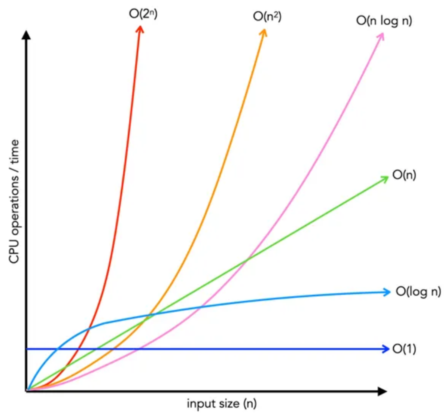

Figure 1: Complexity : Size VS Computing

TD #1: Python

Jonathan Rouzaud-Cornabas (jonathan.rouzaud-cornabas@insa-lyon.fr)

2 week (vacations are not counting so mid November) after the practical session, you should send a report (Evaluation of performance and a short analysis for each part) of what you have done and also your completed codes. The performance evaluation and analysis is as important as the code itself.

You can do the TP alone or by pair (not more). This report will be graded.

The TP contains 5 parts. Each one is more complexe than the previous one. The bare minimum is the first 2 parts.

Performance evaluation methodology:

1. You should automatize the multiple execution (given a range for each parameter)
2. You should repeat each execution multiple time (at least 10 times)

1 Data structures

As an introduction, you will try and examplify the complexity of basic operation on classical containers / data structures (see figure 1).

You will find a Jupyter Notebook on moodle named complexity.ipynb. It includes an insertion sort algorithm that you will optimize.

Question 1.1 Load the notebook and follow the instructions

Question 1.2 Explain the performance

2 Profiling

You will find a Jupyter Notebook on moodle named profiling.ipynb. It includes a code example that you should profile to find the issue.

Question 2.1 Profile you code with cProfile, you can use snakeviz to display the result

Question 2.2 Propose a modification to improve performance and explain why

3 Computing PI

We provide you with two algorithms to compute PI.

First one is computing PI through integral:

Require: num_trial
  $step = 1.0/num\_trial$
  for j = 0; $j < num\_trial$; j++ **do**
    $x = (j - 0.5) * step$
    $sum = sum + 4.0/(1.0 + x * x)$
  end for
  return $sum * step$

Second one is computing PI through the metropolis algorithm:

Require: $num\_trial, seed$
  $counter = 0.0$
  $random\_generator.seed(seed)$
  for j = 0; $j < num\_trial$; j++ **do**
    $x\_val = random\_generator.rand()$
    $y\_val = random\_generator.rand()$
    $radius = x\_val * *2 + y\_val * *2$
    if $radius < 1$ **then**
      $counter+ = 1$
    end if
  end for
  return $4 * counter/num\_trials$

Both of them should be more precise when you increase the number of trial (*i.e.* the number of samples).

3.1 Native Python Implementation

Question 3.1 Implement both of them

Question 3.2 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000

Question 3.3 Draw figures to display performance

Question 3.4 Explain your performance

3.2 Generator and Lambda

Question 3.5 Modify both of your implementation as generator functions

Question 3.6 Modify both of your implementation as lambda

Question 3.7 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000

Question 3.8 Draw figures to display performance

Question 3.9 Explain your performance

3.3 Numpy

Question 3.10 Reimplement both of them in Numpy

Question 3.11 Use `vectorize` to improve performance

Question 3.12 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000

Question 3.13 Draw figures to display performance

Question 3.14 Explain your performance

3.4 Cython

Question 3.15 Reimplement both of them in Numpy + Cython

Question 3.16 Use `vectorize` to improve performance

Question 3.17 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000

Question 3.18 Draw figures to display performance

Question 3.19 Explain your performance

3.5 Numba

Question 3.20 Reimplement both of them in Numba

Question 3.21 Reimplement both of them in Numba + Parallelism

Question 3.22 Use `vectorize` to improve performance

Question 3.23 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000

Question 3.24 Draw figures to display performance

Question 3.25 Explain your performance

3.6 Numba

Question 3.26 Reimplement both of them in Pandas

Question 3.27 Reimplement both of them in Dask

Question 3.28 Use `vectorize` to improve performance

Question 3.29 Evaluate the performance for number of steps : 1000000, 100000000, 10000000000, 1000000000000

Question 3.30 Draw figures to display performance

Question 3.31 Explain your performance

4 Matrix multiplication

You will find a Jupyter Notebook on moodle named matrix_multiplication.ipynb.

Naive matrix multiplication algorithm:

Require: A, B, C
  for k *in* $range(0, length(A[0])$ **do**
    for i *in* $range(0, length(A)$ **do**
      $t = A[i][k]$
      for j *in* $range(0, length(B[0])$ **do**
        $C[i][j] += t * B[k][j]$
      end for
    end for
  end for

Question 4.1 Implement the naive matrix multiplication algorithm in native Python

Question 4.2 Implement the dot-product matrix multiplication (using Numpy)

Question 4.3 Implement the matrix multiplication using Numpy

Question 4.4 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 4.5 Draw figures to display performance

Question 4.6 Explain your performance

Question 4.7 Modify the code to use Cython

Question 4.8 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 4.9 Draw figures to display performance

Question 4.10 Explain your performance

Question 4.11 Modify the code to use Numba (with and without parallelism)

Question 4.12 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 4.13 Draw figures to display performance

Question 4.14 Explain your performance

Question 4.15 Modify the code to change the type of the matrix from double to float

Question 4.16 Evaluate the performance for:

- N : 1000, 4000, 8000, 12000, 18000
- M : 1000, 4000, 8000, 12000, 18000

Question 4.17 Draw figures to display performance

Question 4.18 Explain your performance