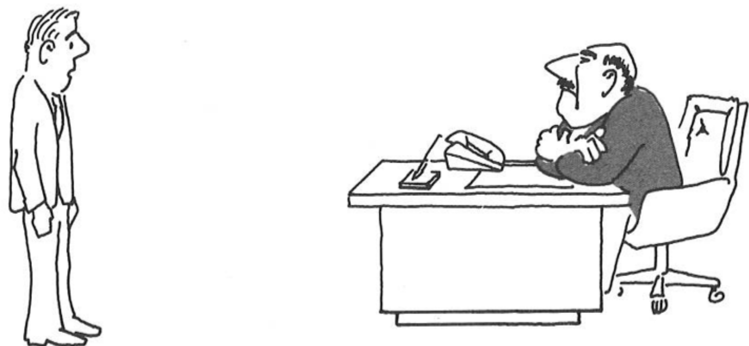
Outline

1. Complexity and Cost
   - measuring complexity: big o
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - examples of cost considerations

MCS 260 Lecture 31
Introduction to Computer Science
Jan Verschelde, 19 July 2023

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 o
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - examples of cost considerations

The Big O Notation

to measure complexity

Let n be the dimension of our problem.

Definition (Big O)

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 O defines the order of complexity, some examples:

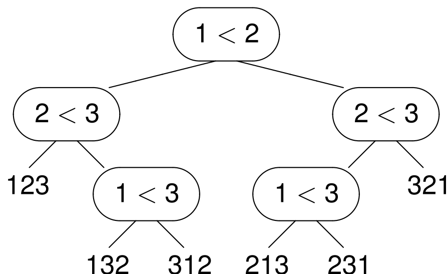
- f is $O(\log(n))$: logarithmic in n
- f is $O(n)$: linear in n
- f is $O(n \log(n))$: quasilinear in n
- f is $O(n^2)$: quadratic in n
- f is $O(2^n)$: exponential in n

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 o
   - complexity classes
   - counting flops: floating-point operations
2. Cost of Algorithms
   - timing Python programs
   - examples of cost considerations

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 number of 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 o
   - complexity classes
   - counting flops: floating-point operations

2. Cost of Algorithms
   - timing Python programs
   - examples of cost considerations

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.

complexity and cost
timing Python code

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

2. Cost of Algorithms
    - timing Python programs
    - examples of cost considerations

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. measure time at the command line, 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 o
   - complexity classes
   - counting flops: floating-point operations

2. Cost of Algorithms
   - timing Python programs
   - examples of cost considerations

examples of cost considerations

Consider the following questions:

- Is the time to sort a list of n elements $O(n \log(n))$?
- Does try-except cost more than if-else?
- Importing the module or from module import a function?
- What is the cost of working with files?
- Is shorter code more efficient?
  Why we care about list comprehensions.

Exercises

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 Exercises

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.