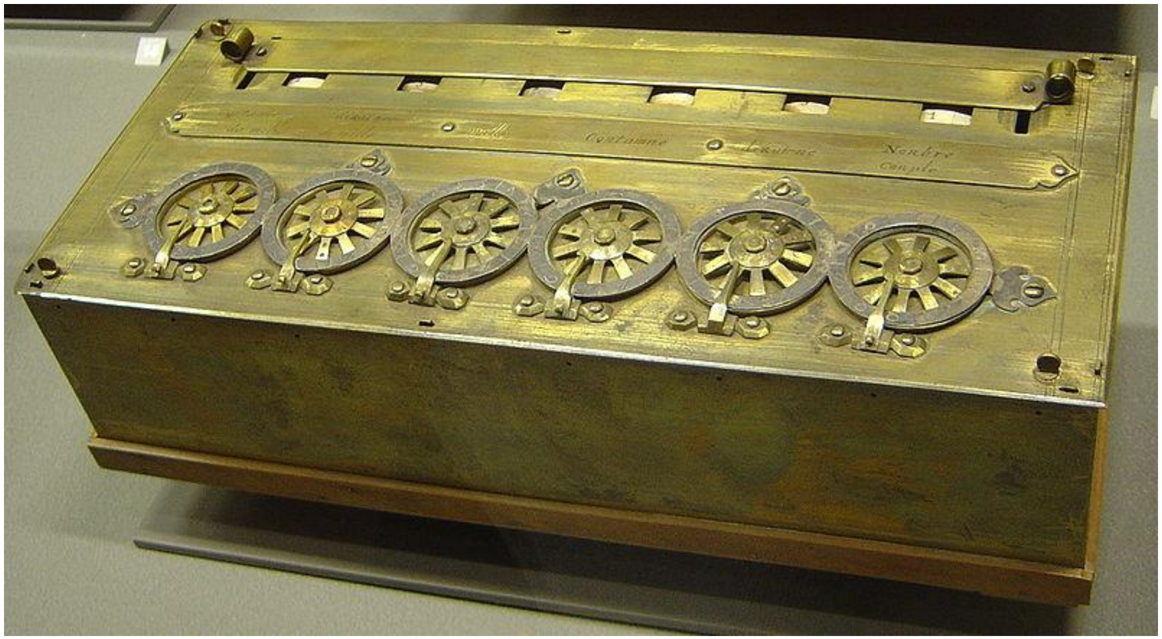
2.1 Computational Tractability

Pascaline



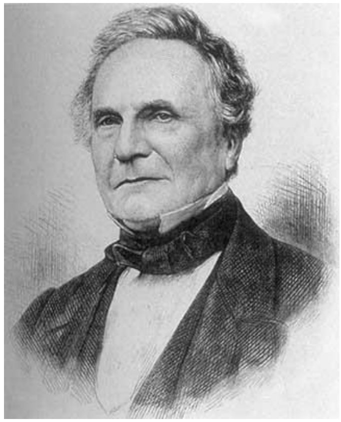
http://en.wikipedia.org/wiki/Pascal%27s_calculator

Blaise Pascal's 17th century calculator (addition and subtraction)

TU Delft

Computational Tractability

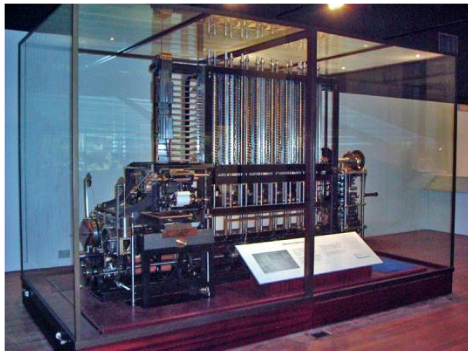
As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*



Charles Babbage (1864)

"father of the computer"

http://en.wikipedia.org/wiki/Babbage#cite_note-0



Difference machine (1991)

http://en.wikipedia.org/wiki/Difference_machine

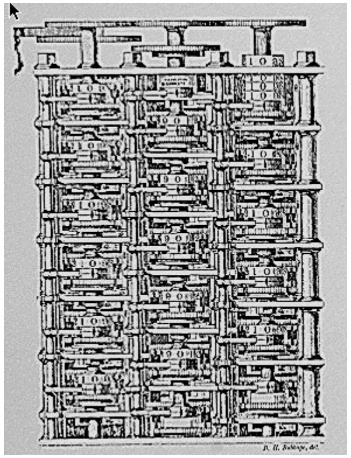
TU Delft

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*

Ada Lovelace (1838)

http://en.wikipedia.org/wiki/Ada_Lovelace



Analytic Engine (schematic)

http://en.wikipedia.org/wiki/Analytical_engine

TU Delft

Computational Tractability

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

- Typically takes 2^n time or worse for inputs of size n.
- Unacceptable in practice: n increased by 1, computation doubles.

Q. What is the brute-force running time for stable matching?

A.

Computational Tractability

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

- Typically takes 2^n time or worse for inputs of size n.
- Unacceptable in practice: n increased by 1, computation doubles.

Q. What is the brute-force running time for stable matching?

A. n! Because first man has n possible women, second man n-1, etc.

TU Delft

Polynomial-Time

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

E.g. If run-time bounded by n^2, then $(2n)^2 = 4n^2$, so slowdown is 4.

TU Delft

Polynomial-Time

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

E.g. If run-time bounded by n^2, then $(2n)^2 = 4n^2$, so slowdown is 4.

Q. What is slow-down when run-time is bounded by $c \cdot n^d$ steps and input doubles?

A.

TU Delft

Polynomial-Time

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

E.g. If run-time bounded by n^2, then $(2n)^2 = 4n^2$, so slowdown is 4.

Q. What is slow-down when run-time is bounded by $c \cdot n^d$ steps and input doubles?

A. cn^d becomes $c(2n)^d = c2^d n^d$, so slowdown is 2^d

Eg. if algorithm uses $40 \cdot n^3$ steps, slowdown is 8.

> There exists constants c > 0 and d > 0 such that on every input of size n, its running time is bounded by $c \cdot n^d$ steps.

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

Worst-Case Analysis

Worst case running time. Obtain (upper) bound on largest possible run time of algorithm on input of a given size n (or n and m for graphs, or…).

- 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.

TU Delft

Worst-Case Polynomial-Time

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

Justification: It really works in practice!

- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents (not $6.02 \times 10^{23} \times n^{20}$ or so).
- 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.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

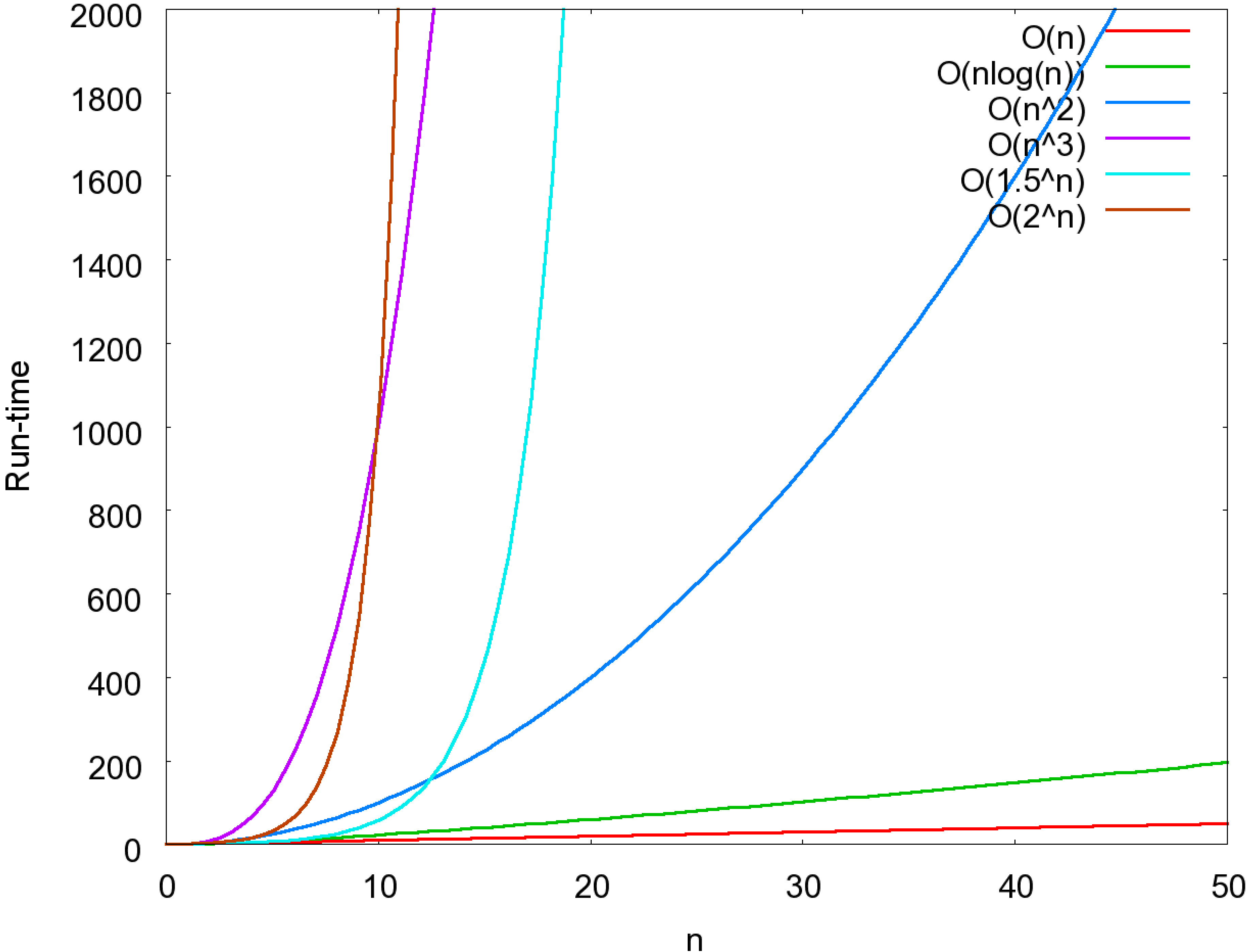
↑ simplex method
Unix grep

TU Delft

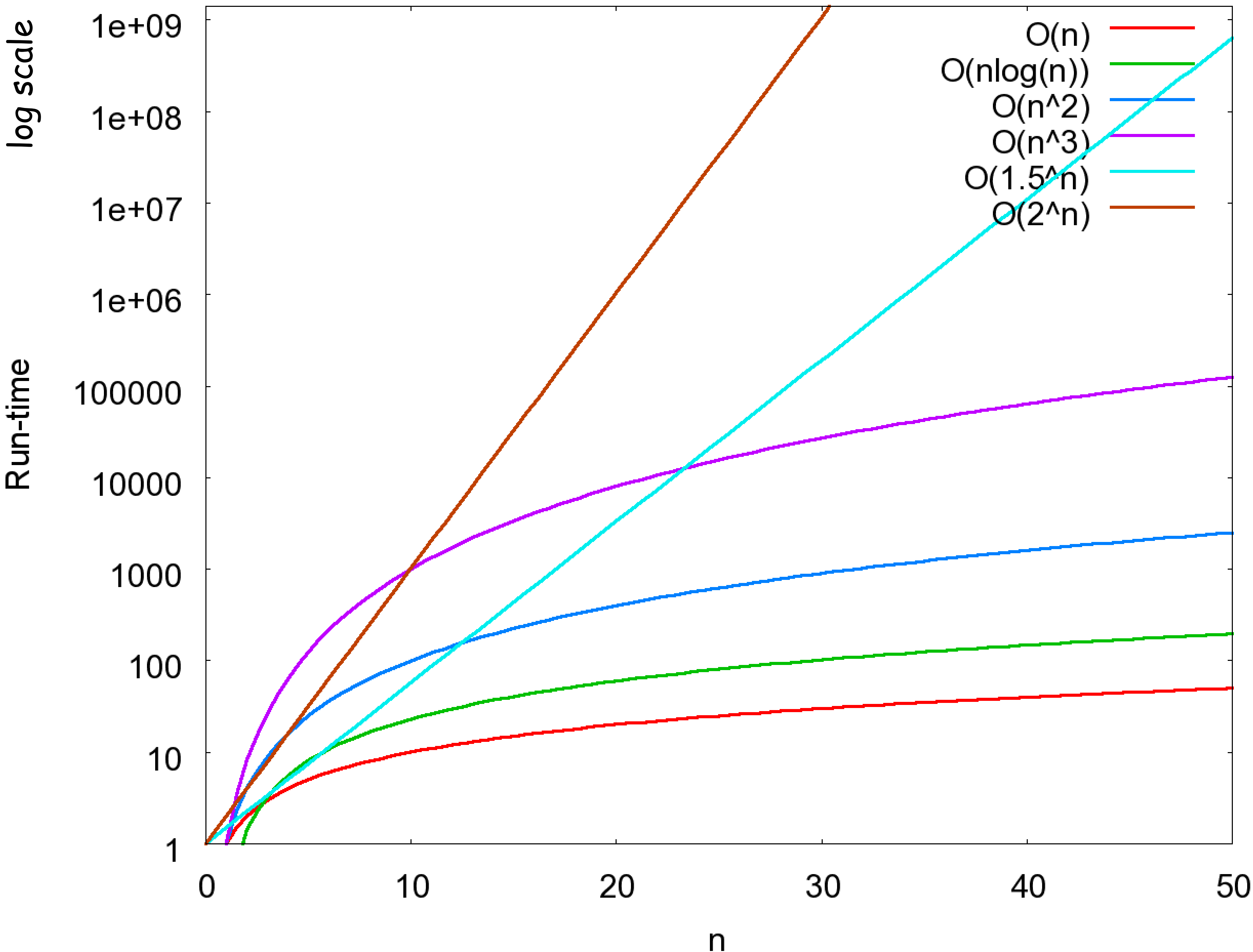
Why It Matters

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



O(n)
O(nlog(n))
O(n^2)
O(n^3)
O(1.5^n)
O(2^n)
Run-time
n
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
10
20
30
40
50


log scale
Run-time
1e+09
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
10
20
30
40
50
n
O(n)
O(nlog(n))
O(n^2)
O(n^3)
O(1.5^n)
O(2^n)