Complexity Analysis of Algorithms

Jordi Cortadella
Department of Computer Science

Estimating runtime

What is the runtime of g(n)?

```
void g(int n) {
  for (int i = 0; i < n; ++i) f();
}
```

$$\text{Runtime}(g(n)) \approx n \cdot \text{Runtime}(f())$$

```
void g(int n) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) f();
}
```

$$\text{Runtime}(g(n)) \approx n^2 \cdot \text{Runtime}(f())$$

Estimating runtime

What is the runtime of g(n)?

```
void g(int n) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j <= i; ++j) f();
}
```

$$
\begin{aligned}
\mathrm{Runtime}(g(n)) &\approx (1+2+3+\cdots+n)\cdot\mathrm{Runtime}(f()) \\
&\approx \frac{n^2+n}{2}\cdot\mathrm{Runtime}(f())
\end{aligned}
$$

Complexity analysis

- A technique to characterize the execution time of an algorithm independently from the machine, the language and the compiler.

- Useful for:
  - evaluating the variations of execution time with regard to the input data
  - comparing algorithms

- We are typically interested in the execution time of large instances of a problem, e.g., when $n \to \infty$, (asymptotic complexity).

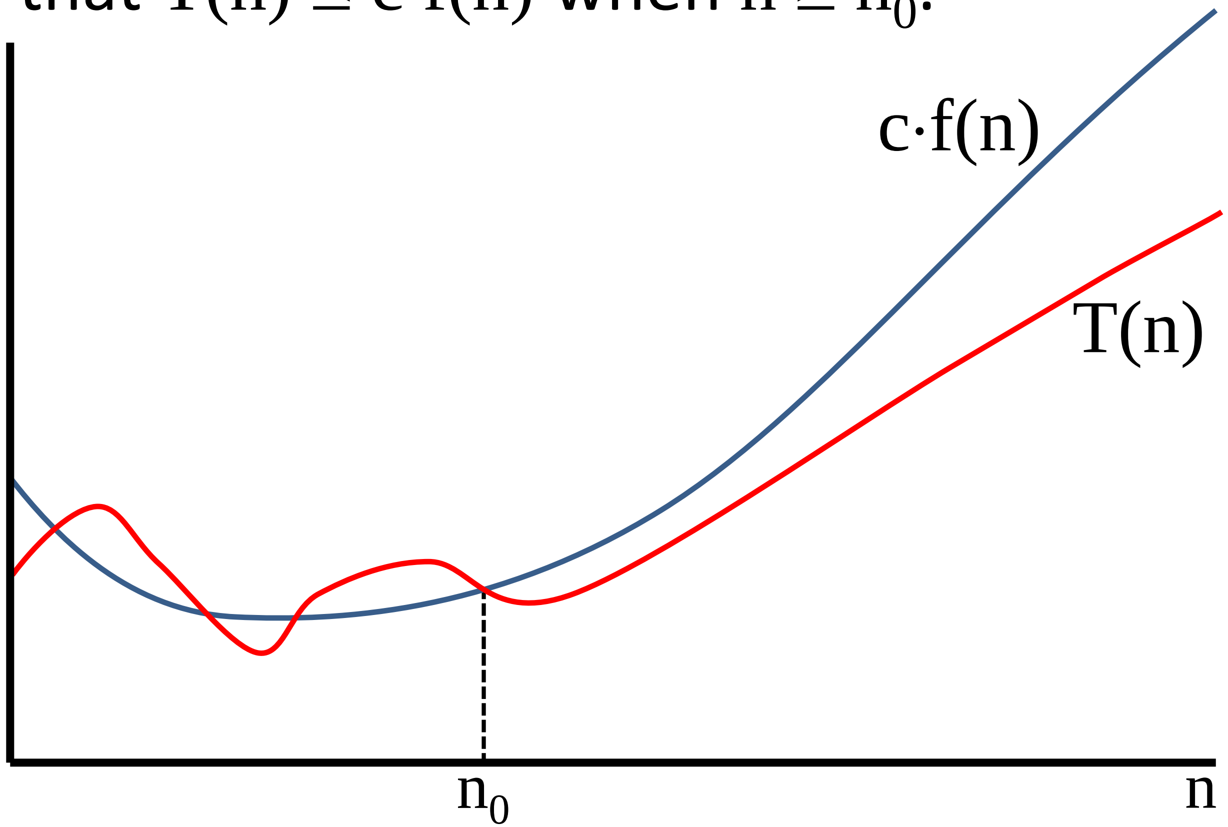
Big O

- A method to characterize the execution time of an algorithm:
  - Adding two square matrices is $O(n^2)$
  - Searching in a dictionary is $O(\log n)$
  - Sorting a vector is $O(n \log n)$
  - Solving Towers of Hanoi is $O(2^n)$
  - Multiplying two square matrices is $O(n^3)$
  - …

- The O notation only uses the dominating terms of the execution time. Constants are disregarded.

Big O: formal definition

- Let $T(n)$ be the execution time of an algorithm when the size of input data is n.
- $T(n)$ is $O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$.



Big O: example

- Let $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$
- Proof:
  - Let $c = 4$ and $n_0 = 100.05$
  - For $n \geq 100.05$, we have that $4n^2 \geq 3n^2 + 100n + 5$

- $T(n)$ is also $O(n^3)$, $O(n^4)$, etc.
  Typically, the smallest complexity is used.

Big O: examples

$T(n)$	Complexity
$5n^3 + 200n^2 + 15$	$\mathrm{O}(n^3)$
$3n^2 + 2^{300}$	$\mathrm{O}(n^2)$
$5\log_2 n + 15\ln n$	$\mathrm{O}(\log n)$
$2\log n^3$	$\mathrm{O}(\log n)$
$4n + \log n$	$\mathrm{O}(n)$
2^{64}	$\mathrm{O}(1)$
$\log n^{10} + 2\sqrt{n}$	$\mathrm{O}(\sqrt{n})$
$2^n + n^{1000}$	$\mathrm{O}(2^n)$

Complexity ranking

Function	Common name
$n!$	factorial
2^n	exponential
$n^d,\ d > 3$	polynomial
n^3	cubic
n^2	quadratic
$n\sqrt{n}$	
$n \log n$	quasi-linear
n	linear
$\sqrt{n}$	root - n
$\log n$	logarithmic
1	constant

Complexity analysis: examples

Let us assume that `f()` has complexity $O(1)$

```
for (int i = 0; i < n; ++i) f();
```

$\longrightarrow O(n)$

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) f();
```

$\longrightarrow O(n^2)$

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j <= i; ++j) f();
```

$\longrightarrow O(n^2)$

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < n; ++k) f();
```

$\longrightarrow O(n^3)$

```
for (int i = 0; i < m; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < p; ++k) f();
```

$\longrightarrow O(mnp)$

Complexity analysis: recursion

```
void f(int n) {
   if (n > 0) {
      DoSomething(n); // O(n)
      f(n/2);
   }
}
```

$$T(n) = n + T(n/2)$$

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 2 + 1$$

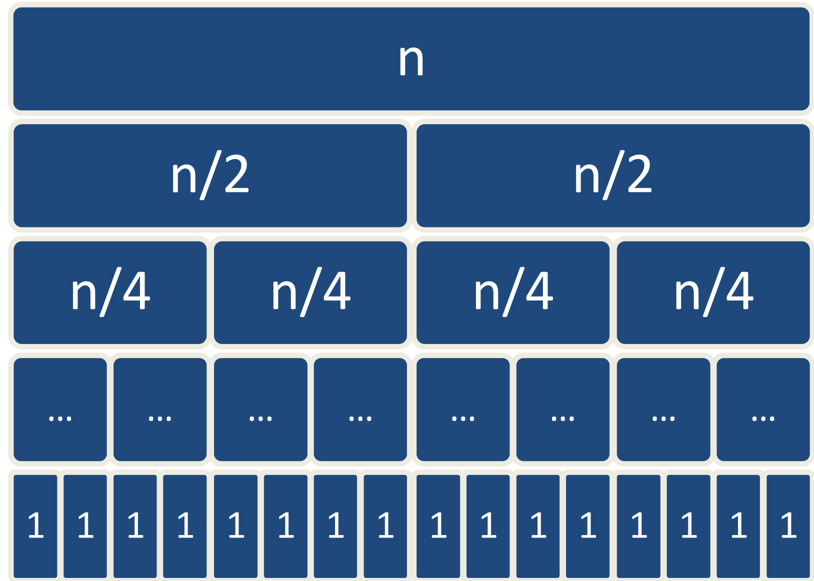
$$2 \cdot T(n) = 2n + n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2$$

$$2 \cdot T(n) - T(n) = T(n) = 2n - 1$$

$T(n)$ is $\mathrm{O}(n)$

Complexity analysis: recursion

```
void f(int n) {
   if (n > 0) {
      DoSomething(n); // O(n)
      f(n/2); f(n/2);
   }
}
```



$$
\begin{aligned}
T(n) &= n + 2 \cdot T(n/2) \\
&= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + \cdots \\
&= \underbrace{n + n + n + \cdots + n}_{\log_2 n} \;=\; n \log_2 n
\end{aligned}
$$

$$T(n) \text{ is } \mathrm{O}(n \log n)$$

Complexity analysis: recursion

```
void f(int n) {
   if (n > 0) {
      DoSomething(n); // O(n)
      f(n-1);
   }
}
```

$$
\begin{aligned}
T(n) &= n + T(n-1) \\
T(n) &= n + (n-1) + (n-2) + \cdots + 2 + 1 \\
T(n) &= \frac{n^2 + n}{2}
\end{aligned}
$$

$$T(n) \text{ is } \mathrm{O}(n^2)$$

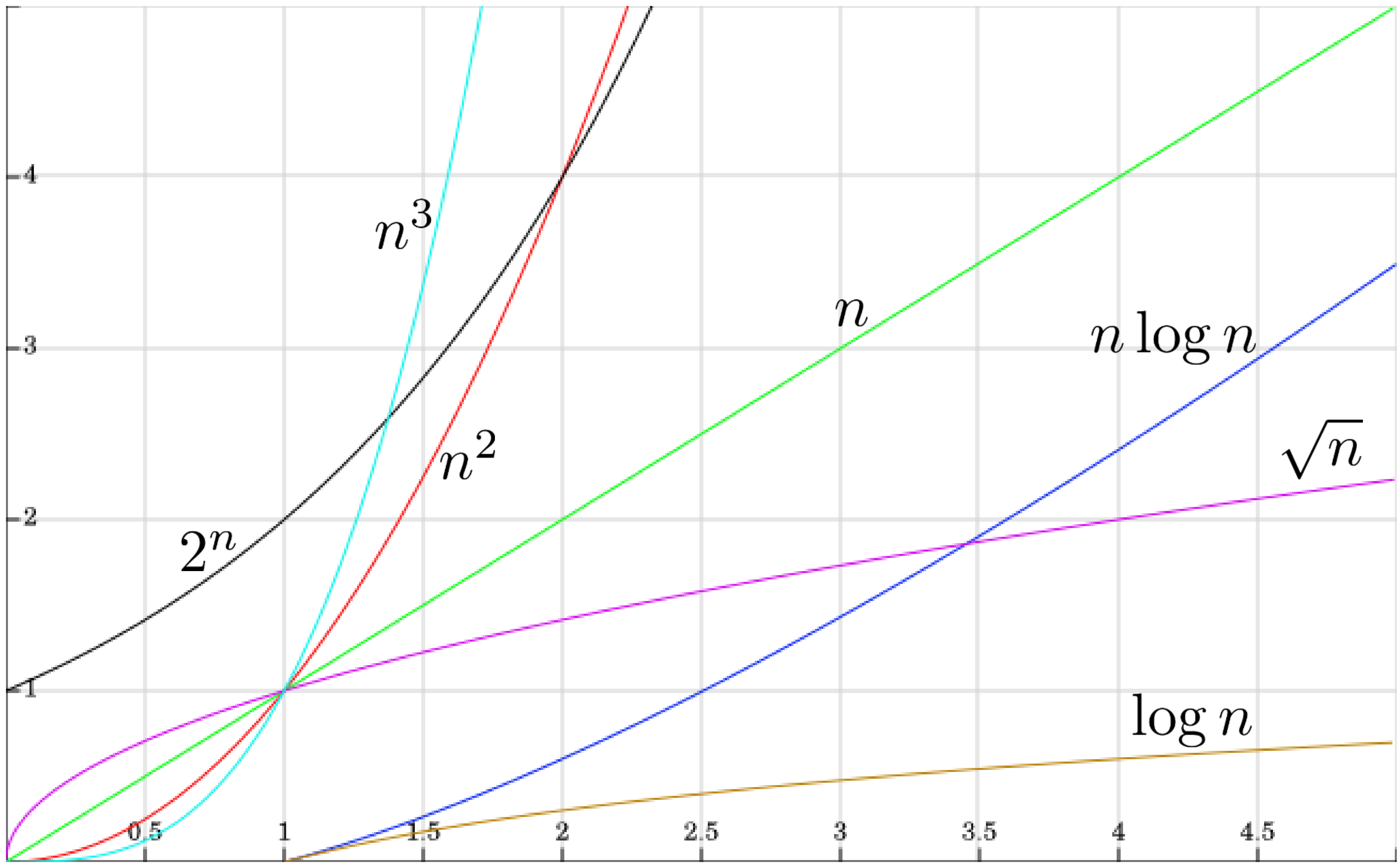
Complexity analysis: recursion

```
void f(int n) {
  if (n > 0) {
    DoSomething(); // O(1)
    f(n-1); f(n-1);
  }
}
```

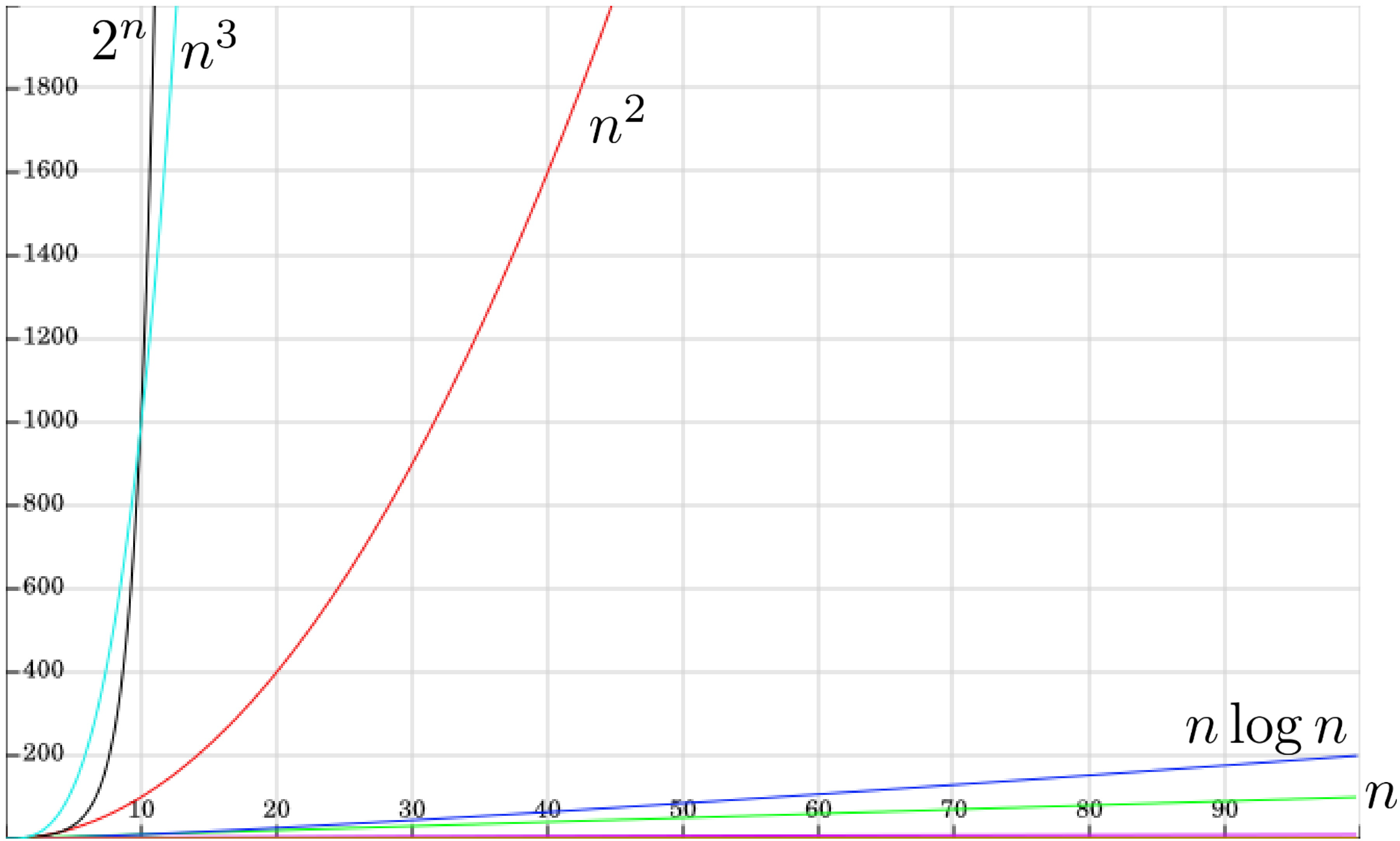
$$\begin{aligned}
T(n) &= 1 + 2 \cdot T(n-1) \\
&= 1 + 2 + 4 \cdot T(n-2) \\
&= 1 + 2 + 4 + 8 \cdot T(n-3) \\
&\vdots \\
&= 1 + 2 + 4 + 8 + \cdots + 2^{n-1} \\
&= \sum_{i=0}^{n-1} 2^i = 2^n - 1
\end{aligned}$$

$T(n)$ is $\mathrm{O}(2^n)$

Asymptotic complexity (small values)



Asymptotic complexity (larger values)



Execution time: example

Let us consider that every operation can be executed in 1 ns (10^{-9} s).

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

How about “big data”?

Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

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

This is often the practical limit for big data

Summary

- Complexity analysis is a technique to analyze and compare algorithms (not programs).
- It helps to have preliminary back-of-the-envelope estimations of runtime (milliseconds, seconds, minutes, days, years?).
- Worst-case analysis is sometimes overly pessimistic. Average case is also interesting (not covered in this course).
- In many application domains (e.g., big data) quadratic complexity, $O(n^2)$, is not acceptable.
- Recommendation: avoid last-minute surprises by doing complexity analysis before writing code.