the exchange program with large values of `a`, you probably noticed that the execution time grew rapidly. The algorithm is recursive but it is difficult to analyze because time does not just depend on the *sum* to be exchanged but also on the *number* of coins and their values. Here, we can use memoization to avoid solving the same problem repeatedly. However, you need a matrix with the number of coins on one axis and their values on the other.

Rewrite the `exchange` function to use memoization. What time did it take to calculate large sums such as 1000 and 2000? Note that it is possible to write a solution for a=2000 without increasing recursion depth limit!

Exercise 6: Recursive zip.

Write a recursive function `zippa(l1, l2)` which returns a list of every other element from the list `l1` and every other from the list `l2`.

Example: The call

```
zippa(['a', 'b', 'c'], [2, 4, 6, 'x', 10])
```

should return the list

```
['a', 2, 'b', 4, 'c', 6, 'x', 10]
```

Note: You must *not* use the built-in method `zip`!

Exercise 7: Divide

Write a recursive function (i.e. without iteration) `divide(t, n)` which calculates how many times `n` goes in `t` (i.e. an *integer division*) *without* using division or multiplication. You may assume that `t` and `n` are greater than 0.

3 Linked lists

Exercise 8: Mean of linked list

Write a method which calculates and returns the mean of the numbers in the list.

Exercise 9: The index operator.

In regular lists, you can use an index to reach an element at a certain position. Implement the index operator for linked lists!

Note that you do not want to be able to use the index operator to *change* the value because that may destroy the sorting property!

4 Binary trees

A binary tree with k filled levels contains n nodes there

$$n = 1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

dvs.

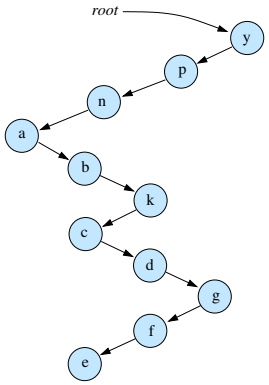
$$k = \log_2(n+1) \approx \log_2 n$$

This is the *minimum height of a binary tree with n nodes.*

In such a binary search tree, the work for

- searching is $\mathcal{O}(\log n)$
- insertion is $\mathcal{O}(\log n)$
- deletion is $\mathcal{O}(\log n)$

The largest possible height is obtained if each node has only one child:



In such a binary search tree, we have at least in the worst case

- searching is $\Theta(n)$
- insertion is $\Theta(n)$
- deletion is $\Theta(n)$

We would like to know the average work to search (add, delete) a key in an 'average' tree.

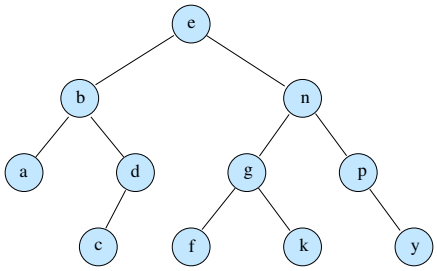
The following three measures are used to analyze how good trees are for search / insert / delete:

- *height*: for worst case analysis
- *internal path length*: for successful search analysis

- *external path length*: for unsuccessful search analysis

The *internal path length* (*ipl* or I) is defined as the sum of all nodes' *levels*

The root has level 1, the root's child level 2, etc.



Adjacent trees have internal path length

$$i = 1 + 2 \cdot 2 + 4 \cdot 3 + 4 \cdot 4 = 33$$

The average work to find a stored key ("a successful search") is

$$\frac{i}{n} = \frac{33}{11} = 3$$

The *external path length* (*epl* or E) is defined as the sum of the levels of the places where a new node would end up.

In the example tree above, it will be

$$E = 4 + 4 + 5 + 5 + 4 + 5 + 5 + 5 + 5 + 4 + 5 + 5 = 56.$$

Inserting a new key requires on average $E/(n+1)$ node visits (there are $n+1$ possible locations and we count the creation of the node as a visit).

The following relationship between internal and external road length applies

$$E = I + 2n + 1$$

which can be shown with induction.

It can be shown that the average internal path length for all trees formed by all $n!$ permutations of n keys is

$$1.39 \cdot n \log_2 n + \mathcal{O}(n)$$

Thus, to find a key in the "average tree" takes on average $1.39 \cdot \log_2 n + \mathcal{O}(1)$ operations.

For example, searching a tree with 10^6 nodes thus requires on the order of approximately $1.39 \cdot \log_2(10^6) \approx 28$ node visits. The corresponding search in a linked list would require $500,000$ node visits.

Insertions and deletions can also be made with $\mathcal{O}(\log(n))$ operations.

Thus, a binary search tree is an efficient structure for *storing*, *searching* and *deleting* nodes with keys. Since it also maintains a sorting of the keys, operations such as *find the smallest* and *find largest* are efficient.

However, there are many insertion orders that lead to so called *degenerated* trees i.e. trees where almost no nodes have two children. See e.g. the figure at the top of the previous page. The operations on the tree then have the same complexity as the corresponding operations on a list i.e. typically $\Theta(n)$. This will for example happen if you insert the keys in sorted order. The logarithmic complexity applies to the 'average tree'. There are balancing mechanisms that ensure that the trees do not degenerate i.e. they *guarantee* $\mathcal{O}(\log n)$ for the operations. The insertion algorithms are modified so that they adjust the

tree if it becomes too unbalanced. See for example AVL-trees* and Red-black trees *. In these, all the operations we have discussed are $\mathcal{O}(\log n)$ - even in the worst case.

Exercise 10: The method `ipl`.

Write the method `ipl (self)` which calculates and returns the internal path length as defined above.

Exercise 11: Verify the theory with experiments.

Experimentally verify that the internal path length (IPL) grows as $1.39n\log_2(n)+O(n)$ in trees built up with random numbers. At the same time, try to see how the height seems to depend on the number of nodes.

Hint: Write a function (not a method) `random_tree(n)` which creates and returns a tree with `n` random numbers. Use `random.random()` which returns random numbers (floats) in the range $[0, 1)$. Then write the code that examines trees of different sizes at the end of the `main` function. Print the observed height and $\frac{\text{IPL}}{n}$. How well does that agree with the theory? What can you guess about the height?

Exercise 12: The method `remove_all`.

Write a recursive method `remove_all(self, x)` which removes *all* nodes which contains `x` as data. The method should return the number of removed nodes.

Exercise 13: The method `insert`.

Write the `insert` method without using recursion.