

# Profiling and Benchmarking

## Software Engineering for Scientists

### CU Boulder, Fall 2019

[https://github.com/swe4s/lectures/tree/master/benchmark\\_profile](https://github.com/swe4s/lectures/tree/master/benchmark_profile)

Efficiency is often a major concern when developing scientific software, and improving how long a program takes to run or how much memory it uses requires a detailed understanding of how the program works and the ability to compare the efficiency of two programs (or different versions of the same program). **Profiling** is the process of measuring how much time is spent in each section of a program during a particular execution. **Benchmarking** is the process of comparing the resources used by two different programs on the same input.

## Profiling

A major challenge in program optimization is knowing where code can (and should) be improved. Before any time is spent improving a particular region of your program, you need to determine the proportion of time that was consumed by that code. This proportion will determine the maximum benefit that any optimization could provide. For example, if the total runtime for a program was 1 minute and a particular function ran for a total of 5 seconds, then the best possible optimization of that function (where the runtime of the function goes to zero) would produce a program that runs 1.09X faster. Understanding the balance between costs and benefits of different optimizations will help you identify and prioritize future improvements.

Python provides the dynamic analysis module **cProfile** for measuring the time spent running different functions. To profile a particular script, **simply load the cProfile module in the execution command** and the profile will be printed to standard out. You can control the order of the profile with the `-s` flag.

```
$ python -m cProfile -s tottime search.py 10000 1000 L
60244 function calls (60201 primitive calls) in 0.289 seconds

Ordered by: internal time
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000	0.261	0.000	0.261	0.000	search.py:4(linear_search)
11000	0.007	0.000	0.013	0.000	random.py:173(randrange)
6	0.006	0.001	0.006	0.001	{built-in method _imp.create_dynamic}
11000	0.004	0.000	0.006	0.000	random.py:223(_randbelow)
11000	0.003	0.000	0.016	0.000	random.py:217(randint)
1	0.002	0.002	0.016	0.016	search.py:29(<listcomp>)
11000	0.001	0.000	0.001	0.000	{method 'bit_length' of 'int' objects}
1	0.001	0.001	0.289	0.289	search.py:1(<module>)

The output summarises the total amount of work done in function calls and time, plus six columns for each function call:

**ncalls**: the number of calls,

**tottime**: the total time spent in the given function (and excluding time made in calls to sub-functions)

**percall**: the quotient of tottime divided by ncalls

**cumtime**: the cumulative time spent in this and all subfunctions (from invocation till exit).

**percall**: is the quotient of cumtime divided by primitive calls

**filename:lineno(function)**: provides the respective data of each function

In the table above, the function call list is sorted by total time (`-s tottime`) and the functions that are higher in the list took more time and are good candidates for optimization.

It is important to understand that cProfile measures the performance of function calls. Since the time for all of the code in a function is summarized together, cProfile cannot identify subsets of functions that are slow.

## Benchmarking

The most basic way to test the resources used by a program is to use the GNU time program. This is not to be confused with `time` that is part of bash. While the bash `time` is useful for measuring how long a program ran, GNU `time` can also measure memory consumption.

It is very likely that your system has bash `time`, and it may also have GNU `time`. To tell the difference run `time -V` to get the version. If there is an error, then it is the bash version

### GNU time

```
(swe4s) $ time -V
time (GNU Time) 1.8
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by David Keppel, David MacKenzie, and Assaf Gordo
```

### bash time

```
(swe4s) $ time -V
bash: -V: command not found
```

Even if the `time -V` gave an error, GNU `time` may still be on your system. Use the `which` command to check.

```
(swe4s) $ time -V
bash: -V: command not found

real    0m0.002s
user    0m0.001s
sys     0m0.001s
(swe4s) $ which time
/home/jovyan/.conda/envs/swe4s/bin/time
(swe4s) $ /home/jovyan/.conda/envs/swe4s/bin/time -V
time (GNU Time) 1.8
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by David Keppel, David MacKenzie, and Assaf Gordon.
```

### To install GNU time

```
(swe4s) $ conda install -c conda-forge time
```

The output of GNU `time` is highly configurable using the `-f` flag along with an output pattern. To get the total runtime and memory usage use `-f '%e\t%M'`. `%e` gives total elapsed time in seconds and `%M` gives the maximum memory usage in kilobytes.

grow.py

```
import sys
N = int(sys.argv[1])
A = []
for i in range(N):
    A.append(i)
```

```
(swe4s) $ time -f '%e\t%M' python grow.py
1000000
0.02 7200
```

When benchmarking a program, it is important to get the memory and time over a range of input sizes. These results will help you understand how your program will perform when considering larger problems.

```
(swe4s) $ for i in `seq 1 200000 1000000`; do time -f '%e\t%M' python grow.py $i; done
0.03 7080
0.05 24440
0.07 37784
0.10 45692
0.12 53628
```

GNU `time` can only measure full program execution. To get a more detailed assessment of how long individual sections of code run, you must instrument your code with timers. Python provides the `time` module that makes it easy to store the time before the target block with `t0 = time.time()` and after the block with `t1 = time.time()`. The difference between `t1` and `t0` gives the elapsed time in seconds.

Suppose we are considering a program that uses binary search, which is faster than linear search but also requires the data to be sorted. To measure how much of the program is spent sorting and how much is spent searching we check the time before and after the two sections and compare the differences.

```
t0_sort = time.time()
D.sort()
t1_sort = time.time()

t0_search = time.time()
for q in Q:
    binary_search(q, D)
t1_search = time.time()

total_time = t1_search - t0_sort

print(total_time, (t1_sort-t0_sort)/(total_time),
      (t1_search-t0_search)/(total_time))
```

```
(swe4s) $ python search.py 10000 1000 B
0.0038118362426757812 0.4446459844883663 0.5550412809607206
```

The total time is 0.004 seconds with 44.5% of that time consumed by sort and 55.5% by search.