COL100: Introduction to Computer Science

6.2: Big O notation

Motivation

An algorithm that takes $n!$ time is much worse than one that takes n^3 time…
or even $10^6\, n^3$ time.

We want a way to express the complexity of algorithms…

1. for very large problems
   - Will it take seconds? days? years?
2. while ignoring irrelevant factors and extra terms
   - $1000\, n^3$ and $n^3 + 1000$ are both “similar” to n^3 and “better” than 2^n

Big O notation

Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say

$$f(n) \text{ has order of growth } O(g(n))$$

$$\text{or, } f(n) = O(g(n))$$

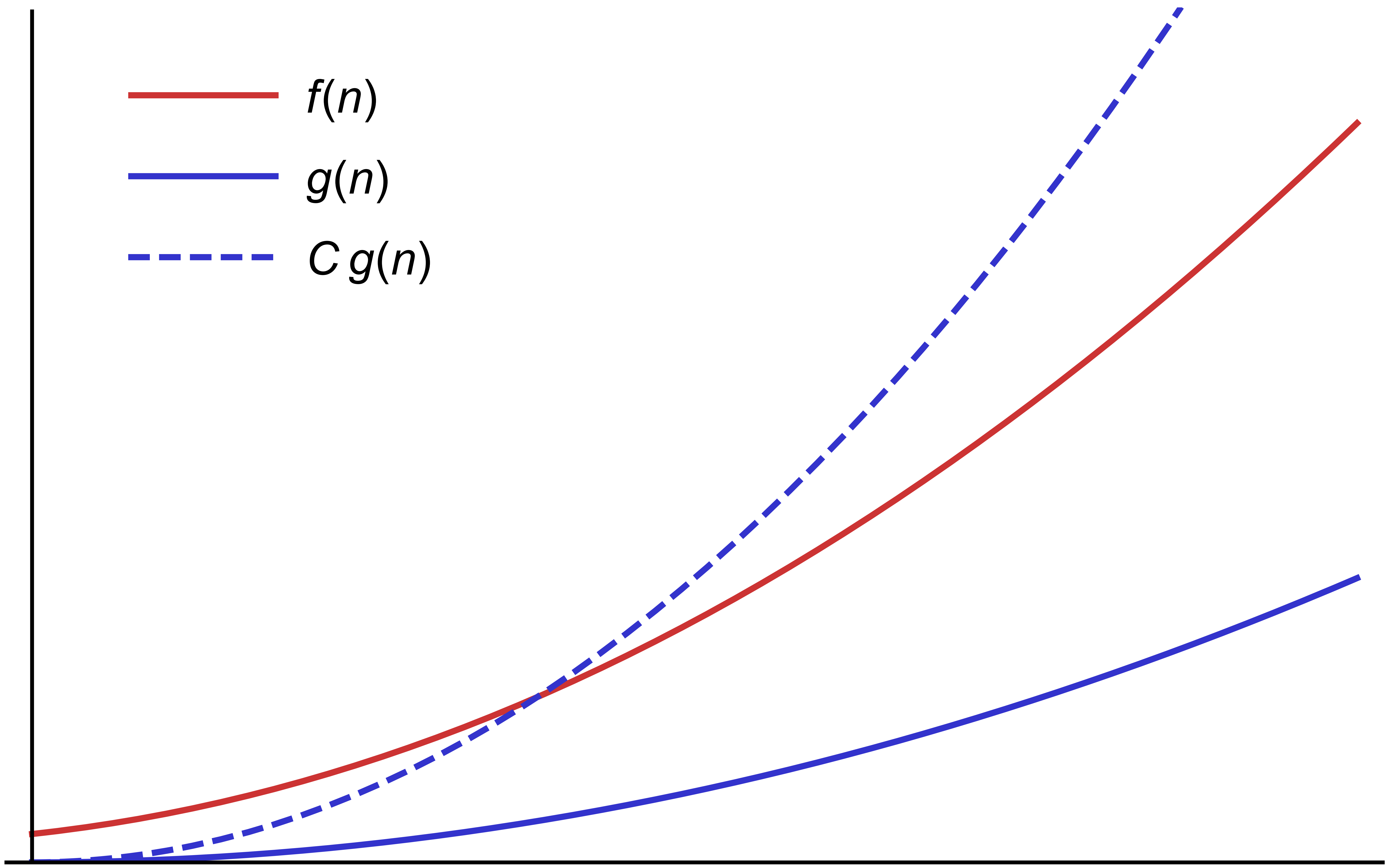
Should have been
f(n) ∈ O(g(n))…

if there exist constants C, n_0 such that $f(n) \leq Cg(n)$ for all $n \geq n_0$.

In other words, I can scale up $g(n)$ so that it is eventually always bigger than $f(n)$.

- This describes the *asymptotic* rate of growth of $f(n)$, i.e. as $n \to \infty$.

$f(n) = O(g(n))$ if there exist constants C, n_0 such that $f(n) \leq Cg(n)$ for all $n \geq n_0$.



$f(n) = O(g(n))$ if there exist constants C, n_0 such that $f(n) \leq Cg(n)$ for all $n \geq n_0$.

Examples:

$$an + b = O(n) \text{ for all } a, b.$$

$$\begin{gathered} n = O(n^2), \\ n = O(2^n), \\ n \neq O(\log n), \\ n \neq O(1). \end{gathered}$$

- Big O only gives an *upper bound* on the asymptotic growth of a function.
- What would it mean for an algorithm to have $O(1)$ time complexity?

Examples

- *power*(x, n) has time complexity $n = O(n)$ and space complexity $n + 1 = O(n)$.
- *fastPower*(x, n) has time complexity $2\lceil \log_2 n \rceil + c = O(\log n)$.
- Naive determinant computation has time complexity $O((n + 1)!)$.
- Gaussian elimination has time complexity $\leq n^3 + 2n^2 = O(n^3)$.

Properties of big O

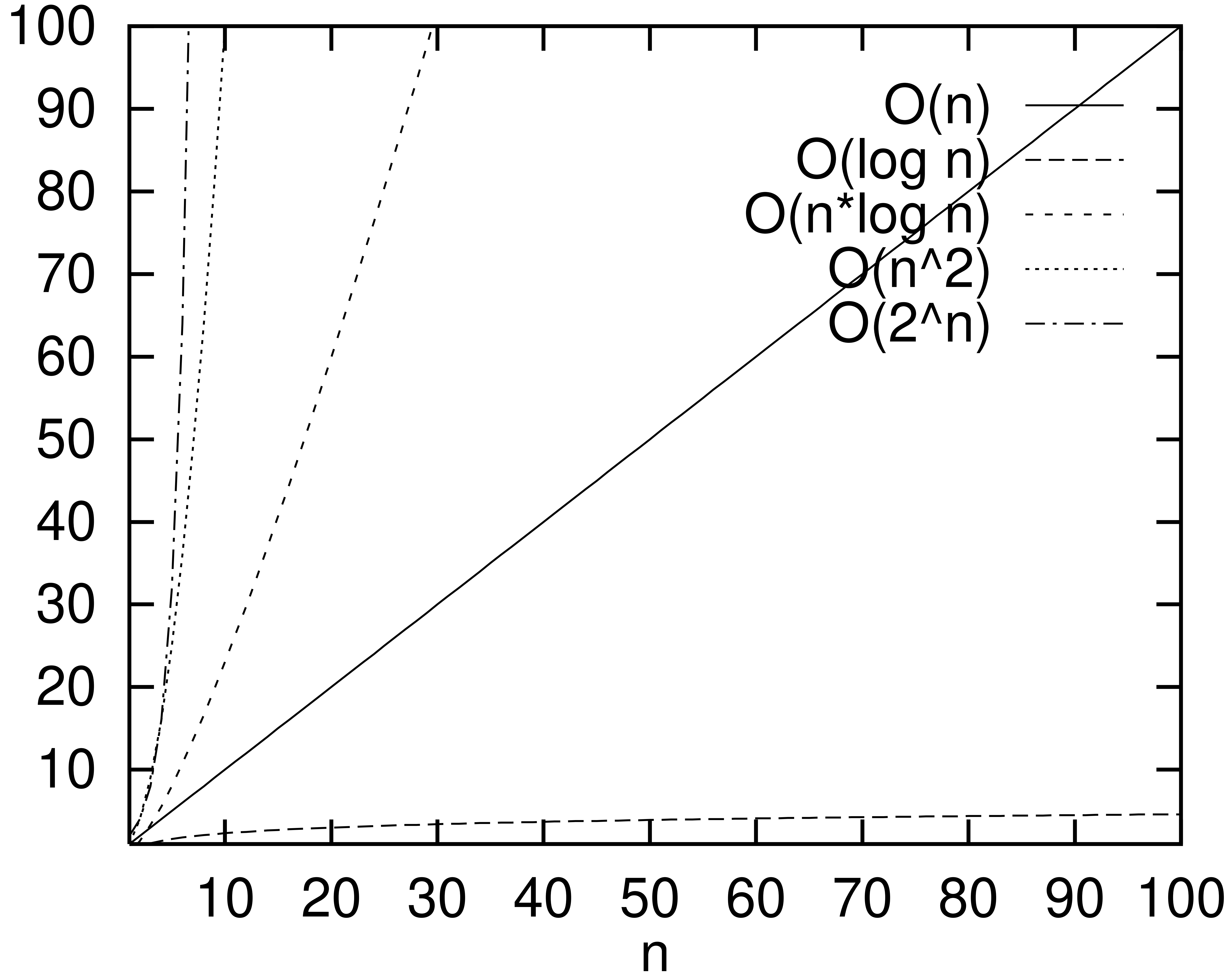
- If $f(n) = O(g(n))$ then $af(n) = O(g(n))$ for any constant $a > 0$.
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.
- If $f(n) = O(h(n))$ and $g(n) = O(k(n))$ then
  - $f(n) + g(n) = O(h(n) + k(n))$,
  - $f(n)\, g(n) = O(h(n)\, k(n))$.

For any polynomial of degree d with leading coefficient $a_d > 0$,

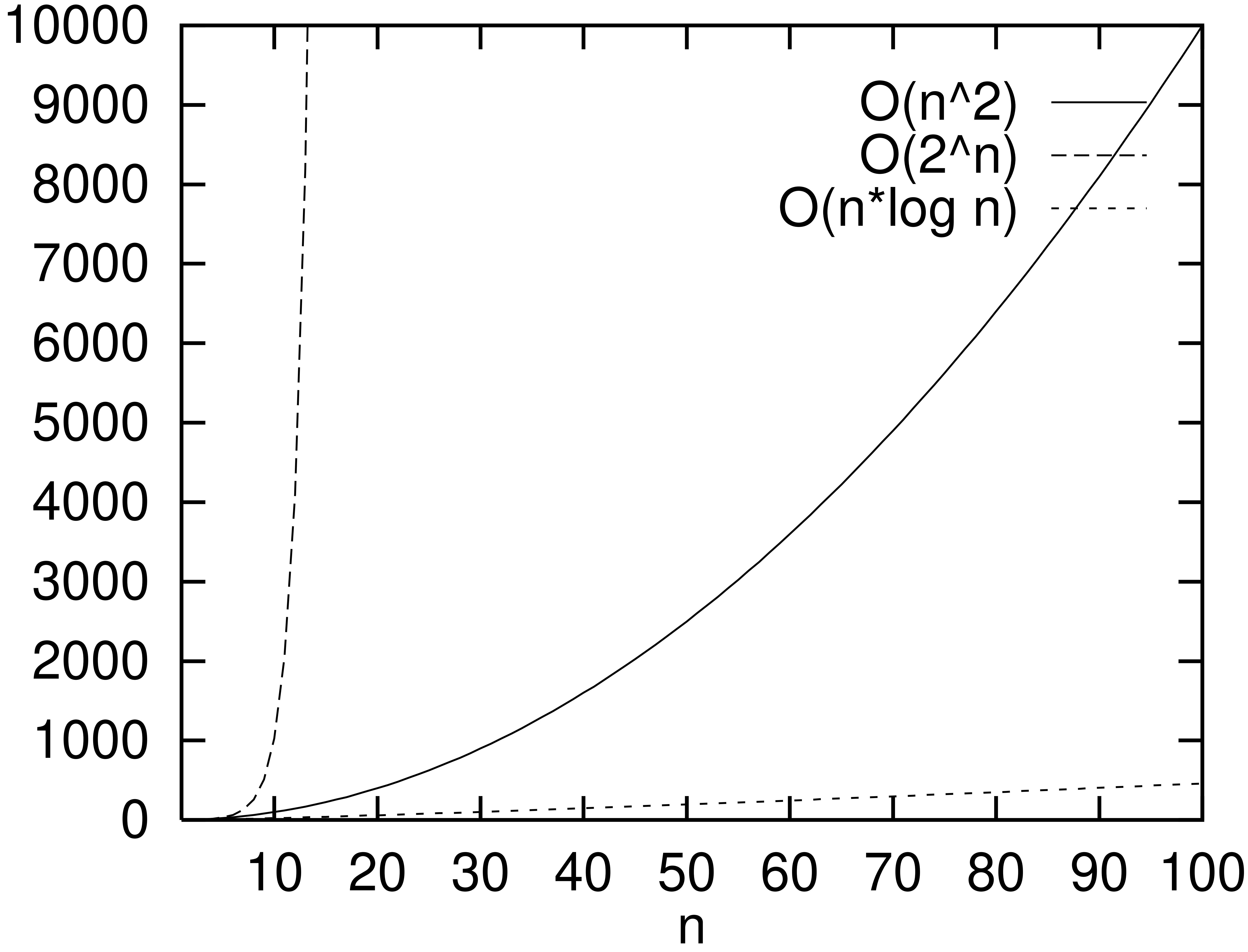
$$a_d\, n^d + \cdots + a_1\, n + a_0 = O(n^d).$$

For any exponential with $a > 1$,

$$a^n \neq O(n^d),$$
$$n^d = O(a^n)$$



100
90
80
70
60
50
40
30
20
10
10
20
30
40
50
60
70
80
90
100
n
O(n)
O(log n)
O(n*log n)
O(n^2)
O(2^n)


10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100
n
O(n^2)
O(2^n)
O(n*log n)

Why big O notation?

Big O is quite a crude analysis, but:

- That makes it easier to do! Ignore constants, lower-order terms

$$cn^3 + O(n^2) = O(n^3)$$

Why big O notation?

Big O is quite a crude analysis, but:

- That makes it easier to do! Ignore constants, lower-order terms
- It gives a good idea of how an algorithm *scales* for large n
  - If $T(n) = O(n^2)$, doubling n will at most quadruple number of steps
  - If $T(n) = \mathrm{O}(2^n)$, increasing n by just 2 could quadruple number of steps!

Why big O notation?

Big O is quite a crude analysis, but:

- That makes it easier to do! Ignore constants, lower-order terms
- It gives a good idea of how an algorithm *scales* for large n
- It makes it easy to compare algorithms for large n
    - If $T_1(n) = O(T_2(n))$ but $T_2(n) \neq O(T_1(n))$, algorithm 1 will always be faster than algorithm 2 on large enough input — even if run on a slower machine!

Afterwards

- Read Sections 3.6.4 and 3.7 of the notes.
- ~~Complete the proofs for the time and space complexity of *fib*(n).~~
- Evaluate the time and space complexity of *isPrime*(n) as defined in the previous lecture, assuming that computing mod takes $O(1)$ time. Design an algorithm for *isPrime*(n) that has $O(\sqrt{n})$ time complexity.