Introduction

Memoization methods consist in storing an intermediate result and returning it whenever this is needed.

These methods reduce the time complexity but deteriorates memory use.

Implementation

Theory

In Python, the `lru_cache` decorator of the `functools` package is a way to do memoization. Recall that a decorator can be applied using `@` before the line of a function/class definition.

In the background, `lru_cache` stores the results of each function call and returns them when the function is called one more time with the same arguments.

`lru_cache` takes a `maxsize` argument so that only a certain number of function calls results are stored ('lru' = 'least recently used'). Whenever the function result has a low memory footprint, one can set `maxsize=None` to leverage the limit.

Example

Code

Let's build a classical case study: the Fibonacci sequence. Let's define 2 versions:

- without memoization
- with memoization, using `lru_cache`

In [1]:
```
from functools import lru_cache

def fib_simple(n):
    if n < 2:
        return n
    return fib_simple(n-1) + fib_simple(n-2)

@lru_cache(maxsize=2**20)  # maxsize must be specified as a power of 2
def fib_memoized(n):
    if n < 2:
        return n
    return fib_memoized(n-1) + fib_memoized(n-2)
```

Results

In [2]:
```
%timeit fib_simple(10)
```

```
19.5 µs ± 1.36 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [3]:
```
%timeit fib_memoized(10)
```

```
88.1 ns ± 10.6 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops eac
h)
```

There is a time sped up of approximately a factor 220!

Details

the decorated function has a `cache_info()` method that gives some information regarding function calls:

In [2]:
```
fib_memoized.cache_info()
```

Out[2]:
```
CacheInfo(hits=0, misses=0, maxsize=1048576, currsize=0)
```

Explanations:

- `hits` : number of function calls that were handled by the cache, i.e. for which the function result was already known and stored. This number is very large since `%imeit` performs several runs.
- `hits` : number of function calls that were not led by the cache. This typically corresponds to the first calls of the function, when the cache is empty.
- `currsize` : number of stored call results

Caution

`lru_cache` relies on a dictionary to store the results of a function call. Yet, keys of a dictionary must be hashable.

Thus, most of **mutable objects cannot be arguments of a function decorated with `lru_cache`**.

Below is a comparison of a `tuple` argument (immutable) and `list` argument (mutable, unhashable).

In [3]:
```
import numpy as np

@lru_cache()
def memoized_function(arg):
    print(arg)
```

In [4]:
```
memoized_function((1, 2, 3))
```

```
(1, 2, 3)
```

In [5]:
```
memoized_function.cache_info()
```

Out[5]:
```
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
```

In [6]:
```
memoized_function([1, 2, 3])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 memoized_function([1, 2, 3])

TypeError: unhashable type: 'list'
```

When to use memoization

The `lru_cache` decorator must be used on functions that **meets the 3 following criterias**:

- it takes a long time to run
- it is frequently called with the same arguments (non mutables)
- it returns intermediate results that have no particular scientific meaning