Optimisation methodology

Theory

The following methodology can be used to reduce the time complexity of any code:

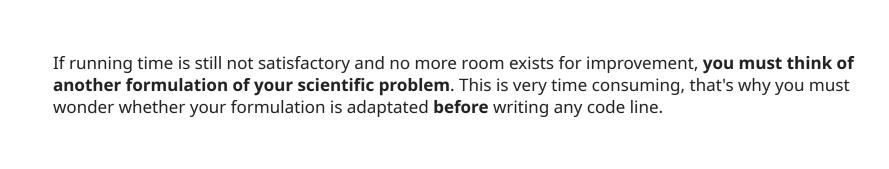
- 1. Evaluate the total running time (timing)
- 2. Sort each function *f* by running time (profiling)

t(*f*)

3. Estimate for each function the possible room for improvement

 $\Delta t(f)$

- 4. Optimize the function f for which $t(f) \times \Delta t(f)$ is the smallest.
- 5. Check that the code is still operating properly
- 6. Go back to step 1 if running time is still too high



Example

Let's assume we have some code that calls 3 functions:

- the first function call is 90% of total running time
- the second function call is 1%
- the third is 9%

Achieving a 10% time reduction on the first function call (which seems possible) will have more effect than a 90% time reduction on the third call (which seems difficult)

Tools

Presentation

Python comes with some tools to measure the running time of some code instructions. There are 2 steps:

1. Measuring the total running time: **timing**.

This can be done using the timeit package.

2. Measuring the running time of each function call: **profiling**.

This can be done using the cProfile package.

iPython

Instead of directly using timeit and cProfile, it is recommended to use the magick commands %timeit and %prun.

These commands require the Python shell to be a *iPython* shell.

iPython can be used:

- by installing the ipython package and by opening a shell using the ipython command
- by installing either Jupyter notebook or Jupyter lab

In addition to the magic commands, *iPython* shells come with a better color code and indentation handling.

Example

Case study definition

Let's define a very naive function and analyse its algorithmic complexity. **This function is designed specifically not to be efficient**, it must not be used.

```
In [1]: def build_list(k):
             result = []
             for j in range(1, k):
                 result.append(j)
             return result
        def sum_list(var):
             s = 0
             for e in var:
                 s += e
             return s
        def f(n):
             assert n >= 3
             result = []
             for j in range(3, n):
                 var = build_list(j)
                 s = sum_list(var)
                 result.append(s)
             prod = 1
             for e in result:
                 prod *= e
```

Processing math: 100%

What does this function do? It computes the product of the sum of q-1 first non zero integers, for $q \in [3, n-1]$:

$$f: n \to \prod_{q=3}^{n-1} \sum_{p=1}^{q-1} p$$

Running time: timing

Let's measure the total running time using <code>%timeit</code> . The percent sign <code>'%'</code> decribes *iPython* magic commands: it tells the Python interpreter that this is not a common variable.

timeit measures the running time several times in a row in order to remove statistical noise.

Processing math: 100%

f function expects a value for variable n. Let's suppose n=10000 is a typical value for the problem we want to solve.

```
In [14]: %timeit f(10000)
```

 $5.84 \text{ s} \pm 404 \text{ ms}$ per loop (mean \pm std. dev. of 7 runs, 1 loop each)

The command described above performs 7 runs (-r 7) of calling f(10000) 1 time (-n 1). Average is done and the mean and std of running time among the different runs is returned. %timeit decides alone what are the good values for -r and -n, but it can also be specified:

```
In [16]: %timeit -r 2 -n 2 f(10000)
```

5.7 s \pm 195 ms per loop (mean \pm std. dev. of 2 runs, 2 loops each)

An object mode also exists: instead of printing the results, they are stored in a dedicated instance:

```
In [17]: running_time = %timeit -o -r 2 -n 2 f(10000)
print(running_time.all_runs)
print(running_time.best)
print(running_time.worst)

5.75 s ± 54.6 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
[11.390800094988663, 11.609388401004253]
5.695400047494331
5.8046942005021265
```

Processing math: 100%

Running time: profiling

Profiling is done using %prun. The result is saved in a file (-T option).

```
In []: %prun -T prun_results.txt f(10000)
```

```
50014995 function calls in 10.999 seconds
 Ordered by: internal time
 ncalls tottime percall cumtime percall filename:lineno(function)
   9997
           5.946
                   0.001
                            8.356
                                    0.001 3227873083.pv:1(build list)
                            2.411 0.000 {method 'append' of 'list' objects}
49994997
           2.411
                   0.000
                  0.000 2.353 0.000 3227873083.py:7(sum list)
           2.353
   9997
           0.289
                  0.289
                           10.999 10.999 3227873083.py:13(f)
                   0.000 10.999 10.999 <string>:1(<module>)
           0.000
                   0.000 10.999 10.999 {built-in method builtins.exec}
           0.000
                                    0.000 {method 'disable' of 'lsprof.Profiler' objects}
           0.000
                   0.000
                            0.000
```

There are several columns:

- on the right, the executed function (file name:line number)
- 1st column on the left (ncalls): number of times this function is called
- 2nd column (tottime): the time spent in this function (excluding the time spent in subfunctions)
- 3rd column (percall): the sum of tottime divided by ncalls

Optimize the relevant code section

The profiling results show that the most time-consuming function is build_list. And this part seems easy to optimize. Thus, optimization of time complexity starts here.

Reminder

Optimization is always done at **the very last moment**, once the code is stable and no functionnalities are to be added.