TIMING PROGRAMS, COUNTING OPERATIONS

(download slides and .py files to follow along)

6.100L Lecture 21

Ana Bell

WRITING EFFICIENT PROGRAMS

- So far, we have **emphasized correctness**. It is the first thing to worry about! But sometimes that is not enough.
- Problems can be very complex
- But data sets can be very large: in 2014 Google served 30,000,000,000,000 pages covering 100,000,000 GB of data

EFFICIENCY IS IMPORTANT

- Separate **time and space efficiency** of a program
- Tradeoff between them: can use up a bit more memory to store values for quicker lookup later
  - Think Fibonacci recursive vs. Fibonacci with memoization
- Challenges in understanding efficiency
  - A program can be **implemented in many different ways**
  - You can solve a problem using only a handful of different **algorithms**
- Want to separate choice of implementation from choice of more abstract algorithm

EVALUATING PROGRAMS

- Measure with a **timer**
- **Count** the operations
- Abstract notion of **order of growth**

ASIDE on MODULES

- A module is a set of python definitions in a file
  - Python provides many useful modules: math, plotting/graphing, random sampling for probability, statistical tools, many others
- You first need to “import” the module into your environment

```
import time
import random
import dateutil
import math
```

- Call functions from inside the module using the module’s name and dot notation

```
math.sin(math.pi/2)
```

TIMING

TIMING A PROGRAM

- Use time module
- Recall that importing means to bring in that class into your own file

- **Start** clock
- **Call** function
- **Stop** clock

```
import time

def c_to_f(c):
    return c*9.0/5 + 32

tstart = time.time()
c_to_f(37)
dt = time.time() - tstart
print(dt, "s,")
```

Seconds since the epoch: Jan 1, 1970

TIMNG `c_to_f`

- Very fast, can't even time it accurately

```
c_to_f(1) took 0.0 seconds
c_to_f(10) took 0.0 seconds
c_to_f(100) took 0.0 seconds
c_to_f(1000) took 0.0 seconds
c_to_f(10000) took 0.0 seconds
c_to_f(100000) took 0.0 seconds
c_to_f(1000000) took 0.0 seconds
c_to_f(10000000) took 0.0 seconds
```

TIMING `mysum`

- As the **input increases**, the time it takes also increases
- Pattern?
  - 0.009 to 0.05 to 0.5 to 5 to ??

```
mysum(1) took 0.0 sec
mysum(10) took 0.0 sec
mysum(100) took 0.0 sec
mysum(1000) took 0.0 sec
mysum(10000) took 0.0019927024841308594 sec
mysum(100000) took 0.009970903396606445 sec
mysum(1000000) took 0.0508952140808105 sec
mysum(10000000) took 0.4966745376586914 sec
mysum(100000000) took 5.688449382781982 sec
```

TIMING `square`

- As the **input increases** the time it takes also increases
- `square` called with 100000 did not finish within a reasonable amount of time
- Maybe we can guess a pattern if we are patient for one more round?

```
square(1) took 0.0 sec
square(10) took 0.0 sec
square(100) took 0.0 sec
square(1000) took 0.06244492530822754 sec
square(10000) took 5.553335428237915 sec
```

TIMING PROGRAMS IS INCONSISTENT

- GOAL: to evaluate different algorithms
- Running time **should vary between algorithms**
- Running time **should not vary between implementations**
- Running time **should not vary between computers**
- Running time **should not vary between languages**
- Running time is **should be predictable** for small inputs

- Time varies for different inputs but cannot really express a relationship between inputs and time needed
- Can only be measured *a posteriori*

COUNTING

COUNTING OPERATIONS

- Assume these steps take **constant time**:
    - Mathematical operations
    - Comparisons
    - Assignments
    - Accessing objects in memory
- Count number of operations executed as function of size of input

c_to_f → 3 ops

```
def c_to_f(c):
    return c*9.0/5 + 32
```

3 ops

mysum → 1+(x+1)*(1+2) = 3x+4 ops

```
def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

1 op

loop x+1 times

1 op

2 ops

square → 1+n*(1)*n*(1+2) = $3n^2$ + 1 ops

```
def square(n):
    sqsum = 0
    for i in range(n):
        for j in range(n):
            sqsum += 1
    return sqsum
```

1 op

1 op

1 op

loop n times

loop n times

2 ops

COUNTING `c_to_f`

- No matter what the input is, the number of **operations is the same**

```
c_to_f(100): 3 ops, 1.0 x more
c_to_f(1000): 3 ops, 1.0 x more
c_to_f(10000): 3 ops, 1.0 x more
c_to_f(100000): 3 ops, 1.0 x more
c_to_f(1000000): 3 ops, 1.0 x more
c_to_f(10000000): 3 ops, 1.0 x more
```

COUNTING `mysum`

- As the input increases **by 10**, the number if operations ran is approx. **10 times more**.

```
mysum(100): 304 ops, 1.0 x more
mysum(1000): 3004 ops, 9.88158 x more
mysum(10000): 30004 ops, 9.98802 x more
mysum(100000): 300004 ops, 9.9988 x more
mysum(1000000): 3000004 ops, 9.99988 x more
mysum(10000000): 30000004 ops, 9.99999 x more
```

COUNTING `square`

- As the input increases **by 10**, the number of operations is approx. **100 times more**.

```
square(1): 5 ops, 1.0 x more
square(10): 311 ops, 62.2 x more
square(100): 30101 ops, 96.78778 x more
square(1000): 3001001 ops, 99.69772 x more
square(10000): 300010001 ops, 99.96998 x more
```

- As the input increases **by 2**, the number of operations is approx. **4 times more**.

```
square(128): 49281 ops, 1.0 x more
square(256): 196865 ops, 3.99474 x more
square(512): 786945 ops, 3.99738 x more
square(1024): 3146753 ops, 3.99869 x more
square(2048): 12584961 ops, 3.99935 x more
square(4096): 50335745 ops, 3.99967 x more
square(8192): 201334785 ops, 3.99984 x more
```

COUNTING OPERATIONS IS INDEPENDENT OF COMPUTER VARIATIONS, BUT …

- GOAL: to evaluate different algorithms
- Running "time" **should vary between algorithms**
- Running "time" **should not vary between implementations**
- Running "time" **should not vary between computers**
- Running "time" **should not vary between languages**
- Running "time" is **should be predictable** for small inputs
- No real definition of **which operations** to count

- Count varies for different inputs and can derive a relationship between inputs and the count

… STILL NEED A BETTER WAY

- Timing and counting **evaluate implementations**
- Timing and counting **evaluate machines**

- Want to **evaluate algorithm**
- Want to **evaluate scalability**
- Want to **evaluate in terms of input size**

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022