How to write faster python code

1. Toy problem
2. Performance analysis (CPU)
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. Wrapping up

1. Toy problem

1. **Toy problem**
    - A. Quicksort algorithm
    - B. Code example
2. Performance analysis (CPU)
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. Wrapping up

1.1. Quicksort algorithm

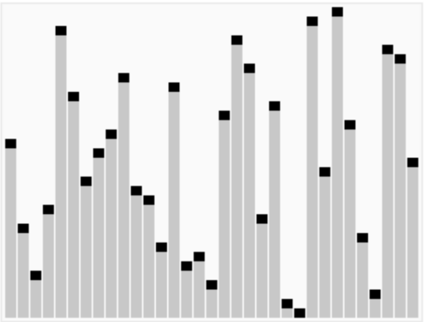
> *Given a **collection** of **orderable** elements, how to **sort** them efficiently with regard to **computation time** ?*

There exists many different sorting algorithms. The fastest, most general purpose, and consequently the most commonly used is the **quicksort** algorithm

1.1. Quicksort algorithm

> *Given a **collection** of **orderable** elements, how to **sort** them efficiently with regard to **computation time** ?*

There exists many different sorting algorithms. The fastest, most general purpose, and consequently the most commonly used is the **quicksort** algorithm



Quicksort

1. Pick an element from the collection, it is called the **pivot**
2. Partition the elements in two subparts such that:
   - In the left part they are **smaller or equal** to the **pivot**
   - In the right part they are **greater** than the **pivot**
3. Position the **pivot** between the two parts
4. Repeat this process on the subparts containing more than one element

1.2. Code example

1. **Toy problem**
    A. Quicksort algorithm
    B. **Code example**
        a. First implementation
        b. Test data (benchmark)
        c. Driver program
2. Performance analysis (CPU)
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. Wrapping up

1.2.1. First implementation

1.2.1. First implementation

```
def swap(array, i, j):
    array[i], array[j] = array[j], array[i]

def partition(array, pivot, low, high):
    i, j = low+1, low+1
    while j < high:
        while j < high and array[j] > pivot:
            j += 1
        if j < high:
            swap(array, i, j)
            i += 1
            j += 1
    return i - 1
```

1.2.1. First implementation

1.2.1. First implementation

```
def quicksort(array, low=0, high=-1):
    if high < 0: high = len(array)
    pivot = array[low]  # choose an element
    pivotPos = partition(array, pivot, low, high)  # build partitions
    swap(array, pivotPos, low)  # place the pivot in between
    if pivotPos-low > 1:  # repeat on the left
        quicksort(array, low=low, high=pivotPos)
    if high-pivotPos > 2:  # repeat on the right
        quicksort(array, low=pivotPos+1, high=high)
```

1.2.2. Test data (benchmark)

1.2.2. Test data (benchmark)

```
from random import shuffle, randint

def produceArray(size, repeat):
    array = []
    for _ in range(repeat):
        start = randint(0, size//2)
        array.extend(range(start, start+size))
    shuffle(array)
    return array

arrays = [  # !! Use the same data across different runs
    produceArray(4000, 5)  # arrays of 20000 numbers
    for _ in range(50)  # !! Have several (different) samples
]
```

Benchmark: Checklist

- Enough different samples (generalisation)
- But not too much to speed up development
  - Especially in early stages of the optimisation process
  - Depending on the problem, may not have the choice
- Same data across the different runs

To compare the performances of different versions of your code, the data **must** be the same !
Otherwise, you cannot be certain if a gain/loss of speed is due to a change in the code or a change in the data

1.2.3. Driver program

1.2.3. Driver program

```
nRuns = 10

def main():
    for _ in range(nRuns):  # !! Make several runs
        for array in arrays:
            # copy the data so the original is not altered
            cpArray = [o for o in array]
            # run your code
            quicksort(cpArray)
            # test the result (working code > fast code)
            assert all((cpArray[i] <= cpArray[i+1] for i in range(len(cpArray)-1)))
```

Driver program: Checklist

- Ensure repeatability of the runs
- Do not alter benchmark data
- **Don't break your code !** (Test it)
- Make several runs to reduce the impact of the noise

Driver program: Checklist

- Ensure repeatability of the runs
- Do not alter benchmark data
- **Don't break your code !** (Test it)
- Make several runs to reduce the impact of the noise

Noise ?

- The performance of your code also depends on the environment in which it runs.
- For reasons out of your control, the system may suddenly run slower, hence impacting the performances.

Reduce the possible interferences if you can (shut down other programs, ...)

2. Performance analysis (CPU)

1. Toy problem
2. **Performance analysis (CPU)**
    A. First step into profiling: cProfile
    B. Opening the blackboxes: line_profiler
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. Wrapping up

Profiling

> *In software engineering, **profiling** ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the **frequency and duration of function calls**. Most commonly, **profiling information serves to aid program optimization**, and more specifically, performance engineering.*

(source: Wikipedia))

2.1. First step into profiling: cProfile

1. Toy problem
2. **Performance analysis (CPU)**
    A. **First step into profiling: cProfile**
        a. Profile the code
        b. Examine the stats
        c. Review
    B. Opening the blackboxes: line_profiler
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. Wrapping up

2.1.1. Profile the code

2.1.1. Profile the code

From within a python script :

```
import cProfile

with cProfile.Profile() as pr:
    main()

# Generates a file containing statistics to be examined later :
pr.dump_stats("cProfOut/quicksort.stats")
```

2.1.1. Profile the code

From within a python script :

```
import cProfile

with cProfile.Profile() as pr:
    main()

# Generates a file containing statistics to be examined later :
pr.dump_stats("cProfOut/quicksort.stats")
```

From the terminal :

```
python3 -m cProfile -o <cProfOut/quicksort.stats> <quicksort.py>
```

2.1.2. Examine the stats

2.1.2. Examine the stats

```
import pstats
import pandas as pd

prof = pstats.Stats("cProfOut/quicksort.stats")

#  The following code only serves to present the stats in a dataframe.
kCols = ['file', 'line', 'fn']
vCols = ['cc', 'ncalls', 'tottime', 'cumtime', 'callers']
data = {k: [] for k in vCols + kCols}

for k, v in prof.stats.items():
    for col, val in zip(kCols, k):
        data[col].append(val)

    for col, val in zip(vCols, v):
        data[col].append(val)
# ---------------------------------------------------------------------

df = pd.DataFrame(data)
df = df.sort_values("cumtime", ascending=False)
```

```
df[["ncalls", "tottime", "cumtime", "fn"]][:8]
```

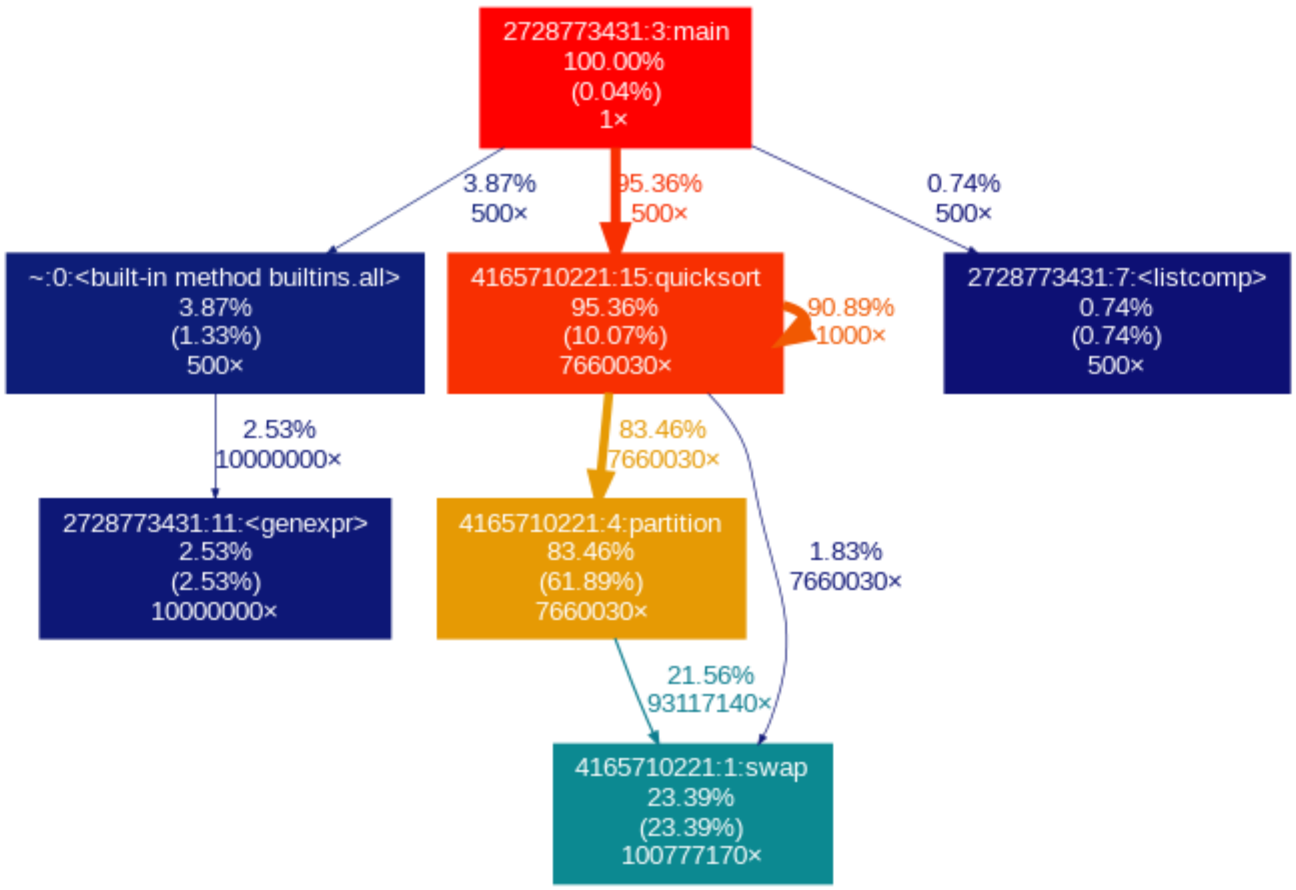
	ncalls	tottime	cumtime	fn
7	1	0.015210	32.772541	main
9	7654920	3.272737	31.136494	quicksort
8	7654920	20.264309	27.262684	partition
6	101220550	7.599361	7.599361	swap
0	500	0.431345	1.253280	<built-in method builtins.all>
3	10000000	0.821936	0.821936	<genexpr>
2	500	0.367472	0.367472	<listcomp>
1	1000	0.000172	0.000172	<built-in method builtins.len>

2.1.2. Examine the stats

2.1.2. Examine the stats

```
# Translates the produced file in a call graph (.dot file)
!gprof2dot -f pstats cProfOut/quicksort.stats -o cProfOut/quicksortCallGraph.dot

# From the .dot file, draw the call graph in the desired format
!dot -Tpng cProfOut/quicksortCallGraph.dot > cProfOut/quicksortCallGraph.png
#dot -Tsvg cProfOut/quicksortCallGraph.dot > cProfOut/quicksortCallGraph.svg
#dot -Tjpg cProfOut/quicksortCallGraph.dot > cProfOut/quicksortCallGraph.jpg
```



2728773431:3:main
100.00%
(0.04%)
1×
3.87%
500×
95.36%
500×
0.74%
500×
~:0:<built-in method builtins.all>
3.87%
(1.33%)
500×
4165710221:15:quicksort
95.36%
(10.07%)
7660030×
90.89%
1000×
2728773431:7:<listcomp>
0.74%
(0.74%)
500×
2.53%
10000000×
83.46%
7660030×
2728773431:11:<genexpr>
2.53%
(2.53%)
10000000×
4165710221:4:partition
83.46%
(61.89%)
7660030×
1.83%
7660030×
21.56%
93117140×
4165710221:1:swap
23.39%
(23.39%)
100777170×

2.1.3. Review

- Built-in python module
- Non-intrusive (from terminal)
- Great at pinpointing bottlenecks
- Very verbose. Needs sorting/filtering to extract useful information

But no in-depth analysis of the code. Better used at high-level to highlight the slowest parts of a large project

2.2. Opening the blackboxes: line_profiler

1. Toy problem
2. **Performance analysis (CPU)**
    A. First step into profiling: cProfile
    B. **Opening the blackboxes: line_profiler**
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. Wrapping up

2.2. Opening the blackboxes: line_profiler

2.2. Opening the blackboxes: line_profiler

```
import line_profiler as lp

pr = lp.LineProfiler()
pr.add_function(partition)
pr.add_function(quicksort)
prMain = pr(main)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/quicksort.lprof")
```

2.2. Opening the blackboxes: line_profiler

```
import line_profiler as lp

pr = lp.LineProfiler()
pr.add_function(partition)
pr.add_function(quicksort)
prMain = pr(main)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/quicksort.lprof")
```

```
prof = lp.load_stats("kernProfOut/quicksort.lprof")

with open("kernProfOut/quicksort.txt", "wt") as f:
    lp.show_text(prof.timings, prof.unit, stream=f)

#lp.show_text(prof.timings, prof.unit)
```

Instead of using `.add_function()`, one can mark functions directly inside the code

```
@lp.profile
def f():
    ...

class C:

    @lp.profile
    def f(self):
        ...
```

```
       Timer unit: 1e-09 s

Total time: 210.368 s
File: /tmp/ipykernel_10358/2728773431.py
Function: main at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def main():
     4        11       4206.0    382.4      0.0      for _ in range(nRuns):  # !! Make several runs
     5       510     255175.0    500.3      0.0          for array in arrays:
     6                                                       # copy the data so the original is not altered
     7       500  781550816.0    2e+06      0.4              cpArray = [o for o in array]
     8                                                       # run your code
     9       500        2e+11    4e+08     98.8              quicksort(cpArray)
    10                                                       # test the result (working code > fast code)
    11       500 1703153954.0    3e+06      0.8              assert all((cpArray[i] <= cpArray[i+1] for i in
range(len(cpArray)-1)))

Total time: 104.947 s
File: /tmp/ipykernel_10358/3731680109.py
Function: partition at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           def partition(array, pivot, low, high):
     5   7654920 1015027267.0    132.6      1.0      i, j = low+1, low+1
     6 102407300        1e+10    118.7     11.6      while j < high:
     7 172146540        3e+10    154.6     25.4          while j < high and array[j] > pivot:
     8  77394160 7669472248.0     99.1      7.3              j += 1
     9  94752380 8664380964.0     91.4      8.3          if j < high:
    10  93565630        3e+10    293.8     26.2              swap(array, i, j)
    11  93565630        1e+10    113.1     10.1              i += 1
    12  93565630 9916953965.0    106.0      9.4              j += 1
    13   7654920  846301657.0    110.6      0.8      return i - 1

Total time: 200.594 s
File: /tmp/ipykernel_10358/3969842243.py
Function: quicksort at line 1
```

3. From analysis to improvement: algorithmically

1. Toy problem
2. Performance analysis (CPU)
3. **From analysis to improvement: algorithmically**
4. Optimising the constants
5. Wrapping up

3. From analysis to improvement: algorithmically

- Write faster code by using fewer instructions
- Remove as much slow operations as possible

WARNING !! The new code *must* be functionally equivalent. Remember the `assert` clause in the `main()` function ? **Test your code**

```python
def betterPartition(array, pivot, low, high):
    i = low+1
    for j in range(low+1, high):
        if array[j] <= pivot:
            swap(array, i, j)
            i += 1
    return i - 1
```

```python
def betterPartition(array, pivot, low, high):
    i = low+1
    for j in range(low+1, high):
        if array[j] <= pivot:
            swap(array, i, j)
            i += 1
    return i - 1
```

- No more comparisons `j < high`
- No more increment `j += 1`
    - (j is incremented in the `for` loop)
- One single loop

```python
def betterQuicksort(array, low=0, high=-1):
    if high < 0:
        high = len(array)
    pivot = array[low]
    pivotPos = betterPartition(array, pivot, low, high)
    swap(array, pivotPos, low)
    if pivotPos-low > 1:
        betterQuicksort(array, low=low, high=pivotPos)
    if high-pivotPos > 2:
        betterQuicksort(array, low=pivotPos+1, high=high)

def betterMain():
    for _ in range(nRuns):
        for array in arrays:
            cpArray = [o for o in array]
            betterQuicksort(cpArray)
            assert all((cpArray[i] <= cpArray[i+1] for i in range(len(cpArray)-1)))
```

```python
pr = lp.LineProfiler()
pr.add_function(betterPartition)
pr.add_function(betterQuicksort)
prMain = pr(betterMain)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/betterQuicksort.lprof")
```

```python
pr = lp.LineProfiler()
pr.add_function(betterPartition)
pr.add_function(betterQuicksort)
prMain = pr(betterMain)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/betterQuicksort.lprof")
```

```python
prof = lp.load_stats("kernProfOut/betterQuicksort.lprof")

with open("kernProfOut/betterQuicksort.txt", "wt") as f:
    lp.show_text(prof.timings, prof.unit, stream=f)

#lp.show_text(prof.timings, prof.unit)
```

```
       Timer unit: 1e-09 s

Total time: 152.431 s
File: /tmp/ipykernel_10358/1631651410.py
Function: betterQuicksort at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def betterQuicksort(array, low=0, high=-1):
     2   7654920  802417745.0    104.8      0.5      if high < 0:
     3       500     180015.0    360.0      0.0          high = len(array)
     4   7654920  771164113.0    100.7      0.5      pivot = array[low]
     5   7654920        1e+11  18963.8     95.2      pivotPos = betterPartition(array, pivot, low, high)
     6   7654920 2413291142.0    315.3      1.6      swap(array, pivotPos, low)
     7   7654920  930344695.0    121.5      0.6      if pivotPos-low > 1:
     8   5501960  964403621.0    175.3      0.6          betterQuicksort(array, low=low, high=pivotPos)
     9   7654920  961126640.0    125.6      0.6      if high-pivotPos > 2:
    10   2152460  422467237.0    196.3      0.3          betterQuicksort(array, low=pivotPos+1, high=high)

Total time: 162.846 s
File: /tmp/ipykernel_10358/1631651410.py
Function: betterMain at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           def betterMain():
    13        11       3354.0    304.9      0.0      for _ in range(nRuns):
    14       510     332520.0    652.0      0.0          for array in arrays:
    15       500  794540552.0    2e+06      0.5              cpArray = [o for o in array]
    16       500        2e+11    3e+08     98.5              betterQuicksort(cpArray)
    17       500 1605240771.0    3e+06      1.0              assert all((cpArray[i] <= cpArray[i+1] for i in 
range(len(cpArray)-1)))

Total time: 78.6656 s
File: /tmp/ipykernel_10358/3108704730.py
Function: betterPartition at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def betterPartition(array, pivot, low, high):
```

4. Optimising the constants

- Compile some heavily used functions
- Reduce the overhead by using low-level instructions instead of actual python code

WARNING !! JIT compiling has a *once only* overhead. Use it only when it is worth it (a function used **a lot** of times in your code)

4.1. Just In Time compilation

1. Toy problem
2. Performance analysis (CPU)
3. From analysis to improvement: algorithmically
4. **Optimising the constants**
    A. **Just In Time compilation**
        a. Introducing Numba
        b. Numpy, types and how they can help
    B. WTH is going on ?
5. Wrapping up

4.1.1. Introducing Numba

4.1.1. Introducing Numba

```
import numba as nb

@nb.jit(nopython=True)
def fasterSwap(array, i, j):
    array[i], array[j] = array[j], array[i]
```

4.1.1. Introducing Numba

```
import numba as nb

@nb.jit(nopython=True)
def fasterSwap(array, i, j):
    array[i], array[j] = array[j], array[i]
```

Numba compiles python functions and runs them on a Low Level Virtual Machine (LLVM).

- `@numba.jit` marks a function to be compiled *Just In Time*
- `nopython=True` means that the function does not use python objects
    - Removes the overhead induced by python
    - But `array` is a python list !

We are dealing with simple int values, using a list is clearly overkill

4.1.2. Numpy, types and how they can help

Numpy is certainly the most widely used python library in research

- It is coded in C (hence compiled).
- It provides multidimensional arrays of **primitive types**
- It is meant for fast numerical manipulation of matrices, tensors, ...
- Used alone or with SciPy, matplotlib, TensorFlow, PyTorch, ...

Numpy arrays are considered as `nopython` by numba, solving our issue with almost no change in the code

```python
def betterFasterPartition(array, pivot, low, high):
    i = low+1
    for j in range(low+1, high):
        if array[j] <= pivot:
            fasterSwap(array, i, j)
            i += 1
    return i - 1

def betterFasterQuicksort(array, low=0, high=-1):
    if high < 0:
        high = len(array)
    pivot = array[low]
    pivotPos = betterFasterPartition(array, pivot, low, high)
    fasterSwap(array, pivotPos, low)
    if pivotPos-low > 1:
        betterFasterQuicksort(array, low=low, high=pivotPos)
    if high-pivotPos > 2:
        betterFasterQuicksort(array, low=pivotPos+1, high=high)
```

```python
import numpy as np

def betterFasterMain():
    for _ in range(nRuns):
        for array in arrays:
            cpArray = np.array(array, dtype=np.int32)
            betterFasterQuicksort(cpArray)
            assert all((cpArray[i] <= cpArray[i+1] for i in range(len(cpArray)-1)))
```

```python
pr = lp.LineProfiler()
pr.add_function(betterFasterPartition)
pr.add_function(betterFasterQuicksort)
prMain = pr(betterFasterMain)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/betterFasterQuicksort.lprof")
```

```python
pr = lp.LineProfiler()
pr.add_function(betterFasterPartition)
pr.add_function(betterFasterQuicksort)
prMain = pr(betterFasterMain)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/betterFasterQuicksort.lprof")
```

```python
prof = lp.load_stats("kernProfOut/betterFasterQuicksort.lprof")

with open("kernProfOut/betterFasterQuicksort.txt", "wt") as f:
    lp.show_text(prof.timings, prof.unit, stream=f)

#lp.show_text(prof.timings, prof.unit)
```

```
       Timer unit: 1e-09 s

Total time: 87.6272 s
File: /tmp/ipykernel_10358/3056985298.py
Function: betterFasterPartition at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def betterFasterPartition(array, pivot, low, high):
     2   7654920  732464990.0     95.7      0.8      i = low+1
     3 178614710        2e+10    103.0     21.0      for j in range(low+1, high):
     4 170959790        3e+10    165.0     32.2          if array[j] <= pivot:
     5  93565630        3e+10    295.6     31.6              fasterSwap(array, i, j)
     6  93565630        1e+10    127.0     13.6              i += 1
     7   7654920  754540079.0     98.6      0.9      return i - 1

Total time: 162.512 s
File: /tmp/ipykernel_10358/3056985298.py
Function: betterFasterQuicksort at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           def betterFasterQuicksort(array, low=0, high=-1):
    10   7654920  849384717.0    111.0      0.5      if high < 0:
    11       500     186064.0    372.1      0.0          high = len(array)
    12   7654920 1045890896.0    136.6      0.6      pivot = array[low]
    13   7654920        2e+11  20235.7     95.3      pivotPos = betterFasterPartition(array, pivot, low, high)
    14   7654920 2344058767.0    306.2      1.4      fasterSwap(array, pivotPos, low)
    15   7654920 1037109548.0    135.5      0.6      if pivotPos-low > 1:
    16   5501960  939187883.0    170.7      0.6          betterFasterQuicksort(array, low=low, high=pivotPos)
    17   7654920  964300891.0    126.0      0.6      if high-pivotPos > 2:
    18   2152460  429405255.0    199.5      0.3          betterFasterQuicksort(array, low=pivotPos+1, high=high)

Total time: 172.848 s
File: /tmp/ipykernel_10358/3146197496.py
Function: betterFasterMain at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def betterFasterMain():
```

4.2. WTH is going on ?

1. Toy problem
2. Performance analysis (CPU)
3. From analysis to improvement: algorithmically
4. **Optimising the constants**
    - A. Just In Time compilation
    - B. **WTH is going on ?**
        - a. IS IT SLOWER !?
        - b. What just happened ?
5. Wrapping up

4.2. WTH is going on ?

- The compiled version version of swap did not speed up the process
- The purpose of compiling swap was to make it faster
- Is it possible that compiled python is slower ?

4.2.1. IS IT SLOWER !?

4.2.1. IS IT SLOWER !?

```
def cmpJIT(array):
    swap(array, 0, 1)
    swap(array, 0, 1)
    swap(array, 0, 1)
    fasterSwap(array, 0, 1)
    fasterSwap(array, 0, 1)
    fasterSwap(array, 0, 1)

pr = lp.LineProfiler()
prCmpJIT = pr(cmpJIT)
prCmpJIT(np.arange(2))
pr.dump_stats("kernProfOut/cmpJIT.lprof")

prof = lp.load_stats("kernProfOut/cmpJIT.lprof")
with open("kernProfOut/cmpJIT.txt", "wt") as f:
    lp.show_text(prof.timings, prof.unit, stream=f)
```

```
        Timer unit: 1e-09 s

Total time: 0.0552353 s
File: /tmp/ipykernel_10358/4007453503.py
Function: cmpJIT at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def cmpJIT(array):
     2         1       6628.0   6628.0      0.0      swap(array, 0, 1)
     3         1        686.0    686.0      0.0      swap(array, 0, 1)
     4         1        522.0    522.0      0.0      swap(array, 0, 1)
     5         1   55223434.0    6e+07    100.0      fasterSwap(array, 0, 1)
     6         1       3472.0   3472.0      0.0      fasterSwap(array, 0, 1)
     7         1        584.0    584.0      0.0      fasterSwap(array, 0, 1)
```

4.2.1. IS IT SLOWER !?

It could. But why ?

> *Under the hood, Python performs a JIT compilation of each line of code. When doing so, python translates directly in binary and does not use a LLVM*

- What is the point, then ?
- When and how is JIT compilation useful ?

```python
@nb.jit(nopython=True)
def betterFasterStrongerPartition(array, pivot, low, high):
    i = low+1
    for j in range(low+1, high):
        if array[j] <= pivot:
            fasterSwap(array, i, j)  # inlining numba swap
            i += 1
    return i - 1

def betterFasterStrongerQuicksort(array, low=0, high=-1):
    if high < 0:
        high = len(array)
    pivot = array[low]
    pivotPos = betterFasterStrongerPartition(array, pivot, low, high)
    fasterSwap(array, pivotPos, low)
    if pivotPos-low > 1:
        betterFasterStrongerQuicksort(array, low=low, high=pivotPos)
    if high-pivotPos > 2:
        betterFasterStrongerQuicksort(array, low=pivotPos+1, high=high)

def betterFasterStrongerMain():
    for _ in range(nRuns):
        for array in arrays:
            cpArray = np.array(array, dtype=np.int32)
            betterFasterStrongerQuicksort(cpArray)
            assert all((cpArray[i] <= cpArray[i+1] for i in range(len(cpArray)-1)))
```

4.2.2. What just happened ?

4.2.2. What just happened ?

```
pr = lp.LineProfiler()
pr.add_function(betterFasterStrongerQuicksort)
prMain = pr(betterFasterStrongerMain)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/betterFasterStrongerQuicksort.lprof")
```

4.2.2. What just happened ?

```
pr = lp.LineProfiler()
pr.add_function(betterFasterStrongerQuicksort)
prMain = pr(betterFasterStrongerMain)
prMain()

# Generates a file containing statistics to be examined later :
pr.dump_stats("kernProfOut/betterFasterStrongerQuicksort.lprof")
```

```
prof = lp.load_stats("kernProfOut/betterFasterStrongerQuicksort.lprof")

with open("kernProfOut/betterFasterStrongerQuicksort.txt", "wt") as f:
    lp.show_text(prof.timings, prof.unit, stream=f)

#lp.show_text(prof.timings, prof.unit)
```

```
       Timer unit: 1e-09 s

Total time: 11.3117 s
File: /tmp/ipykernel_10358/915939174.py
Function: betterFasterStrongerQuicksort at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           def betterFasterStrongerQuicksort(array, low=0, high=-1):
    11   7654920  842797852.0    110.1      7.5      if high < 0:
    12       500     151562.0    303.1      0.0          high = len(array)
    13   7654920 1075857709.0    140.5      9.5      pivot = array[low]
    14   7654920 3631332532.0    474.4     32.1      pivotPos = betterFasterStrongerPartition(array, pivot, low, high)
    15   7654920 2441257032.0    318.9     21.6      fasterSwap(array, pivotPos, low)
    16   7654920 1109806459.0    145.0      9.8      if pivotPos-low > 1:
    17   5501960  814233799.0    148.0      7.2          betterFasterStrongerQuicksort(array, low=low, high=pivotPos)
    18   7654920 1000356872.0    130.7      8.8      if high-pivotPos > 2:
    19   2152460  395929149.0    183.9      3.5          betterFasterStrongerQuicksort(array, low=pivotPos+1, high=high)

Total time: 21.7926 s
File: /tmp/ipykernel_10358/915939174.py
Function: betterFasterStrongerMain at line 21

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    21                                           def betterFasterStrongerMain():
    22        11       3625.0    329.5      0.0      for _ in range(nRuns):
    23       510     205052.0    402.1      0.0          for array in arrays:
    24       500  354574287.0 709148.6      1.6              cpArray = np.array(array, dtype=np.int32)
    25       500         2e+10    4e+07     87.5              betterFasterStrongerQuicksort(cpArray)
    26       500 2358608360.0    5e+06     10.8              assert all((cpArray[i] <= cpArray[i+1] for i in 
range(len(cpArray)-1)))
```

4.2.2. What just happened ?

- What was said before remains true
    - JIT is useful for heavily used functions
- Numba shines when inlining compiled functions (even one-liners)
- It was useful to compile `swap` but only to use it in other compiled functions
- In doubt, always profile the JIT version of your code VS the original one

5. Wrapping up

1. Toy problem
2. Performance analysis (CPU)
3. From analysis to improvement: algorithmically
4. Optimising the constants
5. **Wrapping up**
    A. Family picture
    B. Do not reinvent the wheel

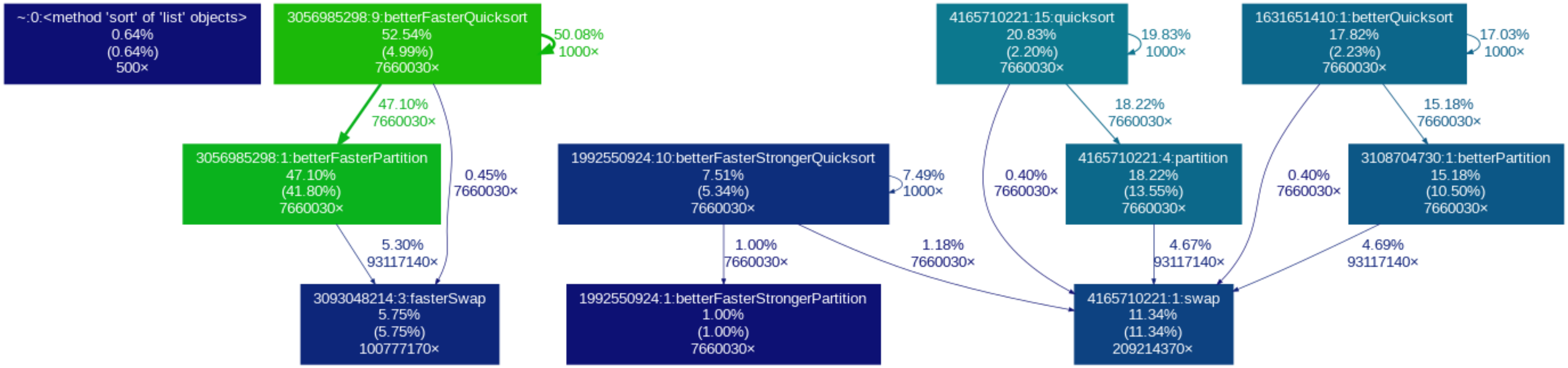
```python
with cProfile.Profile() as pr:
    for _ in range(nRuns):
        for array in arrays:
            cpArray = [o for o in array]
            quicksort(cpArray)
            cpArray = [o for o in array]
            betterQuicksort(cpArray)
            cpArray = np.array(array, dtype=np.int32)
            betterFasterQuicksort(cpArray)
            cpArray = np.array(array, dtype=np.int32)
            betterFasterStrongerQuicksort(cpArray)
            cpArray = [o for o in array]
            cpArray.sort()

# Generates a file containing statistics to be examined later :
pr.dump_stats("cProfOut/quicksortAll.stats")
```

5.1. Family picture

```
# Translates the produced file in a call graph (.dot file)
!gprof2dot -f pstats cProfOut/quicksortAll.stats -o cProfOut/quicksortAllCallGraph
!dot -Tpng cProfOut/quicksortAllCallGraph.dot > cProfOut/quicksortAllCallGraph.png
```

```
# Translates the produced file in a call graph (.dot file)
!gprof2dot -f pstats cProfOut/quicksortAll.stats -o cProfOut/quicksortAllCallGraph
!dot -Tpng cProfOut/quicksortAllCallGraph.dot > cProfOut/quicksortAllCallGraph.png
```



Note: The compiled code is not shown on the picture

5.2. Do not reinvent the wheel

The sort method of a list is *WAAAAYY* faster than anything presented during this talk

- Always ask yourself if what you are doing exists already
- In other words, make a "state of the art" of the existing solutions
- **Worse case scenario:** you lost 2 hours looking for some library that does not exist
- **Best case scenario:** you spent days learning to use a library but you saved weeks of coding/profiling