Introduction to Algorithms

Main Objective: Design Efficient Algorithms that finds optimum solutions in the Worst Case

Defining Efficiency

“Runs fast on typical real problem instances”

Pros:

- Sensible,
- Bottom-line oriented

Cons:

- Moving target (diff computers, programming languages)
- Highly subjective (how fast is “fast”? What is “typical”?)

Measuring Efficiency

Time ≈ # of instructions executed in a simple programming language

- only simple operations (+,*,-,=,if,call,…)
- each operation takes one time step
- each memory access takes one time step
- no fancy stuff (add these two matrices, copy this long string,…) built in; write it/charge for it as above

Time Complexity

Problem: An algorithm can hav
different inputs

Solution: The complexity of an
number **T(N)**, the “time” the al
size **N**.

On which

一个代表算法输入值的字符串的长度的函数。时间复杂度常用大O符号表述，不包括这个函数的低阶项和首项系数。使用这种方式时，时间复杂度可被称为是渐近的（渐近时间复杂度），亦即考察输入值大小趋近无穷时的情况。

当n很大时，你可以把它想象成10000、100000。而公式中的低阶、常量、系数三部分并不左右增长趋势，所以都可以忽略。我们只需要记录一个最大量级就可以了。

Mathematically,

T is a function that maps positive integers giving problem size to positive integers giving number of steps

Time Complexity (N)

Worst Case Complexity: max # steps algorithm takes on any input of size **N**

This Couse

Average Case Complexity: avg # steps algorithm takes on inputs of size **N**

Best Case Complexity: min # steps algorithm takes on any input of size **N**

Why Worst-case Inputs?

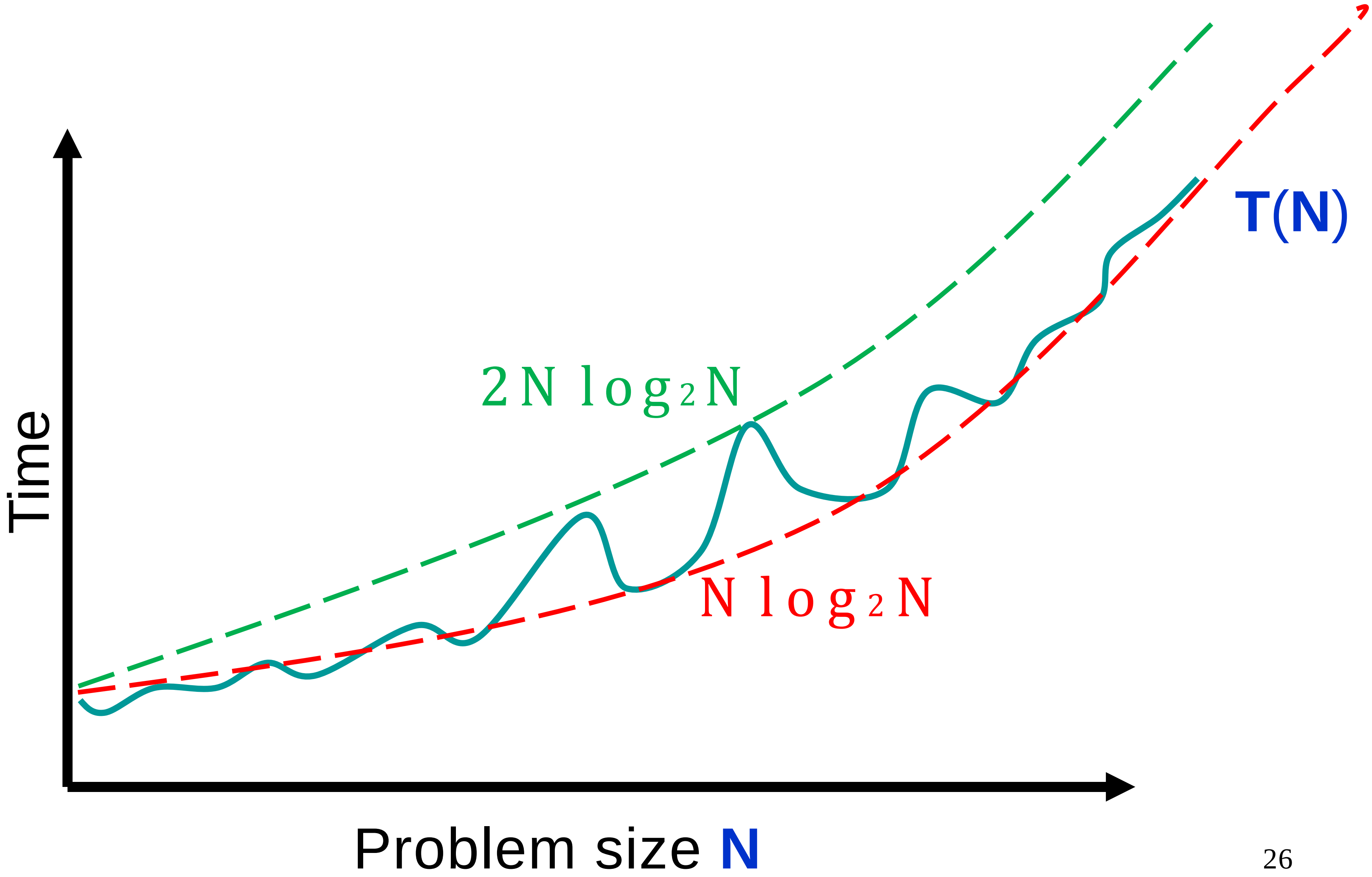
- Analysis is typically easier
- Useful in real-time applications

  e.g., space shuttle, nuclear reactors)
- Worst-case instances kick in when an algorithm is run as a module many times

  e.g., geometry or linear algebra library
- Useful when running competitions

  e.g., airline prices
- Unlike average-case no debate about the right definition

Time Complexity on Worst Case Inputs



O-Notation

Given two positive functions **f** and **g**

- **f(N)** is **O(g(N))** iff there is a constant **c**$>$**0** s.t.,

  f(N) is eventually always $\leq$ **c g(N)**

- **f(N)** is Ω**(g(N))** iff there is a constant $\varepsilon$$>$**0** s.t.,

  f(N) is $\geq$ ε **g(N)** for infinitely

- **f(N)** is Θ**(g(N))** iff there are constants c_1, c_2>0 so that

  eventually always $c_1 g(N) \leq f(N) \leq c_2 g(N)$

Asymptotic Bounds for common fns

- Polynomials:

  $a_0 + a_1 \mathrm{n} + \cdots + a_d n^d$ is $\mathrm{O}(n^d)$

- Logarithms:

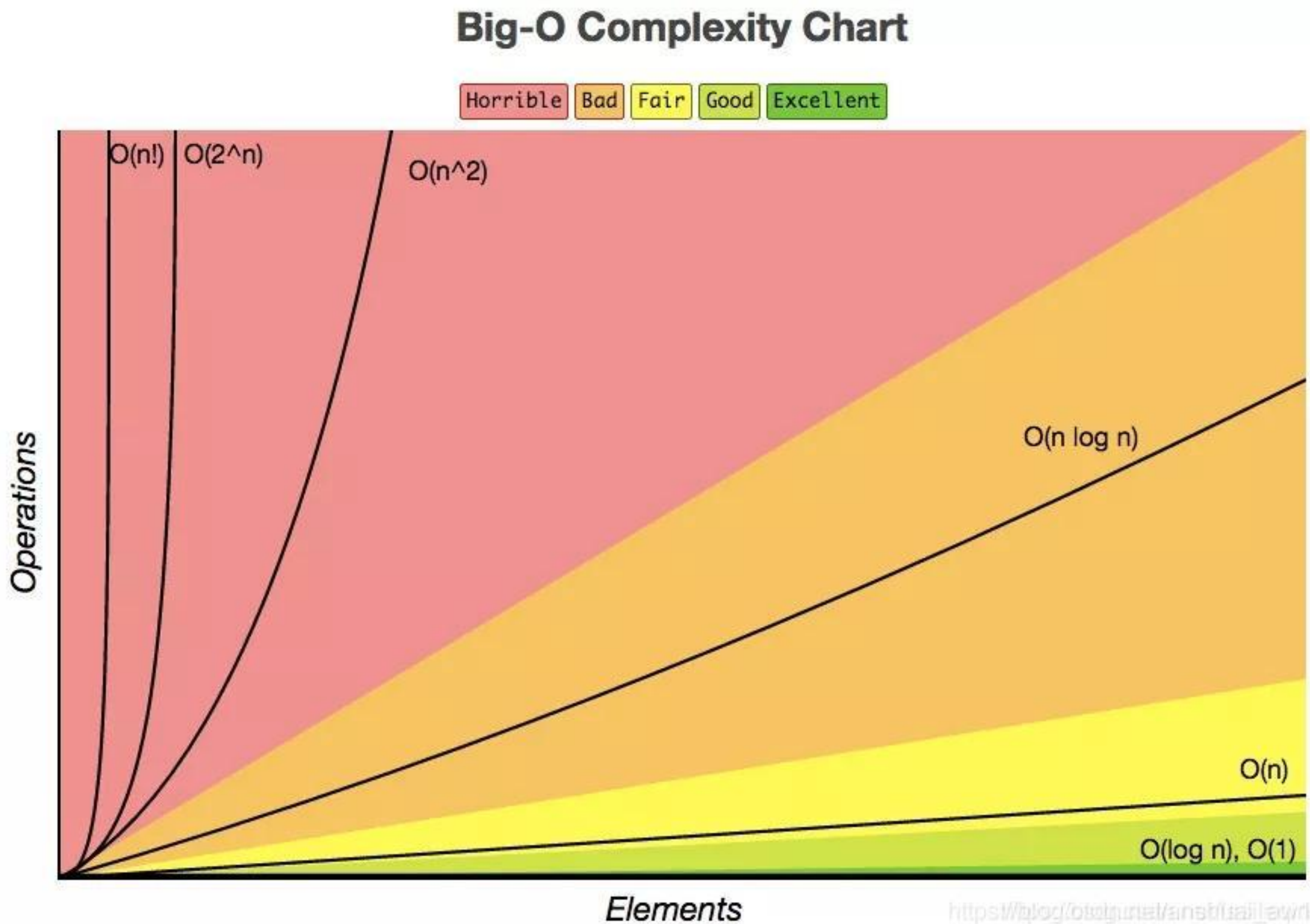
  $\log_a n = O(\log_b n)$ for all constants $a, b > 0$

- Logarithms: log grows slower than every polynomial

  For all $x > 0$, $\log n = O(n^k)$

- $n\ log\ n = O(\mathrm{n}^{1.01})$

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$



Efficient = Polynomial Time

An algorithm runs in polynomial time if T(n)=O(n^d) for some constant d independent of the input size n.

Why Polynomial time?

If problem size grows by at most a constant factor then so does the running time

- E.g. $\mathbf{T(2N) \leq c(2N)^k \leq 2^k(cN^k)}$
- Polynomial-time is exactly the set of running times that have this property

Typical running times are small degree polynomials, mostly less than $\mathbf{N^3}$, at worst $\mathbf{N^6}$, not $\mathbf{N^{100}}$

Why it matters?

- #atoms in universe $<2^{240}$
- Life of the universe $< 2^{54}$ seconds
- A CPU does $< 2^{30}$ operations a second

If every atom is a CPU, a 2^n time ALG cannot solve n=350 if we start at Big-Bang.

	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

not only get very big, but do so *abruptly*, which likely yields erratic performance on small instances

Why “Polynomial”?

Point is not that n^{2000} is a practical bound, or that the differences among n and 2n and n^2 are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

- “My problem is in P” is a starting point for a more detailed analysis
- “My problem is not in P” may suggest that you need to shift to a more tractable variant