Lab 06: Timing your Python Programs

CMPT 145

Laboratory 06 Overview

Section 1

Pre-Lab Reading

Timing Python Programs

- We are interested to determine if a Python program is efficient.
- One thing we can do is run the program and see how much time it requires to finish its work.
- This simple idea has some very subtle problems.
- Let's explore them!

Timing Python Programs: Simple!

- The basic approach is to measure elapsed time:
  1. Look at the clock before the program starts.
  2. Run the program.
  3. Look at the clock after the program ends.
  4. Subtract to get the elapsed time.
- That was easy. What could go wrong?

Problems with Time: Your computer

- A computer has several clocks.
- Different CPU models will have different clocks
- Different purposes, and different precisions.
  - Short programs might start and finish before some clock ticks even once.
  - Such a program may show zero elapsed time!
- The operating system dictates which clocks an application has access to.
  - In particular, the Windows default clock is very low precision!
- Conclusion: we cannot directly compare times reported on different systems.

Problems with Time: Your System

- Your computer is multi-tasking.
  - All running applications are sharing the hardware resources, e.g, CPU, disk, network, graphics card
  - The operating system may pause your program to let other applications run.
  - Other applications can delay your program by using resources you need too!
  - Usually these delays are too small for humans to notice, but not always.
- Conclusion: Using elapsed time to measure a program's performance includes all these delays, and will report times that are too large to be accurate.

Problems with Time: Your Program

- Programs written in introductory courses are likely to run very fast, because they are very small.
- Console I/O is never an interesting part of the run time.
- Conclusion: We measure time only for interesting parts of the program that run for a "long time."

Problems with Time: Python

- Python is an application called an interpreter.
  - It reads your program and executes it line-by-line.
- Python provides tools that make programmer time more productive.
- These tools often cause the computer to do work for you that you not be aware of.
- Timing a Python script also includes the time taken on those things!
- Conclusion: To time a Python script, we have to accept these extra time costs; but the same algorithm in a different language could be significantly faster!

Timing Python Programs: Example

```python
def run_sort(n):
    """
    Purpose:
        Time the execution of Python's sort() operation
    :param n: The size of the list to sort
    :return: A measure of the time taken.
    """
    numbers = []
    for i in range(n):
        numbers.append(rand.randint(0,n))

    start = get_time()
    numbers.sort()
    end = get_time()

    return end - start
```

Notes on the example

- The function `get_time()` will be discussed shortly.
- The function `run_sort()` allows us to collect timing data for a range of sizes.
- The random list makes sure the experiment is objective.
- A separate timing script can collect statistics on several runs.

The Python Module: `time`

`time.time()`

- Returns the current time according to one of the computer's clocks.
    - Measured in seconds (no fractions).
    - Relative to a fixed standard date in the past.
    - Low resolution!
- Useful for things like:
    - Recording the date and time that a file was created or modified.
    - Calendar applications.

The Python Module: `time`

`time.process_time()`

- Uses the system's highest precision clock.
  - Depends on the computer's hardware and operating system.
  - Returns a fractional measure of time in seconds.
- This clock is not elapsed time!
  - This clock measures how much time your program has been running.
  - Excludes any delays caused by your system running other applications.

A timing script

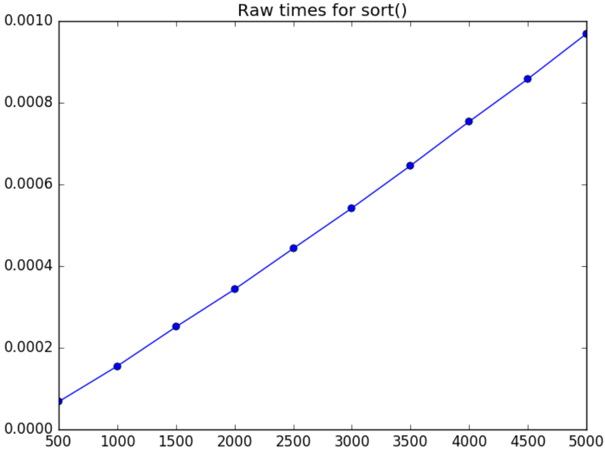
```python
import Statistics as Statistics

trials = 50

print("Statistics for Python's sort()")
sort_sizes = range(500,5001,500)
sort_times = []
for size in sort_sizes:
    t1 = Statistics.create()
    for i in range(trials):
        Statistics.add(t1, run_sort(size))
    rtime = Statistics.minimum(t1)
    print(size, rtime)
    sort_times.append(rtime)
```

Notes on the timing script

- The function `run_sort()` is called for several sizes.
  - Larger lists avoid problems arising from low-precision clocks.
  - If the list sizes are too small, the times reported are unreliable. See Slide 6.
  - If the list sizes are too big, the script takes too long to run!
- We take the minimum time seen over 50 trials as being representative.
- The representative time is displayed, and stored in a list, which we can plot.

Using the minimum time

- In science, all measurements have errors.
- The normal assumption is that errors can result in measurements that are too large or too small.
  - E.g., a human experimenter might stop a stop-watch too early or too late.
- The normal way to address these errors is to take an average of several measurements.
- In a computer, there are obvious ways for an error to result in a time measurement that is too large, e.g., delays caused by multi-tasking.
- But if we are careful, there are very few ways for a measurement to be too small.

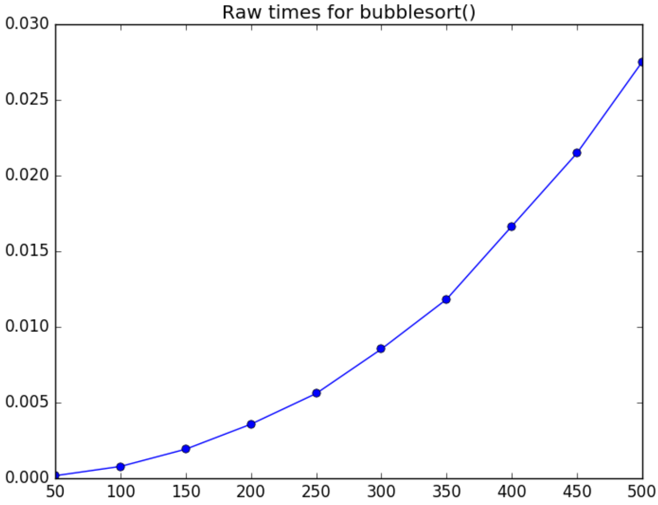

Raw times for sort()
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

Notes on the raw time plot for `sort()`

- The data shows that the time required to sort increases with the size of the list.
- The increase is not linear! There is definitely a curve upwards. The slope is increasing.
- You might well wonder if the result is normal or expected.

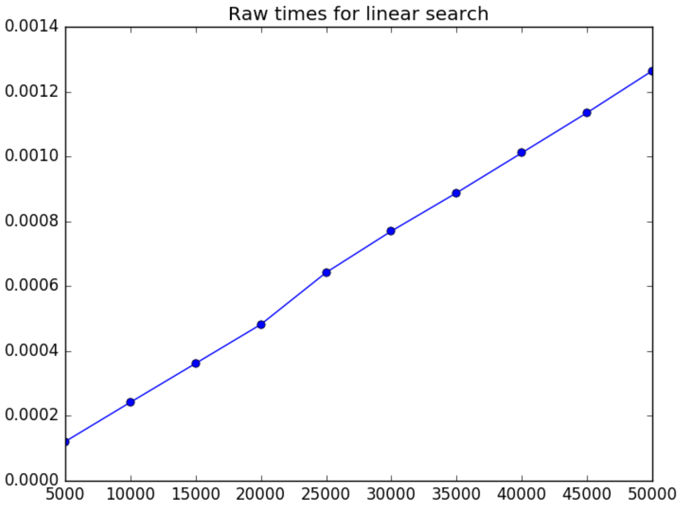
Two other examples

- We'll show results of timing experiments done on two other examples:
    1. BubbleSort also sorts lists. We can compare the two sorting methods to each other. One will be better.
    2. Linear search through an unsorted list. Including this example provides context. Is searching easier than sorting? If so, by how much?


Raw times for bubblesort()
0.030
0.025
0.020
0.015
0.010
0.005
0.000
50
100
150
200
250
300
350
400
450
500

Notes on the raw time plot for `bubblesort()`

- The data shows that the time required to sort increases with the size of the list.
- There is definitely a curve upwards. The slope is increasing faster than for `sort()`
- You might well wonder if the difference between `sort()` and `bubblesort()` is significant.


Raw times for linear search
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

Notes on the raw time plot for linear search

- The data shows that the time required to sort increases with the size of the list.
- The data points show a trend with some variance.
- You might well wonder if the trend is a line, or if there is a curve.

In case you wondered about the graphs

- We can gather data like this for any function, program, or algorithm.
- The shape of the curve is what interests us most.
- We could guess the shape of the curves: linear, quadratic, etc.
- The shape of the curve can be predicted, by algorithm analysis (Chapter 16) of the program code!
- The shape can be verified empirically too.

The Principle of Doubling

- We already showed how to collect raw timing data.
- For doubling, we run two experiments per trial:
  - Once for list of size n. It will take t_n seconds.
  - Once for list of size $2n$. It will take t_{2n} seconds.
- We'll use the ratio $\frac{t_{2n}}{t_n}$ as data points, varying n.
- If the raw time plot is:
  - Linear, we'll get ratios roughly $\frac{t_{2n}}{t_n} = 2$. Why?
  - A quadratic curve, we'll get ratios roughly $\frac{t_{2n}}{t_n} = 4$

Section 2

Laboratory Activities

Timing Python Programs

Download the files from Moodle:

- `times.py`
  An experiment, see Slide 28.
- `ratios.py`
  An experiment, see Slide 30.
- `apparatus.py`
  Functions used by the experiments.
- `Statistics.py`
  A familiar ADT used by the experiments.
- `bbs.py`
  An implementation of BubbleSort.

Timing Python Programs

ACTIVITY: Run `times.py` to check that everything is working properly.

- A window with 3 graphs should pop up eventually. Check behind the PyCharm window!
- The three graphs should be similar to the ones earlier in this set of slides.

Note: If you are seeing flat lines at zero, see Slide 33; change your timer to `time.perf_counter()`.

Managing low precision clocks

ACTIVITY: Open `apparatus.py`, and find the `get_time()` function:

```
def get_time():
    # return time.time()                # wall clock time
    # return time.perf_counter()        # CPU clock system-wide
    return time.process_time()        # CPU clock this process
```

If you saw flat lines at zero for the previous experiments, edit the code so that `get_time()` uses `time.perf_counter()`.

- Rerun both scripts: `times()` and `ratios()`.
- Keep the timer that gives you non-zero data to look at.

Timing Python Programs

ACTIVITY: Run `ratios.py` to check that everything is working properly.

- A window with 3 graphs should pop up eventually. Check behind the PyCharm window!
- The three graphs should show lines that are roughly flat, with some variance.
- The ratios for `sort()` should be just higher than 2.
- The ratios for `bubblesort()` should be around 4.
- The ratios for `linear search` should be around 2.

Observations from the data

- `bubblesort()`
  - The raw timing (`times.py`) curves upward noticeably.
  - The ratios (`ratio.py`) should be around 4.
- `sort()`
  - The raw timing (`times.py`) curves upward slightly.
  - The ratios (`ratio.py`) should be just higher than 2.
- `linear search`
  - The raw timing (`times.py`) has no obvious curve.
  - The ratios (`ratio.py`) should be around 2.

Drawing conclusions from the data

- The two sorting algorithms are quite different!
- The linear search and `sort()` are quite different, too.
- Linear search is definitely linear.
- BubbleSort is definitely quadratic.
- The timing curve (raw times) for `sort()` is between linear and quadratic.

Using different clocks

ACTIVITY: Open `apparatus.py`, and find the `get_time()` function:

```
def get_time():
    return time.time()                # wall clock time
    # return time.perf_counter()        # CPU clock system-wide
    # return time.process_time()        # CPU clock this process
```

- Edit the code so that `get_time()` uses `time.time()`.
- Rerun both scripts: `times()` and `ratios()`.
- What you see will depend on your computer, your operating system, and your version of Python.
- Describe the differences between what you see here, and what you saw earlier.
- Put this comparison in your hand-in file, e.g.

```
Activity: Using different clocks
When I used time.time() I observed that ...
```

Using the average time

ACTIVITY: Open `times.py`

- Edit the code so that the data collection for all three tasks uses `Statistics.mean()` instead of `Statistics.minimum()`
- Rerun the script.
- Compare the graphs to what you saw previously.
- Add a brief comment in your hand-in file about what you observed, e.g.

```
Activity: Using average instead of minimum
When I used mean() I observed that ...
```

- Change it back to what it was!

Changing the experiment

ACTIVITY: Open `times.py`

- Find where `bubblesort_sizes` is created.
- The values in `sort_sizes` are 10 times bigger.
- To understand why, make `bubblesort_sizes` the same as `sort_sizes`.
- Rerun the script.
- Compare the graphs to what you saw previously.
- Add a brief comment in your hand-in file about what you observed.

  ```
  Activity: Changing the experiment
  When I used larger lists for bubblesort, I observed...
  ```

- Change the list sizes back to what they were!

Changing the experiment

ACTIVITY: Open `ratios.py`

- Edit the code so that all three experiments use much smaller list sizes.
- Rerun the script.
- Compare the graphs to what you saw previously.
- Add a brief comment in your hand-in file about what you observed.

```
Activity: Changing the experiment
When I used small lists I observed that ...
```

- Change the list sizes back to what they were!

Section 3

Hand In

What To Hand In

Hand in your `lab06-responses.txt` file showing:

- What you observed when you used `time.time()`
- What you observed when you used `mean()`
- What you observed when you increased the size of the lists for BubbleSort
- What you observed when you used smaller lists for all tasks