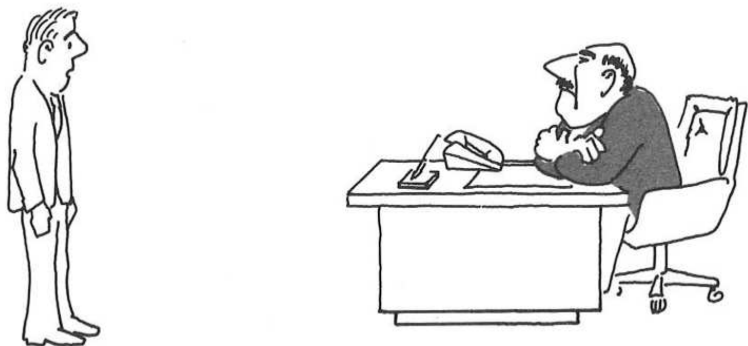
Outline

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

MCS 260 Lecture 29
Introduction to Computer Science
Jan Verschelde, 18 March 2016

imagine a meeting with your boss ...



"I can't find an efficient algorithm, I guess I'm just too dumb."

From *Computers and intractability. A Guide to the Theory of NP-Completeness* by Michael R. Garey and David S. Johnson, Bell Laboratories, 1979.

what you want to say is

"I can't find an efficient algorithm, because no such algorithm is possible!"

From *Computers and intractability. A Guide to the Theory of NP-Completeness* by Michael R. Garey and David S. Johnson, Bell Laboratories, 1979.

you better have some backup

"I can't find an efficient algorithm, but neither can all these famous people."

From *Computers and intractability. A Guide to the Theory of NP-Completeness* by Michael R. Garey and David S. Johnson, Bell Laboratories, 1979.

Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.

Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

- **space** for intermediate and final results;
- **time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

complexity and cost
timing Python code

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

The big-oh Notation

to measure complexity

Let n be the dimension of the problem,
e.g.: n is number of elements to add, sort, etc...

A function $f(n)$ is $O(g(n))$ (we say: *f is of order g*)
if there exists a positive constant c (*independent of n*): $f(n) \leq cg(n)$,
for sufficiently large n.

Big-oh defines the order of complexity, some examples:

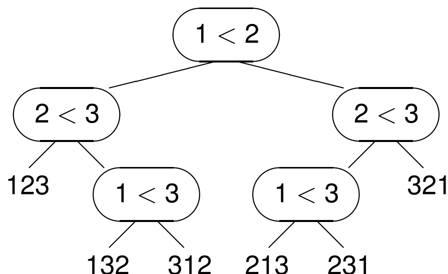
- f is $O(\log(n))$: logarithmic in n;
- f is $O(n)$: linear in n;
- f is $O(n^2)$: quadratic in n;
- f is $O(2^n)$: exponential in n.

Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort n numbers?

#permutations equals $n! = n \cdot (n-1) \cdots 2 \cdot 1$.
A sort computes a permutation to order the list.



$S(n)$ = minimal #comparisons. From the tree: $n! \leq 2^{S(n)}$.
Stirling: $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n\log(n))$.
A lower bound on sorting complexity: $O(n\log(n))$.

complexity and cost
timing Python code

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

Complexity Classes

We distinguish three big classes of complexity:

P polynomial time

The problem can be solved in $O(f(n))$, where $f(n)$ is a polynomial in n.
Example: evaluate a polynomial.

NP nondeterministic polynomial time

A solution to the problem can be verified in polynomial time.
Example: root finding.

#P counting problems

How many solutions does a problem have?
Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win $1,000,000: is P = NP?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* undecidable!

complexity and cost
timing Python code

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

Counting Flops

floating-point operations

A flop is short for floating-point operation.

In scientific computation, the cost analysis is often measured in flops.

An application of Object Oriented Programming:

1. An object `FlopFloat` stores a `float` and `flops`.
2. Value of `flops` = cost of a number as object data attribute.
3. Overloading arithmetical operators we count the flops.

Recall the lecture on operator overloading.

We use `FlopFloats` to count the flops to evaluate a polynomial of degree d with random coefficients.

evaluation of polynomial with `floppoly.py`

```
from random import gauss
from flopfloats import FlopFloat

print('#flops to evaluate a polynomial')
DEGRAW = input('Give degree : ')
DEG = int(DEGRAW)
X = FlopFloat(gauss(0, 1))
RESULT = FlopFloat()
for i in range(0, DEG+1):
    term = FlopFloat(gauss(0, 1))
    for j in range(0, i):
        term = term*X
    RESULT = RESULT + term
print('evaluating a polynomial of degree '+ \
    '%d \ntakes %d flops' % (DEG, RESULT.flops))
```

running `floppoly`

Evaluating polynomials with random coefficients:

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 10
evaluating a polynomial of degree 10
takes 66 flops

$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 20
evaluating a polynomial of degree 20
takes 231 flops
```

The #flops is not linear in the degree.

complexity and cost
timing Python code

1 Complexity and Cost
- measuring complexity: big-oh
- complexity classes
- counting flops: floating-point operations

2 Cost of Algorithms
- timing Python programs
- the unix command time
- try-except costs more than if-else
- importing modules or importing functions?
- the cost of working with files
- the efficiency of list comprehensions

3 Summary + Assignments

Performance Analysis

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`, ex: sort is $O(n \log(n))$?
2. use module `time`, ex: cost of exception handling
3. use `timeit`, ex: importing module or functions
4. use `os.times()`, ex: cost of handling files
5. profiling code, ex: are list comprehensions efficient?

Pushing a program to its limits is a *stress test*.

complexity and cost
timing Python code

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

Timing Programs: time on unix like systems

The Unix `time` command runs as

```
$ time < program name >
```

and returns three times: real, user, and system time.
The real time is the wall time. User time is also called CPU time, and system time measures the overhead.

```
$ time python sortrandnumb.py 100000

real    0m1.639s
user    0m1.530s
sys     0m0.030s
```

The script `sortrandnumb.py` sorts *n* random numbers.
The value for *n* is retrieved from the command line.

the script sortrandnumb.py

```
"""
This program illustrates how to time the
sort of lists.  The optional command line
argument is the length of the list to sort.
"""
import sys
from random import gauss
# count number of arguments of command line
if len(sys.argv) < 2:
    N = int(input('Give #elements : '))
else:
    N = int(sys.argv[1]) # get argument
RNG = list(range(0, N))
NUM = [gauss(0, 1) for i in RNG]
NUM.sort()
```

Edited Output

running on a not so new PowerBook G4

```
$ time python sortrandnumb.py 100000
real 0m1.639s  user 0m1.530s  sys  0m0.030s

$ time python sortrandnumb.py 200000
real 0m3.264s  user 0m3.130s  sys  0m0.060s

$ time python sortrandnumb.py 400000
real 0m6.591s  user 0m6.310s  sys  0m0.140s

$ time python sortrandnumb.py 800000
real 0m13.689s  user 0m13.080s  sys 0m0.250s

$ time python sortrandnumb.py 1600000
real 0m30.954s  user 0m27.360s  sys 0m0.540s

$ time python sortrandnumb.py 3200000
real 1m0.447s  user 0m56.980s  sys 0m0.980s
```

Timing Python Code: using `time.clock`

```
import time
starttime = time.clock()
< code to be timed >
stoptime = time.clock()
elapsed = stoptime - starttime
print 'elapsed time: %.2f seconds' % elapsed
```

Replace `< code to be timed >` by a function call.

timing the execution of `sort`

```
import time
import sys
from random import gauss

if len(sys.argv) < 2:
    N = int(input('Give #elements : '))
else:
    N = int(sys.argv[1])
NUM = [gauss(0, 1) for i in range(N)]
START_TIME = time.clock()
NUM.sort()
STOP_TIME = time.clock()
ELAPSED = STOP_TIME - START_TIME
print('sorting %d floats' % N \
    + ' took %.3f seconds ' % ELAPSED)
```

complexity and cost
timing Python code

1 Complexity and Cost
- measuring complexity: big-oh
- complexity classes
- counting flops: floating-point operations

2 Cost of Algorithms
- timing Python programs
- the unix command time
- try-except costs more than if-else
- importing modules or importing functions?
- the cost of working with files
- the efficiency of list comprehensions

3 Summary + Assignments

cost of exception handlers: a first case study

Exception handlers cost more than a regular conditions.

Rule: reserve exceptions for exceptional situations.

Let us try to test this rule on a simple case.

Computational experiment: repeated division.

```
for _ in range(nbr):
    num = randint(-1, 1)
    den = randint(0, 1)
    result = num/den
```

We have a 50% chance that `den` is zero.

handling division by zero: if-else or try-except

Executing the division `result = num/den`.
Count the number of times `den` is zero.
Using `if-else` statement:

```
if den != 0:
    result = num/den
else:
    count = count + 1
```

Using `try-except` statement:

```
try:
    result = num/den
except ZeroDivisionError:
    count = count + 1
```

Claim: `try-except` costs more.

ifelse: handle division by zero by if-else

```
def ifelse(nbr):
    """
    Does nbr divisions of p by q,
    checks first if q is zero.
    """
    count = 0
    for _ in range(nbr):
        num = randint(-1, 1)
        den = randint(0, 1)
        if den != 0:
            result = num/den
        else:
            count = count + 1
    print('#divisions by zero :', count)
```

tryexcept: handle division by zero by try-except

```
def tryexcept(nbr):
    """
    Does nbr divisions of p by q,
    treats zero q as exception.
    """
    count = 0
    for _ in range(nbr):
        num = randint(-1, 1)
        den = randint(0, 1)
        try:
            result = num/den
        except ZeroDivisionError:
            count = count + 1
    print('#divisions by zero :', count)
```

the main program

```
def main():
    """
    Compares the efficiency of if-else with
    the try-except to avoid division by zero.
    """
    import time
    nbr = 1000000
    seed(2)
    starttime = time.clock()
    ifelse(nbr)
    stoptime = time.clock()
    elapsed = stoptime - starttime
    print('if-else: %.2f seconds' % elapsed)
    seed(2)
    starttime = time.clock()
    tryexcept(nbr)
    stoptime = time.clock()
    elapsed = stoptime - starttime
    print('try-except: %.2f seconds' % elapsed)
```

running `time_iftry`

on a MacBook 2.2 Ghz Intel Core 2 Duo

```
$ python time_iftry.py
#divisions by zero : 499586
if-else: 6.50 seconds

#divisions by zero : 499586
try-except: 7.68 seconds
$
```

Conclusion: try-except costs more.

But then: 50% of cases is not exceptional.
$\rightarrow$ improper use of `try-except`

complexity and cost
timing Python code

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

import module or from module import?

Task: compute `y = sin(1.2)`.

```
>>> import math
>>> y = math.sin(1.2)
>>> y
0.9320390859672263
```

```
>>> from math import sin
>>> y = sin(1.2)
>>> y
0.9320390859672263
```

Question: *does it matter which way we do it?*

the module `timeit`

In the previous experiment, we executed the division many times to notice the effect.

The `timeit` module has functionality for running a code segment repeatedly (without having to write a loop).

```
import timeit
t = timeit.Timer('< function call >')
t.timeit(< number of runs >)
```

→ prints cpu time in seconds

the script `time_sin.py` – using `timeit`

```
"""
With the timeit module we see the difference in
efficiency between math.sin and imported sin.
"""
import timeit
T = timeit.Timer('sin(1.2)', setup='from math import s
S = T.timeit(100000000)
print('%.2f seconds ' % S)
T = timeit.Timer('math.sin(1.2)', setup='import math')
S = T.timeit(100000000)
print('%.2f seconds ' % S)
```

running `time_sin.py`

on a MacBook 2.2 Ghz Intel Core 2 Duo

```
$ python time_sin.py
28.26 seconds
39.21 seconds
```

39.21/28.26 = 1.387

→ `math.sin()` is about 40% slower than `sin()`

complexity and cost
timing Python code

1 Complexity and Cost
- measuring complexity: big-oh
- complexity classes
- counting flops: floating-point operations

2 Cost of Algorithms
- timing Python programs
- the unix command time
- try-except costs more than if-else
- importing modules or importing functions?
- the cost of working with files
- the efficiency of list comprehensions

3 Summary + Assignments

The Cost of Working with Files

an example with nonzero system time

Consider the following task:

1. take random numbers uniformly in [0,1],
2. compute the largest number in the list.

An *inefficient* solution:

1. write the random numbers to file,
2. read the numbers from file
   and compute the maximum.

how inefficient is this method?

Note that very little internal memory is used.

the function `numbers_on_file`

```
def numbers_on_file(nbr):
    """
    Takes n numbers uniformly in [0,1]
    and writes the numbers to a file.
    Closes and reopens the file to
    find the largest number.
    """
    from random import uniform as u
    workfile = open('work', 'w')
    for _ in range(nbr):
        workfile.write(str(u(0, 1)) + '\n')
    workfile.close()
    maxnum = 0.0
    workfile = open('work', 'r')
    while True:
        line = workfile.readline()
        if line == '':
            break
        data = float(line[:-1])
        if data > maxnum:
            maxnum = data
    print('max = ', maxnum)
```

`os.times()`: user cpu, system, and elapsed time

The total elapsed time consists of

- user cpu time: time spent on user process,
- system time: time spent by operating system.

Interesting to separate user cpu from system time.

```
import os
start = os.times()
< code to be timed >
stop = os.times()
print 'user cpu time : %.4f' % (stop[0] - start[0])
print '   system time : %.4f' % (stop[1] - start[1])
print ' elapsed time : %.4f' % (stop[4] - start[4])
```

the function main()

```
def main():
    """
    Calls numbers_on_file.
    """
    import os
    nbr = 1000000
    start = os.times()
    numbers_on_file(nbr)
    stop = os.times()
    print('user cpu time : %.4f' % (stop[0] - start[0]))
    print('  system time : %.4f' % (stop[1] - start[1]))
    print(' elapsed time : %.4f' % (stop[4] - start[4]))
```

running `time_filework.py`

on a Linux machine:

```
$ time python time_filework.py
max =  0.999999415171
user cpu time : 15.0400
  system time : 0.1900
 elapsed time : 15.2400
15.1u 0.1s 0:15.30 99.9% 0+0k 0+0io 427pf+0w
```

`time` confirms user, system, and elapsed time

System times may fluctuate, because influenced by other programs running at the same time.
Programs working with files on flash drive
may run faster if files are on hard disk.

complexity and cost
timing Python code

1. Complexity and Cost
   - measuring complexity: big-oh
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - the unix command time
   - try-except costs more than if-else
   - importing modules or importing functions?
   - the cost of working with files
   - the efficiency of list comprehensions
3. Summary + Assignments

profiling Python functions using the profile module

Profiling code often reveals computational bottlenecks.

Simple to use:

```
import profile

profile.run(' < function call > ')
```

$\rightarrow$ provides a detailed execution summary

Why do we care about list comprehensions?

Task: take `n` random numbers uniformly from [0,1].

Two ways to do it:

1. with a loop:

```
from random import uniform
L = []
for _ in range(n):
    L.append(uniform(0, 1))
```

2. with a list comprehension:

```
from random import uniform
L = [uniform(0, 1) for x in range(n)]
```

Is shorter code more efficient?

the function `sumlist1`

```
def sumlist1(nbr):
    """
    Generates nbr random numbers uniformly
    distributed in [0,1] and prints the sum.
    """
    from random import uniform, seed
    seed(2)
    num = []
    for _ in range(nbr):
        num.append(uniform(0, 1))
    print(sum(num))
```

the function `sumlist2`

```
def sumlist2(nbr):
    """
    Uses list comprehensions to generate nbr
    random numbers in [0,1] and prints the sum.
    """
    from random import uniform, seed
    seed(2)
    num = [uniform(0, 1) for x in range(nbr)]
    print(sum(num))
```

the function main()

```
def main():
    """
    Profiles the two summations.
    """
    import profile
    profile.run('sumlist1(10000)')
    profile.run('sumlist2(10000)')

if __name__ == "__main__":
    main()
```

running `time_listcomp.py`

Redirect output to `/tmp/out`:

```
$ python time_listcomp.py > /tmp/out
```

Summary of the two `profile.run()` statements:

```
4978.20408089
   30027 function calls in 0.411 CPU seconds

< stuff deleted from /tmp/out >

4978.20408089
   20008 function calls in 0.268 CPU seconds

< stuff deleted from /tmp/out >
```

Summary + Assignments

See §11.5 in *Computer Science, an overview*.

Assignments:

1. Examine the space complexity to sort n numbers.
   Express the memory use as a function of n.
2. If a (double) float occupies 8 bytes, how much space is needed to sort one million numbers? Find out how much internal memory your computer has. What is the largest list you could sort?
3. Modify the class `flopfloats.py` so that multiplications and divisions are counted separately from the additions and subtractions.
4. Run `floppoly` for degrees d ranging from 2 to 20
   and record the flops.
5. Look at the code for `floppoly` and find a formula for its cost in function of d.

More Assignments

6. To handle division by zero, we could have used the name of the proper exception in the handler.
   Modify `time_iftry.py` using the proper name for the exception and compare the timings.
   Does knowing the name of the exception help?
7. Use `timeit` in the script `time_iftry.py`.
8. Make `time_filework` more efficient by avoiding the use of files. Compare between storing all numbers in a list and merging the loop which generates the numbers with the loop which computes the maximum.