Extended Introduction to Computer Science CS1001.py

Chapter C
Lecture 8b Complexity and the $O(\cdot)$ Notation

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

Time Complexity: Basic Notions

- A computational problem is a relation between input and its corresponding output (or mathematically, function parameters and function value)

- An algorithm is a step-by-step procedure, a “recipe”
  - can be represented in pseudo-code, diagrams, animations, etc.
  - an abstract notion, can be implemented as a computer program

- Efficient algorithms are normally preferred
  - fastest – time complexity
  - most economical in terms of memory – space/memory complexity

- Time complexity analysis:
  - measured in terms of operations, not actual time
    - We want to say something about the algorithm, not a specific machine/execution/programming language implementation
  - but can be accompanied by actual time measurements
  - expressed as a function of the problem input size
  - often distinguish best/worst case inputs

Comments on Time complexity Analysis

- So far we analyzed time efficiency in terms of the number of iterations, rather than counting operations.

- What underlying assumption justified this?

- An underlying assumption: the number of operations in each iteration is bounded by some constant.
  - Note that by “operations” we refer to basic ones, such as reading a variable from memory, comparing two computer words, etc.
  - Such operations may require different amount of time on different machines / operating systems or even different executions on the same computer

- Pay attention! This assumption does not always hold (examples?)

Defining Time Complexity

- We will be interested in how the number of operations changes with input size.

- In most cases, we will not care about the exact function, but in its “order”, or growth rate (e.g., logarithmic, linear, quadratic, etc.)

- Sometimes we will only be interested/able to give an upper bound for this growth rate. We will, however, strive to make this upper bound as tight (=low) as we can.
  - In this course, we will almost always be able to give tight upper bounds.

- So we need some formal definition for ”upper bound for the growth rate of the number of operations, as a function of input size”.

"Big O" Notation

- Let $f(n)$ denote the number of operations an algorithm performs on an input of size n.

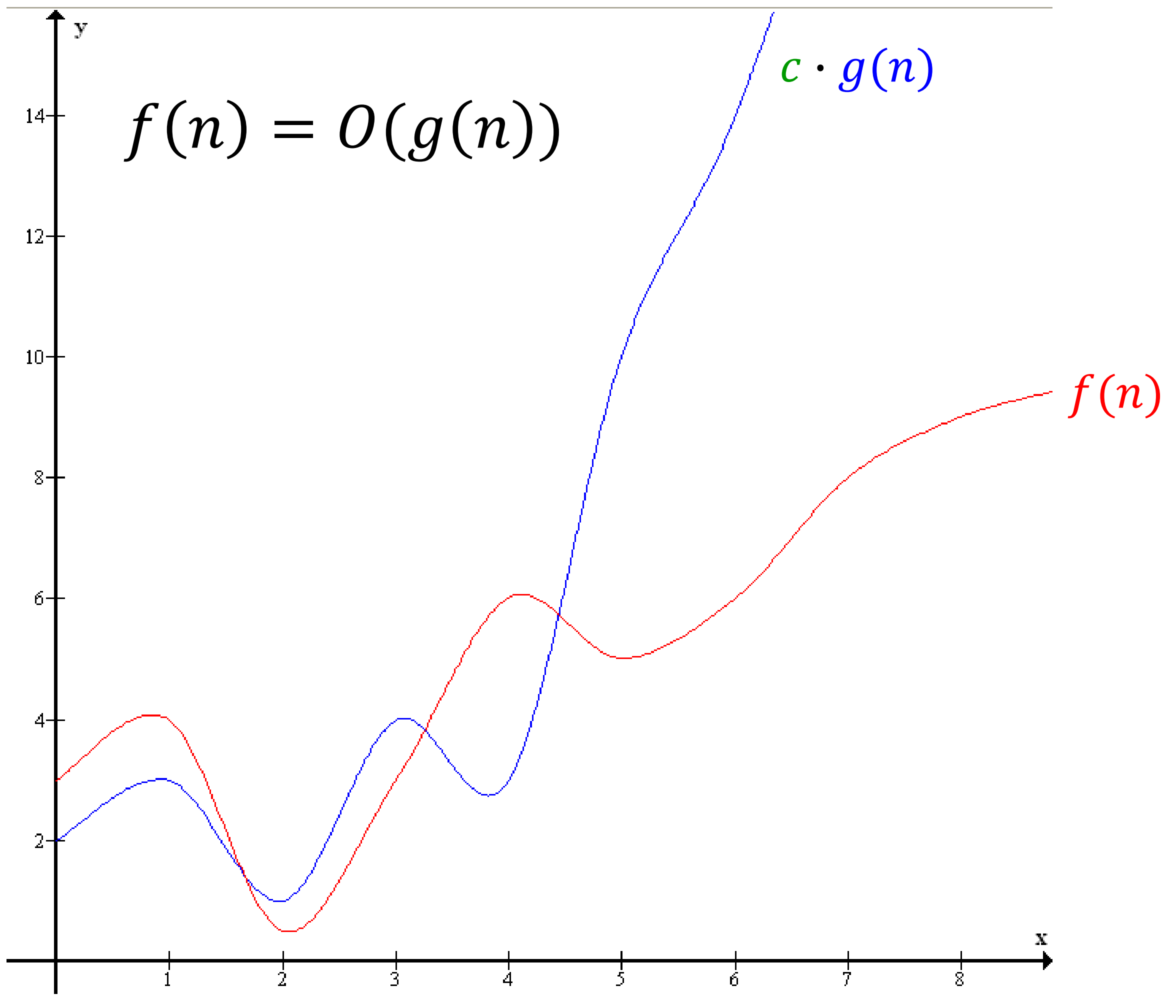
- We say that $f(n)$ *belongs to* $O(g(n))$ if there exists a constant c such that for large enough n,

$$f(n) \leq c \cdot g(n)$$

- This is denoted by $f(n) \in O(g(n))$
- Also commonly denoted by $f(n) = O(g(n))$
  - $=$ is abused and does not mean equality

- Alternatively, $f(n)$ may denote the number of memory cells required by the algorithm on an input of size n

Big O Notation – Visualized



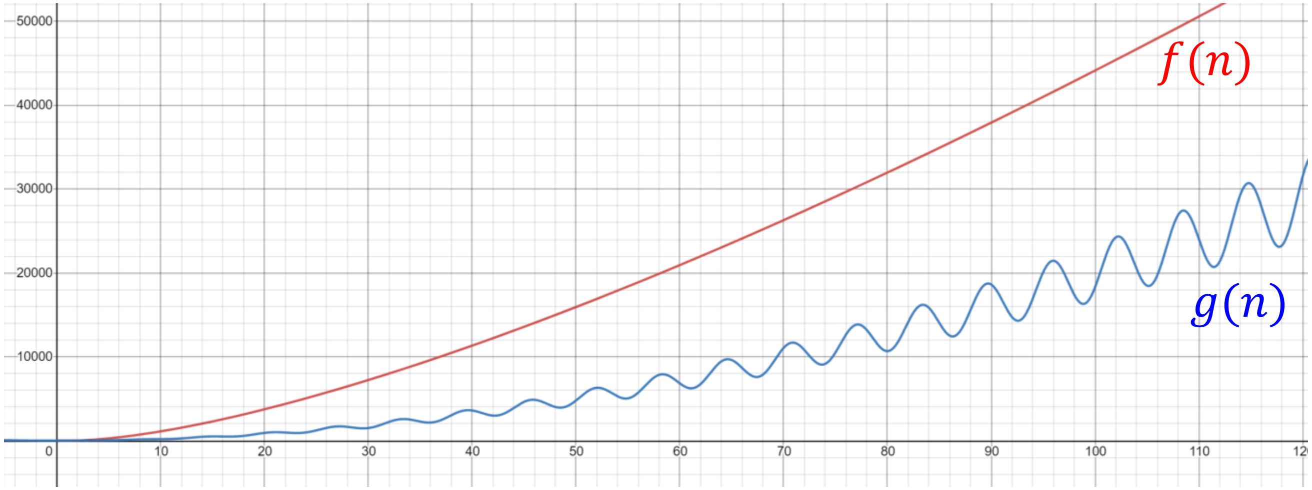
Big O Notation - Examples

- $3n + 7 = O(n)$
- $3n + 7 = O(n^2)$ *
- $3n + 7 \neq O(\sqrt{n})$
- $5n \cdot \log_2 n + 1 = O(n \log n)$ [where did the log base disappear?]
- $6\log_2 n = O(n)$ *
- $2\log_2 n + 12 = O(n)$ *
- $1000 \cdot n \cdot \log_2 n = O(n^2)$ *
- $3^n \neq O(2^n)$
- $2^{n/100} \neq O(n^{100})$

* not the tightest possible bound

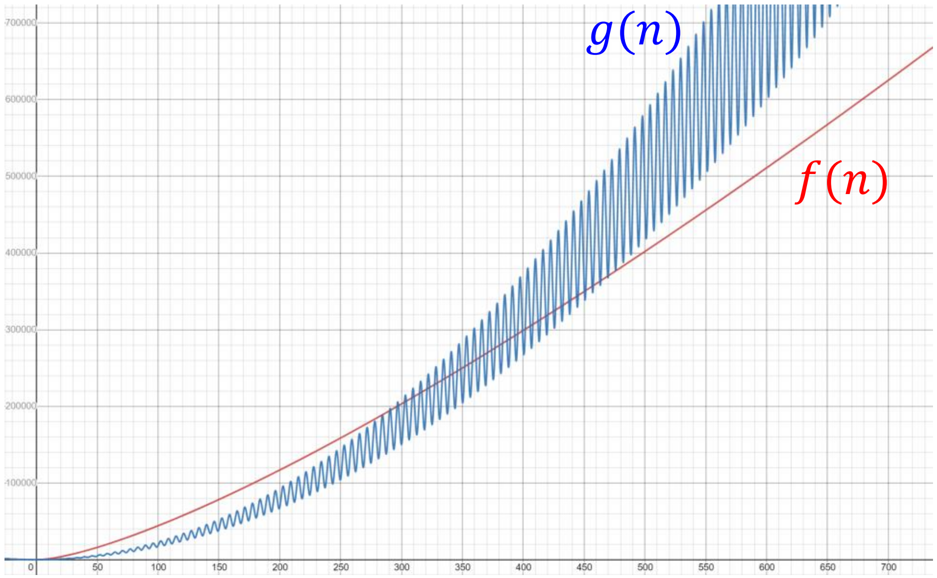
The Asymptotic Nature of Big O

- Consider the two functions $f(n) = 10n\log^2 n + 1$, and
  $g(n) = n^2 \cdot (2 + \sin(n)/3) + 2$
- It is not hard to verify that $f(n) = O(g(n))$.
- Yet, for small values of n, $f(n) > g(n)$, as can be seen in the following plot:



The Asymptotic Nature of Big O (cont.)

- But for large enough n, indeed $f(n) \leq 1 \cdot g(n)$, as can be seen in the next plot:



- Also, remember that for big O, $f(n)$ may be larger than $g(n)$, as long as there is a constant c such that $f(n) \leq c \cdot g(n)$.

Summary of Some Previous Results

- All these results refer to worst case scenarios.

- Algorithms we saw on sequences:
  - Palindrome checking on a string of length n takes $O(n)$ iterations
  - Binary search on a sorted list of length n takes $O(logn)$ iterations
  - Selection Sort on a list of length n takes $O(n^2)$ iterations
  - Merging 2 sorted lists of sizes n and m takes $O(n+m)$ iterations

- Algorithms we saw on integers:
  - Addition of two n-bit integers takes $O(n)$ iterations
  - Multiplication of two n-bit integers takes $O(n^2)$ iterations

Input Size - Clarifications

- We measure complexity as a function of the input size.
- For integers, input size is the number of bits in the representation of the number in the computer.
  - we normally count the number of "simple" bit operations (such as adding or multiplying two bits).
- For lists/strings/dictionaries/other collections, the input size is typically the number of elements in the collection.
  - We normally consider "simple" operations on these elements (such as comparisons, assignments) to take a constant amount of time.
  - There are exceptions to this, however (see example on the next slide).

Input Size – Clarifications (cont.)

- Recall that Selection Sort on a list of n elements runs in $O(n^2)$ time.
- But what if the elements in the list are strings, each of size m?
- Comparing 2 such strings (in each iteration of Selection Sort) takes $O(m)$ in the worst case.
- Overall, Selection Sort will run in $O(n^2 \cdot m)$ time.

Worst / Best Case Complexity

- In many cases, for the same size of input, the content of the input itself affects the complexity. We then separate between worst case and best case complexity.

$$T_{worst}(n) = \max\{time(Input): |Input| = n\}$$

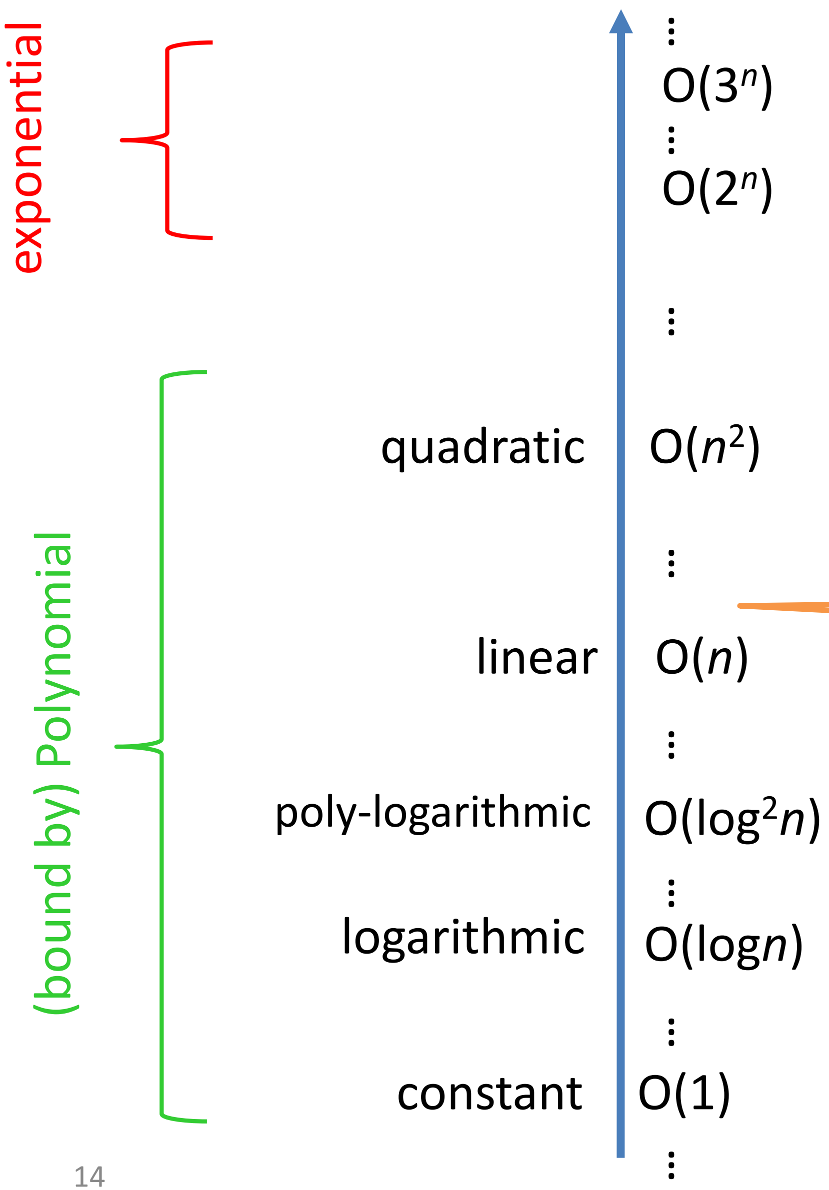
$$T_{best}(n) = \min\{time(Input): |Input| = n\}$$

- Examples:

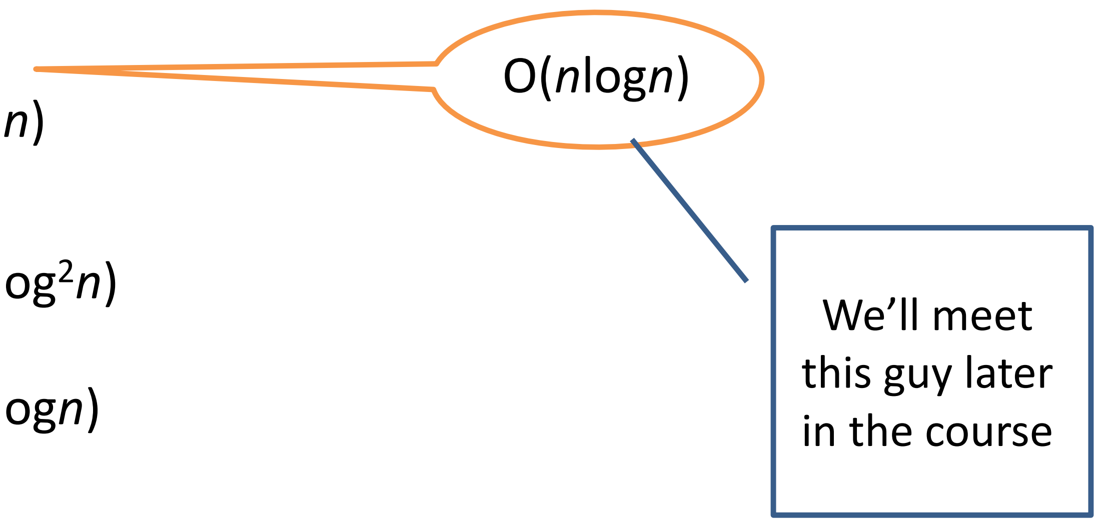
	Best case	Worst case
Binary search	O(1)	O(logn)
Selection sort	$O(n^2)$	$O(n^2)$

- Note that this statement is completely nonsense:
  "The best time complexity is when n is very small…"

Complexity Hierarchy



Unless asked to prove formally, You can use this hierarchical orderings as facts.



O(1)

What is the meaning of this, in terms of time complexity?

a) A very short running time
b) A running time that is independent of the input size (i.e. constant)
c) 1 operation
d) Termination due to Run-time error

(In)Tractability

- How would execution time for a fast, modern processor (10^{10} ops per second, say) vary for a task with the following time complexities and n = input sizes?

	10	20	30	40	50	60
n	1.0E-09 seconds	2.0E-09 seconds	3.0E-09 seconds	4.0E-09 seconds	5.0E-09 seconds	6.0E-09 seconds
n^2	1.0E-08 seconds	4.0E-08 seconds	9.0E-08 seconds	1.6E-07 seconds	2.5E-07 seconds	3.6E-07 seconds
n^3	1.0E-07 seconds	8.0E-07 seconds	2.7E-06 seconds	6.4E-06 seconds	1.3E-05 seconds	2.2E-05 seconds
n^5	1.0E-05 seconds	0.00032 seconds	0.00243 seconds	0.01024 seconds	0.03125 seconds	0.07776 seconds
2^n	1.02E-07 seconds	1.05E-04 seconds	0.107 seconds	1.833 minutes	1.303 days	0.64 years
3^n	5.9E-06 seconds	0.35 seconds	5.72 hours	38.55 years	22764 centuries	1.34E+09 centuries

Modified from Garey and Johnson's classical book

- Polynomial time = tractable. Exponential time = intractable.

What is Tractable in Practice?

- A polynomial-time algorithm is good.
  - n^{100} is polynomial, hence good…
- An exponential-time algorithm is bad.
  - $2^{n/100}$ is exponential, hence bad…

- Yet for input of size $n = 4000$, the n^{100} time algorithm takes more than 10^{35} centuries on the above mentioned machine, while the $2^{n/100}$ algorithm runs in just under two minutes.

Time Complexity - Advice

- Trust, but check! Don't just mumble "polynomial-time algorithms are good", "exponential-time algorithms are bad" because the lecturer told you so.

- Asymptotic run time and the O notation are important, and in most cases help clarify and simplify the analysis.

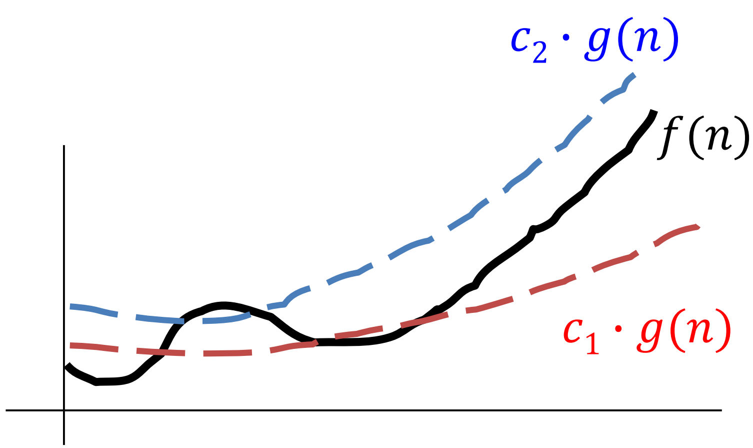
- But when faced with a concrete task on a specific problem size, you may be far away from "the asymptotic".

- In addition, constants hidden in the O notation may have unexpected impact on actual running time.

Tight Bound - Theta Θ

- We say that a function $f(n)$ is $\Theta(g(n))$ if there are two constant c_1,c_2 such that for large enough n,

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



- $f(n) = \Theta(g(n))$ IFF $f(n) = O(g(n))$ and $g(n) = O(f(n))$

- It is very common to use O instead of Θ, but formally O is merely an upper bound