Winter 2016
COMP-250: Introduction to Computer Science

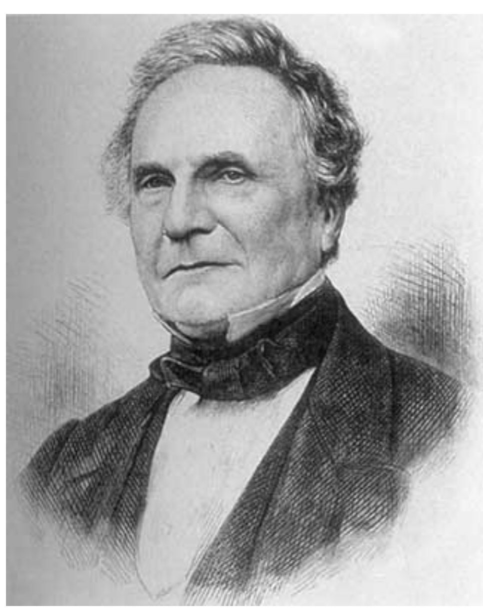
Lecture 9, February 9, 2016

Running Times and Asymptotic Notation

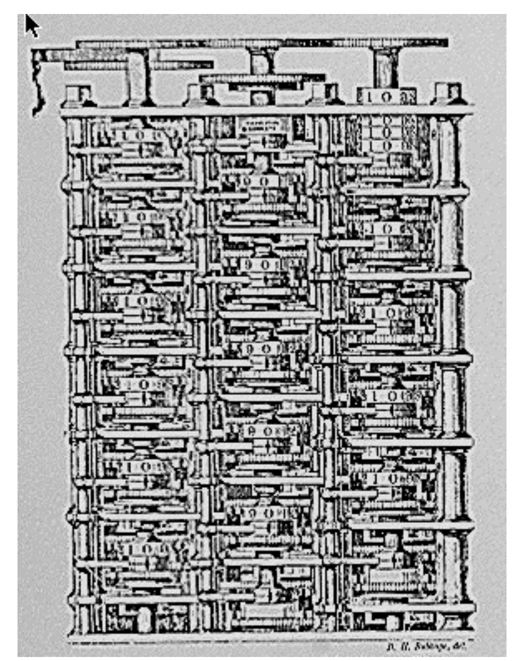
Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?

- Charles Babbage



Charles Babbage (1864)



Analytic Engine (schematic)

Computational Tractability

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that tries every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

even worse : N ! for some problems

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants a > 0 and d > 0 such that on every input of size N, its running time is bounded by a N^d steps.

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.

  Primality testing
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

  simplex method
  Unix grep

Why it matters ?

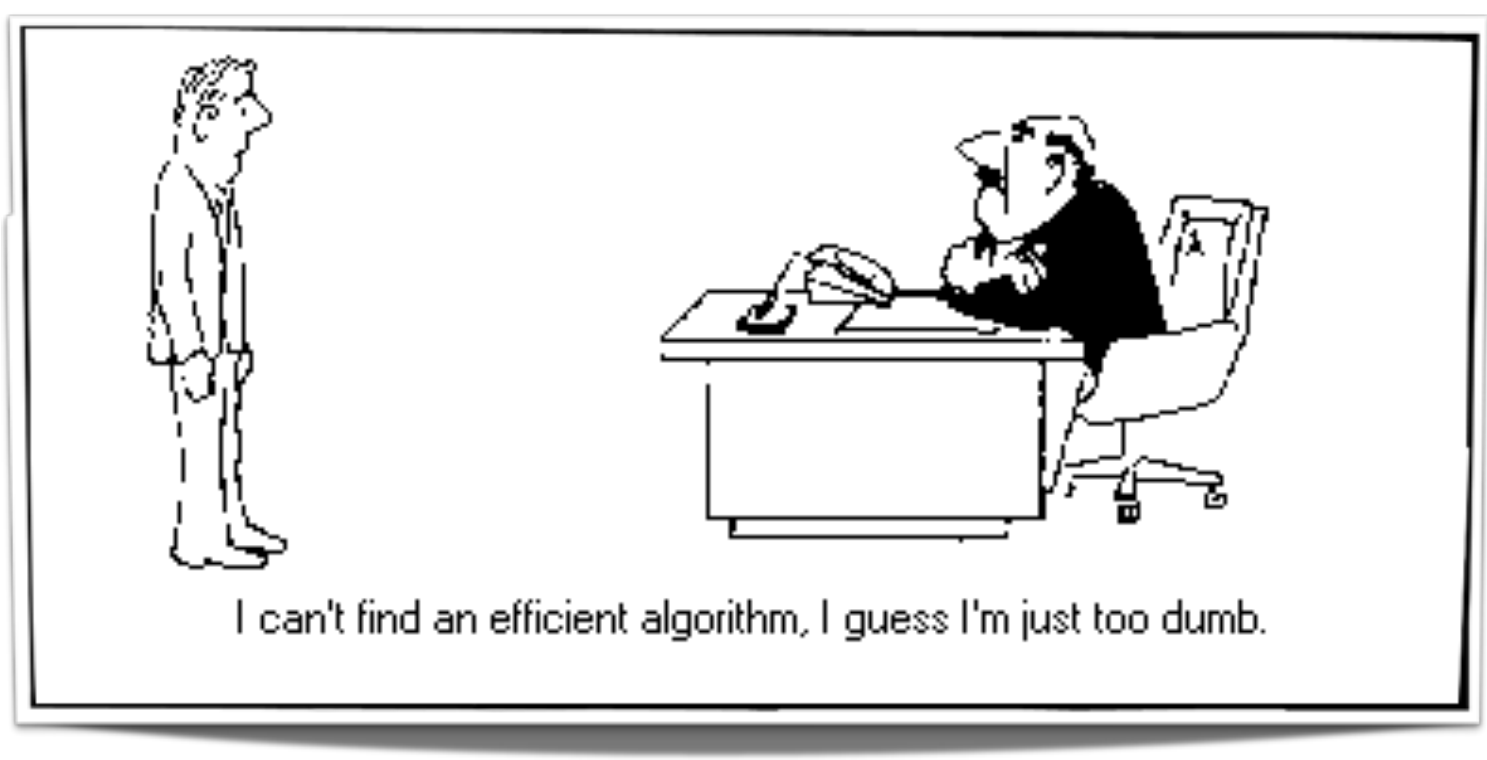
Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Note: age of Universe ~ 10^{10} years…

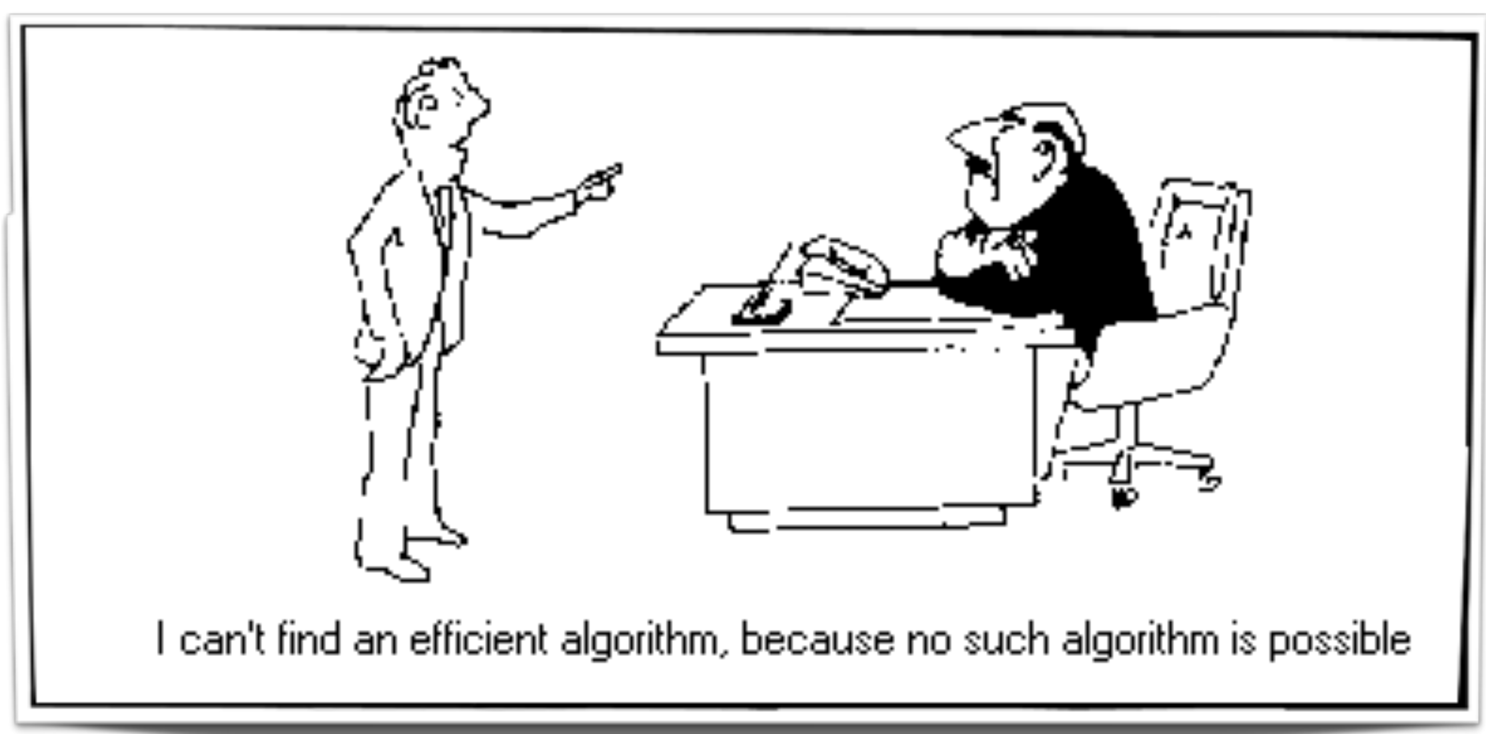
Computer Science Approach to problem solving

- If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem ???



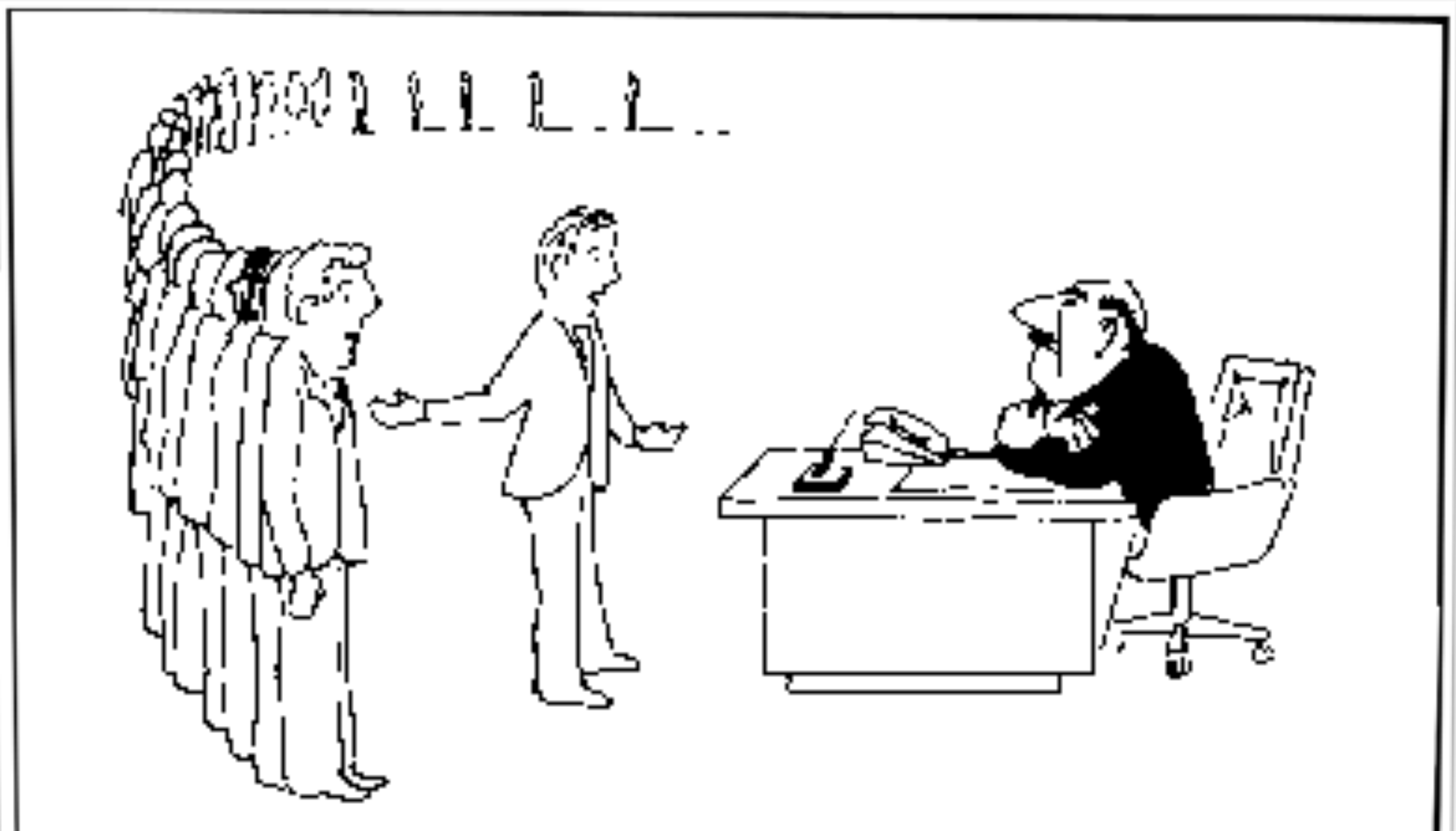
Computer Science Approach to problem solving

- Are there some problems that cannot be solved at all ? and, are there problems that cannot be solved efficiently ??



Computer Science Approach to problem solving

- If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem ???



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

Asymptotic order of Growth and Notation

Asymptotic order of Growth and Notation

Upper bounds. $T(n)$ **is** $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ **is** $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ **is** $\Theta(f(n))$ if $T(n)$ **is** both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ **is** $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ **is not** $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Asymptotic order of Growth and Notation

Upper bounds. $T(n)$ **is** $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ **is** $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ **is** $O(n^2)$ since there exists $c = 81$ and $n_0 = 1$ such that for all $n \geq 1$ we have $T(n) \leq 32n^2 + 17n^2 + 32n^2 = 81n^2$.
- $T(n)$ **is** $\Omega(n^2)$ since there exists $c = 1$ and $n_0 = 0$ such that for all $n \geq 0$ we have $T(n) \geq n^2$.
- $T(n)$ **is not** $O(n)$ since for all $c > 0$ and $n_0 \geq 0$ there exists $n = \lceil c + 1/c + n_0 \rceil$ such that $T(n) > 32(c+1/c+n_0)^2 + 17(c+1/c+n_0) + 32 \geq c^2 + c \bullet n_0 + 32 \geq cn$.

Asymptotic Notation

Transitivity.

- If f **is** O(g) and g **is** O(h) then f **is** O(h).
- If f **is** Ω(g) and g **is** Ω(h) then f **is** Ω(h).
- If f **is** Θ(g) and g **is** Θ(h) then f **is** Θ(h).

Additivity.

- If f **is** O(h) and g **is** O(h) then f + g **is** O(h).
- If f **is** Ω(h) and g **is** Ω(h) then f + g **is** Ω(h).
- If f **is** Θ(h) and g **is** O(h) then f + g **is** Θ(h).

Frequently Used Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ **is** $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0.

can avoid specifying the base

Logarithms. For every $x > 0$, $\log n$ **is** $O(n^x)$.

log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, n^d **is** $O(r^n)$.

every exponential grows faster than every polynomial

Asymptotic Notation

Sometimes one can also obtain an asymptotically tight bound directly by computing a limit as n goes to infinity. Essentially, if the ratio of functions $f(n)$ and $g(n)$ converges to a positive constant as n goes to infinity, then $f(n)$ is $\Theta(g(n))$.

(2.1) *Let f and g be two functions that*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

exists and is equal to some number $c > 0$. Then $f(n)$ is *$\Theta(g(n))$.*

Proof. We will use the fact that the limit exists and is positive to show that $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, as required by the definition of $\Theta(\cdot)$.

Since

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0,$$

it follows from the definition of a limit that there is some n_0 beyond which the ratio is always between $\frac{1}{2}c$ and $2c$. Thus, $f(n) \le 2cg(n)$ for all $n \ge n_0$, which implies that $f(n)$ is $O(g(n))$; and $f(n) \ge \frac{1}{2}cg(n)$ for all $n \ge n_0$, which implies that $f(n)$ is $\Omega(g(n))$. ∎

Winter 2016
COMP-250: Introduction to Computer Science

Lecture 9, February 9, 2016