CSE202
Design and Analysis of Algorithms

Bruno Salvy

Week 1: Overview & Basics

I. Overview of the Course

Summary of Last Year

CSE 101: Computer programming.
Standard datatypes, fundamental algorithms.

B. Smith

CSE 102: Advanced programming.
Data structures and algorithms.
Graph algorithms. Optimization.

P.-Y. Strub

CSE 103: Introduction to algorithms.
Sorting. Graphs. Dynamic programming.
Greedy algorithms.

I. Mackie

Plan for this Course

Introduction	1
Divide & Conquer	2–5
Randomization	6–8
Amortization, balancing	9–10
String Algorithms	11–12
P vs NP	13–14

Basic principles through many examples

data-structures along the way

Organization

Videos, for the time being.

~~Lectures: Tuesdays 15:45—17:00 amphi Cauchy~~

Consultation: Tuesdays 15:30—15:45 starting next week

Tutorials in Python: Thursdays 13:15—15:15 rooms 35 & 36

Material: see Moodle

Questions or comments: Bruno.Salvy@inria.fr

E. Fusy

A. Pouly

Assessment

	Week	Ratio
Weekly assignments	1—14	10 %
Weekly tutorials	1—14	10 %
Midterm	8	40 %
Final	16	40 %

Midterm: programming
Final: written exam

Exam rules:
No collaboration,
no laptop,
no internet.

II. Algorithms

An algorithm is a finite answer to an infinite number of questions.
Stephen Kleene

Example: Binary Powering

Algorithm:

1. A well-specified problem

Input: $(x, n) \in \mathbb{A} \times \mathbb{N}$
Output: x^n

2. A way to solve it

$$x^n = \begin{cases} (x^{n/2})^2, & \text{for even } n \\ (x^{(n-1)/2})^2 x, & \text{otherwise} \end{cases}$$

Implemented in programs:

```
def binpow(x,n):
    if n==0: return 1
    tmp=binpow(x,n//2)
    tmp=tmp*tmp
    if n%2==0: return tmp
    return tmp*x
```

Etymology

Algorithm comes from al-Khwarizmi (c. 780-850), a Persian mathematician who worked in Baghdad.

Wrote many books, including:

- one on solving linear and quadratic equations; al'jabr in his title became *algebra*
- one *On the Calculation with Hindu Numerals*

A data-structure

strings of 0,…,9
with possibly one "."

Algorithms

for +,-,×,÷
in good complexity

Next week:
faster methods

Correctness

Def. An algorithm is *correct* if

1. it terminates;
2. it computes what its specification claims.

A useful proof technique: look for variants and invariants.

```
# Input:  x that can be multiplied
#         n nonnegative integer
# Output: x**n
def binpow(x,n):
    if n==0: return 1
    # n>0
    tmp=binpow(x,n//2) # n//2 < n
    # tmp = x**(n//2)
    tmp=tmp*tmp # tmp = x**(2*(n//2))
    if n%2==0: return tmp
    return tmp*x
```

Specification

$n > 0 \Rightarrow n//2 < n$ proves termination

(Obvious) invariants

Correctness by induction

Correctness: a less obvious example

```
# Input:  x that can be multiplied
#         n nonnegative integer
# Output: x**n
def binpow2(x,n):# let n0=n, x0=x
    if n==0: return 1
    res = 1
    while n>1: # res*(x**n)=x0**n0
      if n%2==1: res *= x
      x *= x
      n //= 2
    return res*x
```

Termination: same argument

Correctness: invariant

Proof. In one iteration of the loop, `res*(x**n)` becomes

`res*(x**2)**(n//2)=res*(x**n)` for even `n`

`res*x*(x**2)**(n//2)=res*(x**n)` for odd `n`

Termination is a very hard problem

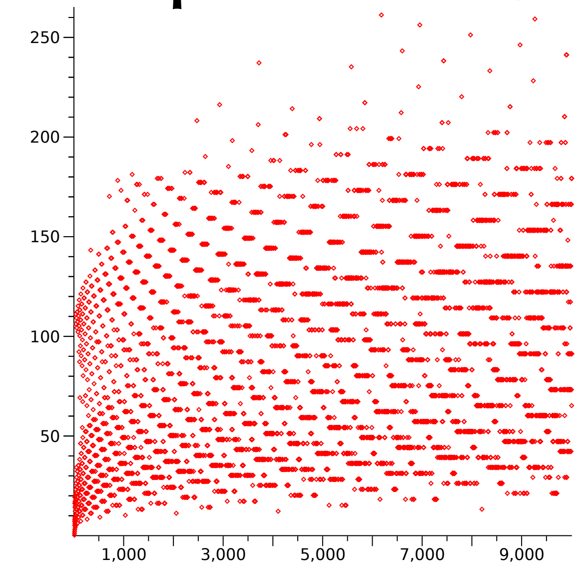
The general problem is undecidable. (See CSE 203)

Already hard for seemingly simple programs:

```
def syracuse(n):
    if n==1: return
    if n%2==0: return syracuse(n//2)
    return syracuse(3*n+1)
```

Conjecture. (3n+1 conjecture, Syracuse problem,…)
This program terminates.

Open since 1937!



stopping times

III. Complexity

Complexity

How long will my program take?
Do I have enough memory?

The scientific approach:

1. Experiment for various sizes;
2. Model;
3. Analyse the model;
4. Validate with experiments;
5. If necessary, go to 2.

Experimental Determination of (Polynomial) Complexity

If the time for a computation grows like $C(n) \sim K n^{\alpha} \log^{p} n$

then doubling n should take time $C(2n) \sim K 2^{\alpha} n^{\alpha} \log^{p} n$

so that
$$\alpha \approx \log_2 \frac{C(2n)}{C(n)}.$$

Example: matrix product

n	10	20	40	80
time (s)	0,023	0,158	1,159	9,075
ln2(ratio)		2.78	2.88	2.97

```
from sympy.matrices import randMatrix
import timeit

def testMatrixMul(size,nbtests):
    total = 0
    for i in range(nbtests):
      A = randMatrix(size)*1.
      B = randMatrix(size)*1.
      def doit():
        return A*B
      total += timeit.timeit(doit,number=1)
    return total/nbtests
```

suggests **cubic** complexity.

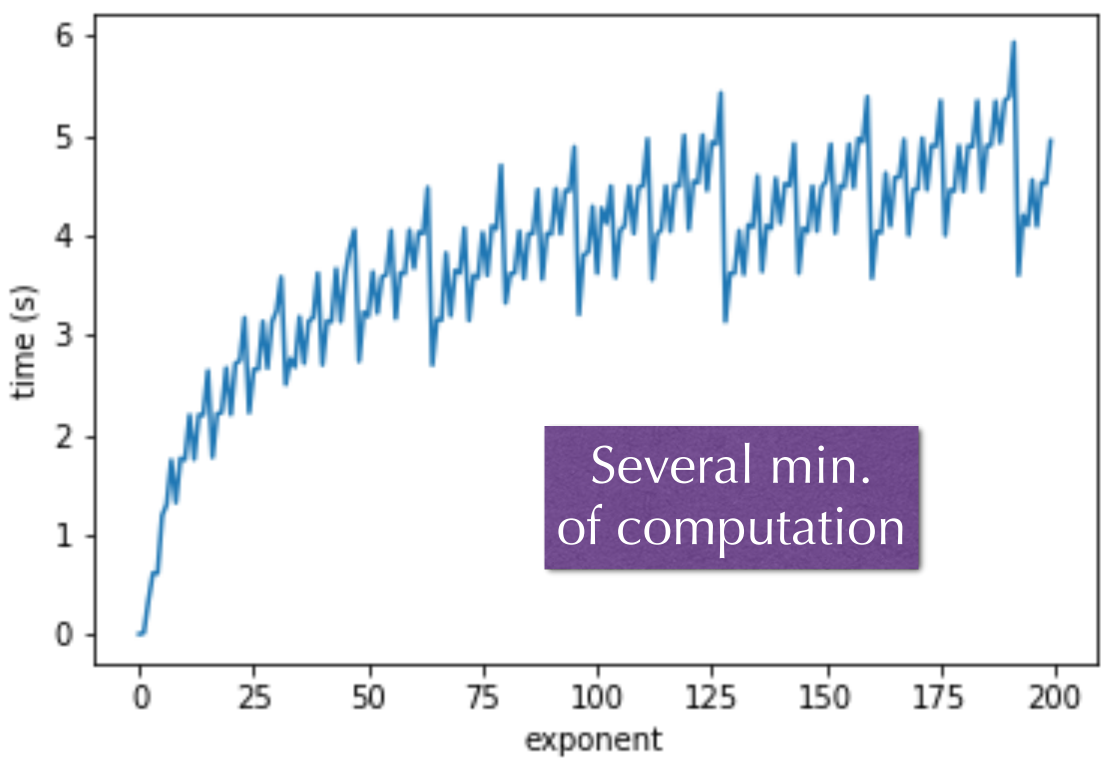
Blackboard:
3 is expected

Binary Powering 1. Model

1. Experiment

```
from sympy.matrices import randMatrix
import timeit

def test(size,maxpow):
    A = randMatrix(size)*1.
    val = [0 for i in range(maxpow)]
    for i in range(maxpow):
        def doit():
            return binpow(A,i)
        val[i] = timeit.timeit(doit,number=3)
    return val
```



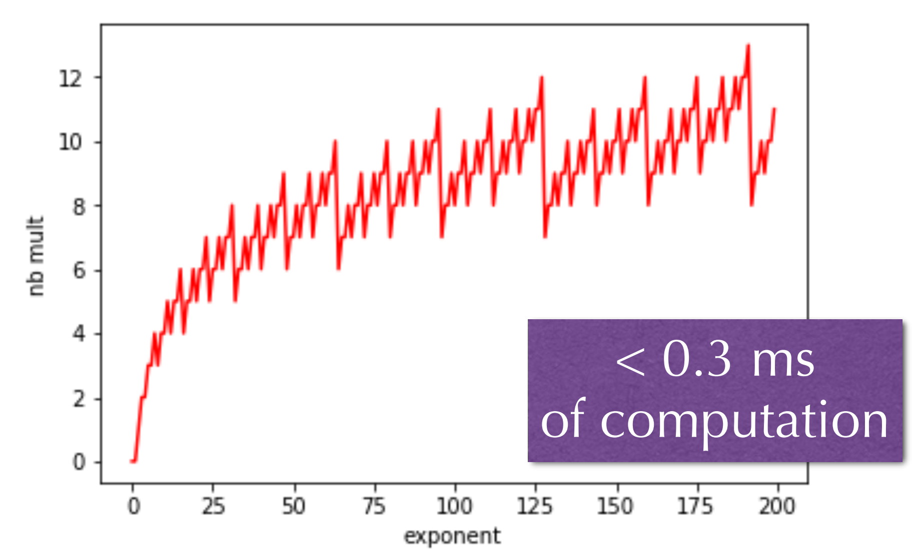
x is a 20x20 matrix of floats

```
def binpow(x,n):
    if n==0: return 1
    if n==1: return x
    tmp=binpow(x,n//2)
    tmp=tmp*tmp
    if n%2==0: return tmp
    return tmp*x
```

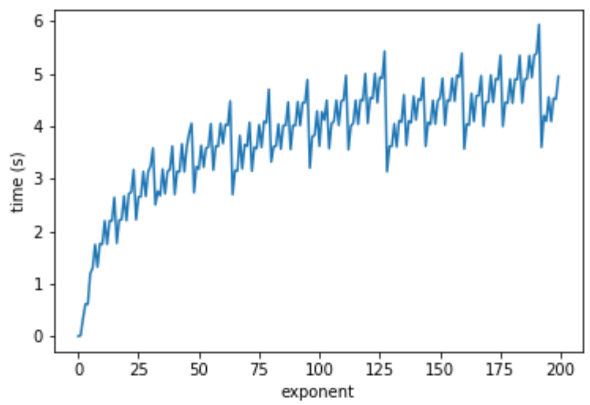
2. Model: count multiplications only

$$C(n) = \begin{cases} C(n/2) + 1, & \text{for even } n > 0 \\ C((n-1)/2) + 2, & \text{for odd } n > 1 \end{cases}$$

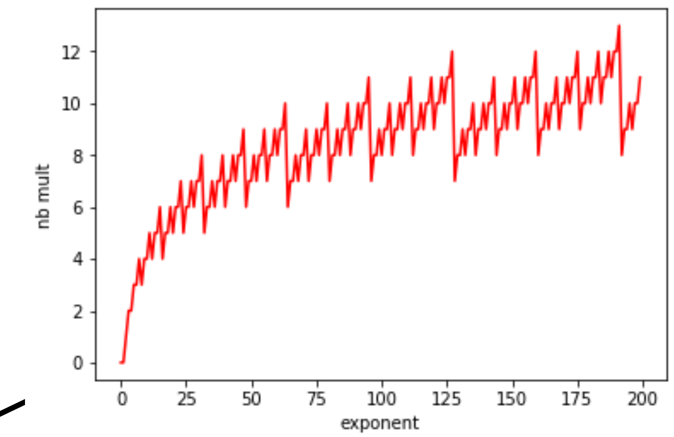
$$C(0) = C(1) = 0.$$



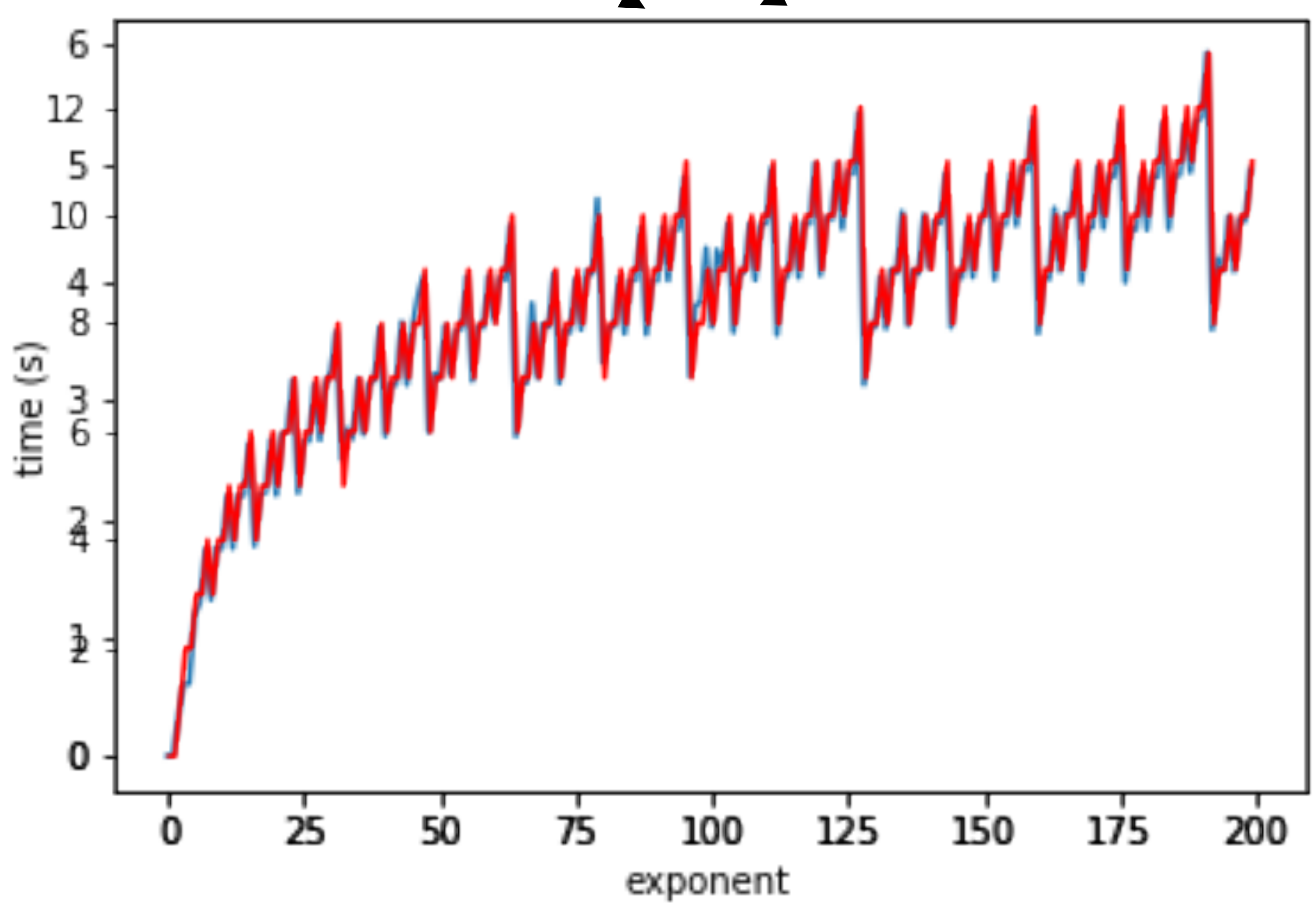
Binary Powering 2. Comparison



time



nb mult



Asymptotically, the cost of multiplications dominates

Binary Powering 3. Analysis

$$C(n) = 1 + \begin{cases} C(n/2), & \text{for even } n > 0 \\ C((n-1)/2) + 1, & \text{for odd } n > 1 \end{cases} \quad \text{with } C(0) = C(1) = 0$$

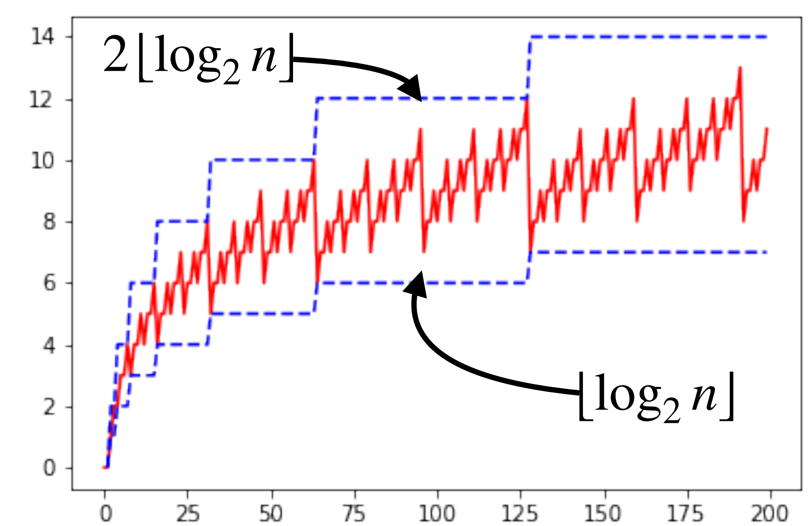
Lemma. For $n \geq 1$, $C(n) = \lfloor \log_2 n \rfloor - 1 + \lambda(n)$,
where $\lambda(n)$ is the number of 1's in the binary expansion of n.

Blackboard proof

Ex. $82 = 64 + 16 + 2 = \overline{1010010}^2$ → 6-1+3=8 mult.

Consequence:

$$\underbrace{\boxed{\lfloor \log_2 n \rfloor \leq C(n) \leq 2\lfloor \log_2 n \rfloor}}_{C(n)=O(\log n)}$$



Notation

Recall:

$$f(n) \sim g(n) \quad \text{means} \quad \lim_{n\to\infty} f(n)/g(n) = 1$$

$$f(n) = O(g(n)) \quad \text{means} \quad \exists K\, \exists M\, \forall n \geq M,\, |f(n)| \leq Kg(n)$$

$$f(n) = \Theta(g(n)) \quad \text{means} \quad f(n) = O(g(n)) \text{ and } g(n) = O(f(n))$$

Exs.:

$$\log(2n) = O(\log n)$$

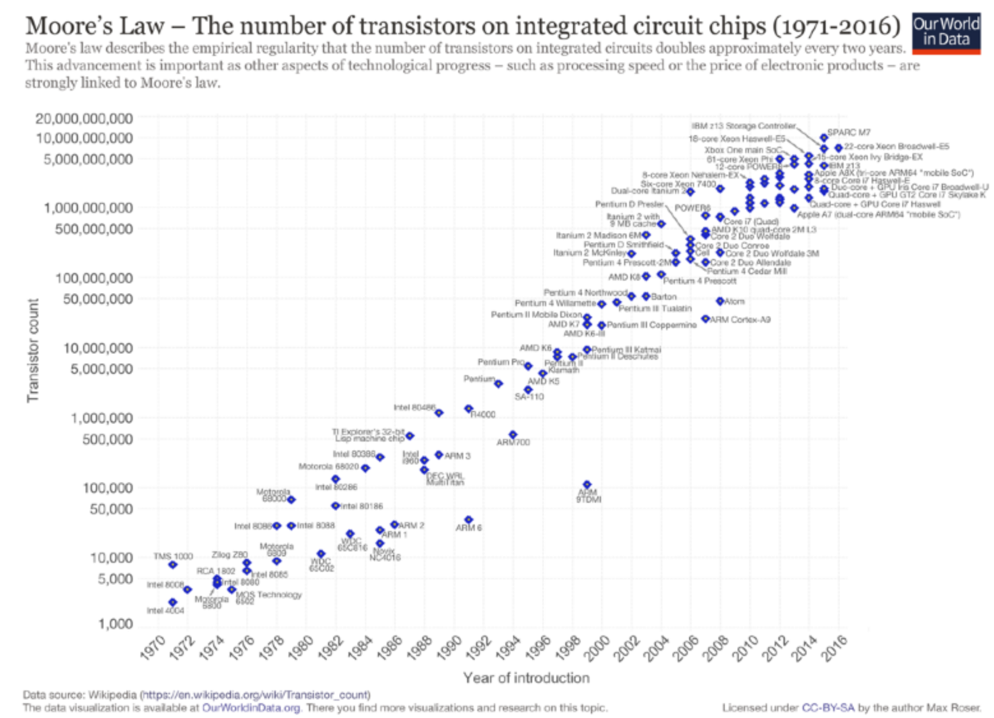
$$10^{10^{10}} n = O(n)$$

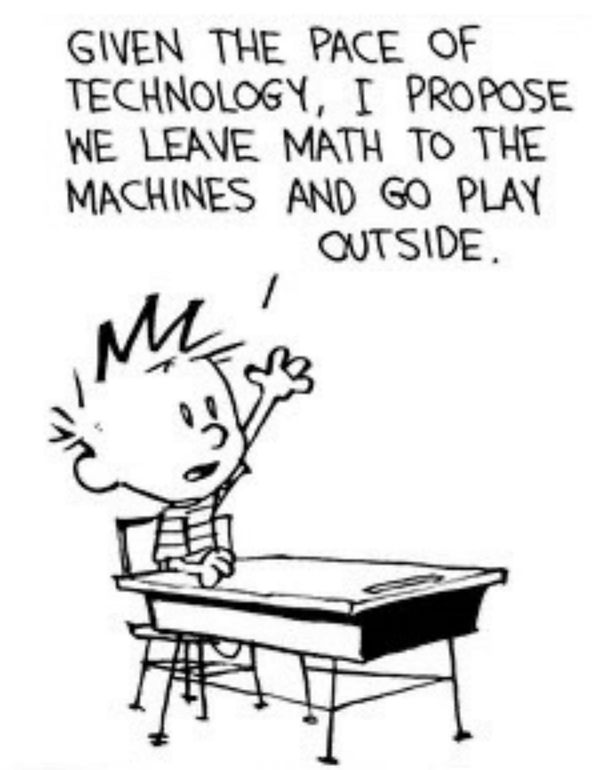
$$10^{10^{10}} n + n^2 = O(n^2)$$

$$n + n^2 = O(n^{20})$$

Moore's "law"

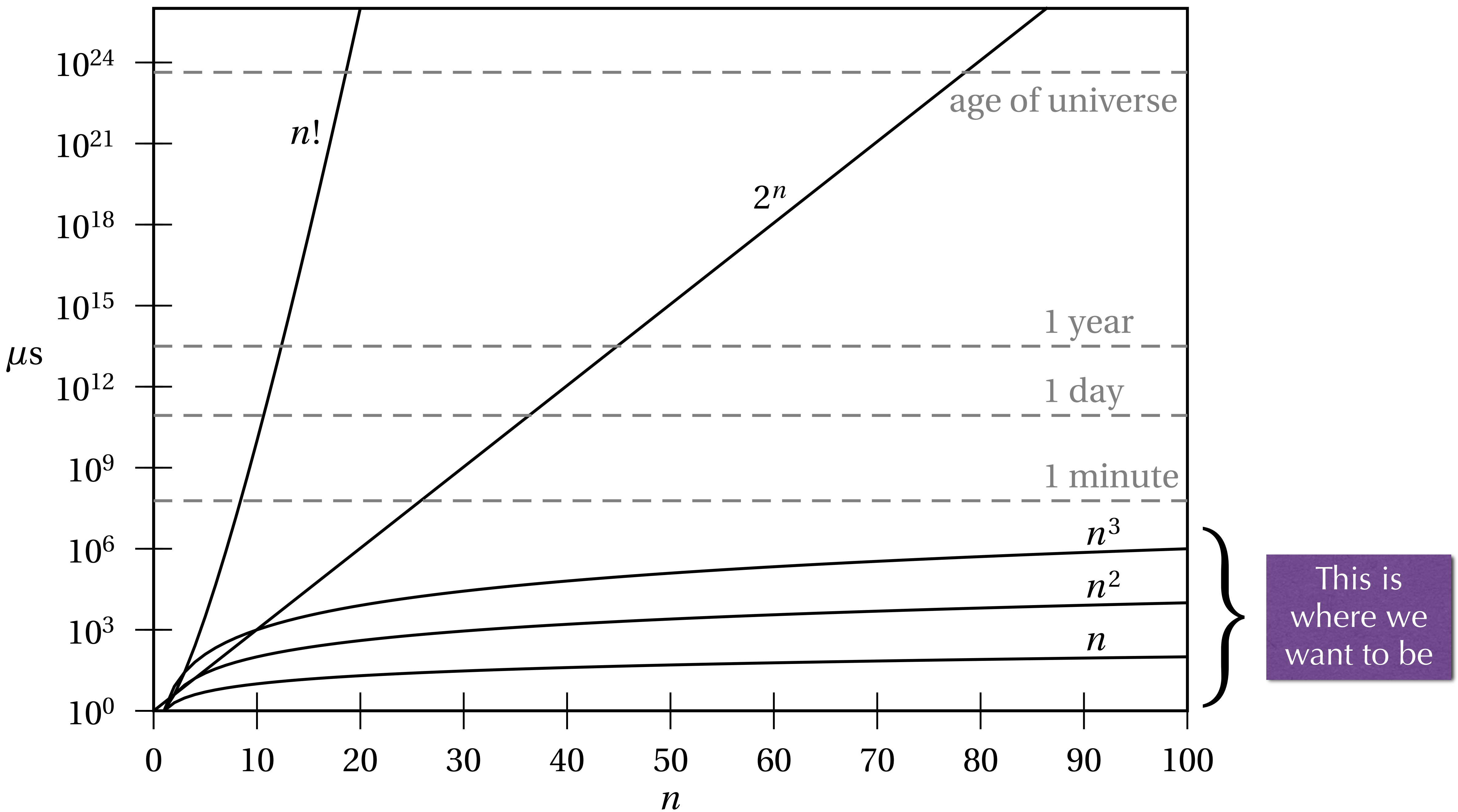
Gordon Moore, co-founder of Intel, predicted in 1965 that the number of transistors on integrated circuits would double every year for 10 years.





The expression Moore's "law" is commonly used to mean that the speed and memory of computers is expected to double every 18 months.

Orders of Growth



Moore's "law" means a small vertical shift from one machine to the next

Picture due to Moore & Mertens (2011).

IV. Lower Bounds

Complexity of a Problem

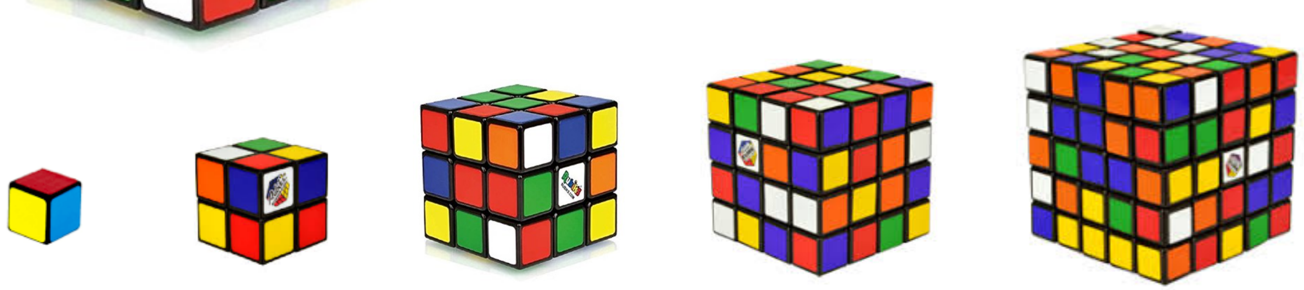
Def. The *complexity of a problem* is that of the most efficient (possibly unknown) algorithm that solves it.

Ex. Sorting n elements has complexity $O(n \log n)$ comparisons.
Proof. Mergesort (CSE103) reaches the bound.

Ex. Sorting n elements has complexity $\Theta(n \log n)$ comparisons.
Proof. k comparisons cannot distinguish more than 2^k permutations and $\log_2 n! \sim n \log_2 n$.

Detailed proof on the blackboard.

Ex. Rubik's 3x3x3 cube has complexity $O(1)$.
Proof. Store the solutions of each of the finitely many configurations, and look them up.



… would be a better problem.

Complexity of Powering

$$(x, n) \in \mathbb{A} \times \mathbb{N} \mapsto x^n \in \mathbb{A}$$

We already know it is $O(\log n)$ multiplications in $\mathbb{A}$.

Can this be improved?

Lower bounds on the complexity require a precise definition (a model) of what operations the "most efficient" algorithm can perform.

Ex. If the only available operation in $\mathbb{A}$ is multiplication, x^{2^k} requires k multiplications, so that $\log_2 n$ is a lower bound.

Ex. In floating point arithmetic, $x^n = \exp(n \log x)$ and the complexity hardly depends on n.

Simple Lower Bounds

In most useful models, reading the input and writing the output take time. Then,

$$\boxed{\text{size(Input)+size(Output)} \leq \text{complexity.}}$$

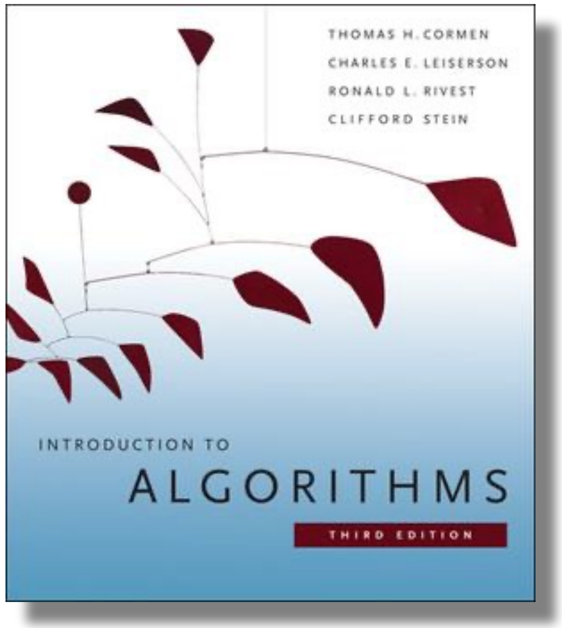
Examples:

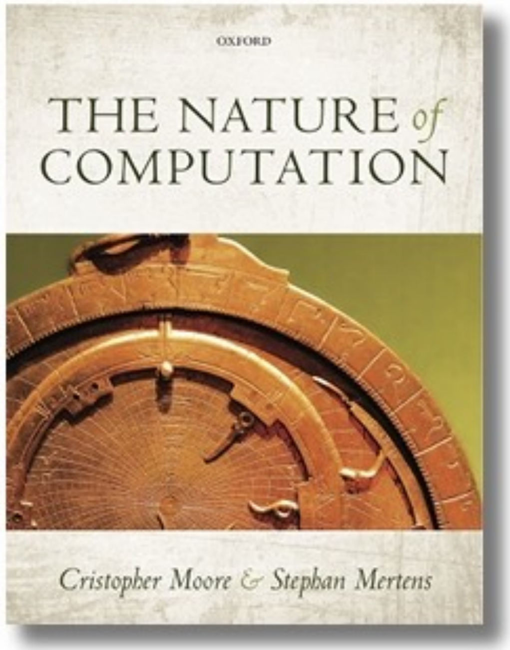
Problem	Input	Simple Lower Bound	Best known algorithm	Measure
Sorting	n elts	n	O(n log n)	comparisons
Polynomial multiplication	degree n	n	O(n log n)	ops on coeffs
Matrix multiplication	size n x n	n^2	$O(n^{2.373})$	ops on coeffs
Subset sum	n integers	n	$2^{O(n)}$	time

References

The slides are designed to be self-contained.

They were prepared using the following
books that I recommend if you want to learn more:





Next

Assignment this week: optimal powering

Next tutorial: fast powering via addition chains

Next week: fast multiplication

Feedback

Moodle

Questions or comments: Bruno.Salvy@inria.fr