(03) Complexity Part 2a (Big ‘O’)

Video (18 mins):

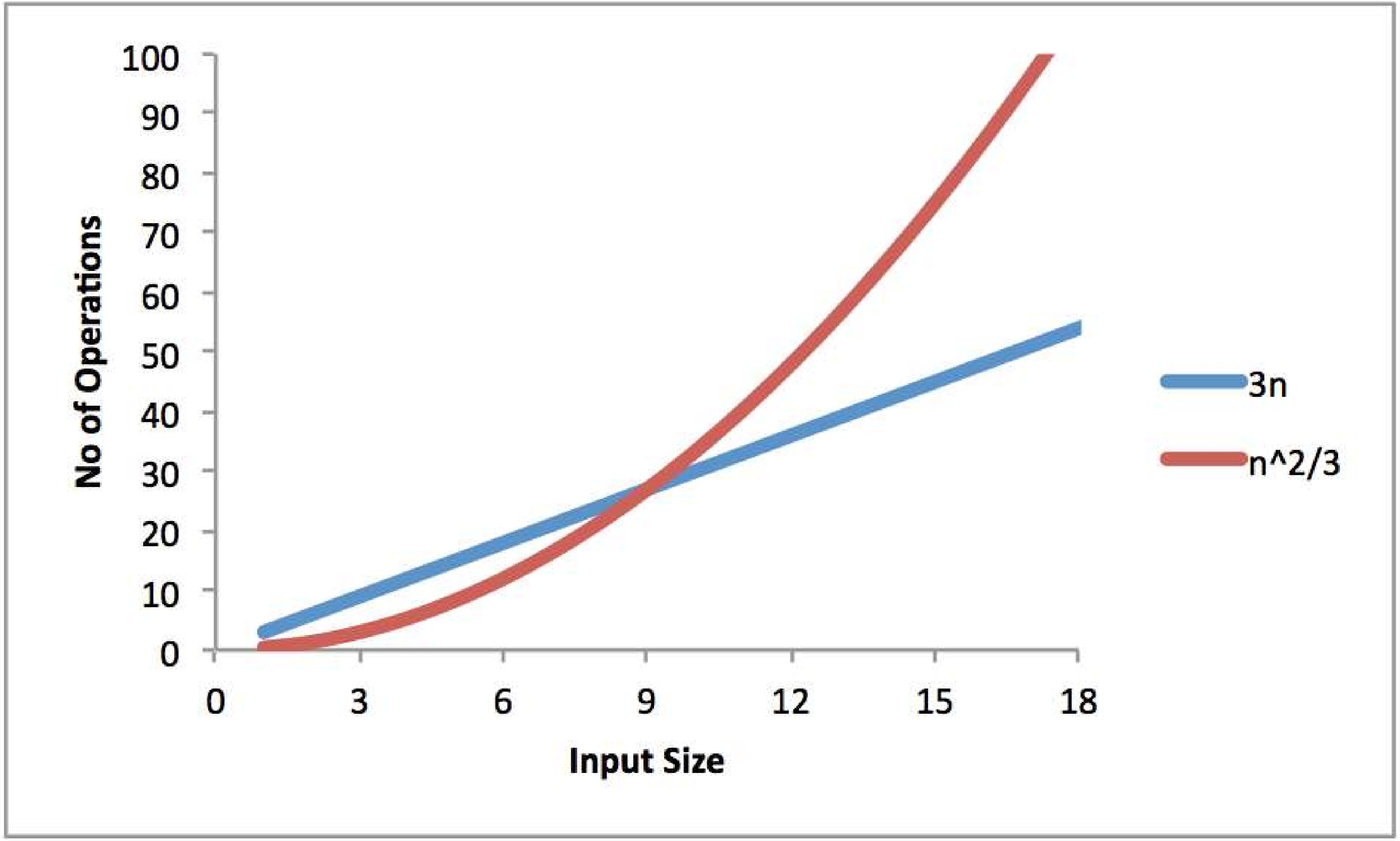
https://www.youtube.com/watch?v=QYf5fgISAbY&list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&index=22

Efficiency is a matter of the growth rate

- Growth rate:
  - how the number of operations grows as the input size increases

A more efficient algorithm has a slower growth rate in running time as the input size increases.

Why do we care about growth rate?



Remember "The Hare vs. The Tortoise" Story?

Asymptotic Order of Growth

- Asymptotic analysis is about describing the behavior of mathematical functions "in the limit"
  - we want to know how the function behaves as the input gets larger and larger without bound, towards infinity
- Why "in the limit"?
  - Small input sizes have fast running times and cause no issue
  - We are usually concerned with the worst-case complexity
- Why worst case, and not best case or average case?
  - Best case is often a "special" situation that does not apply to most inputs
  - Average case is difficult to determine without knowledge of the real world frequencies of input occurrences
  - Worst case is a good predictor of "difficulty" of problems

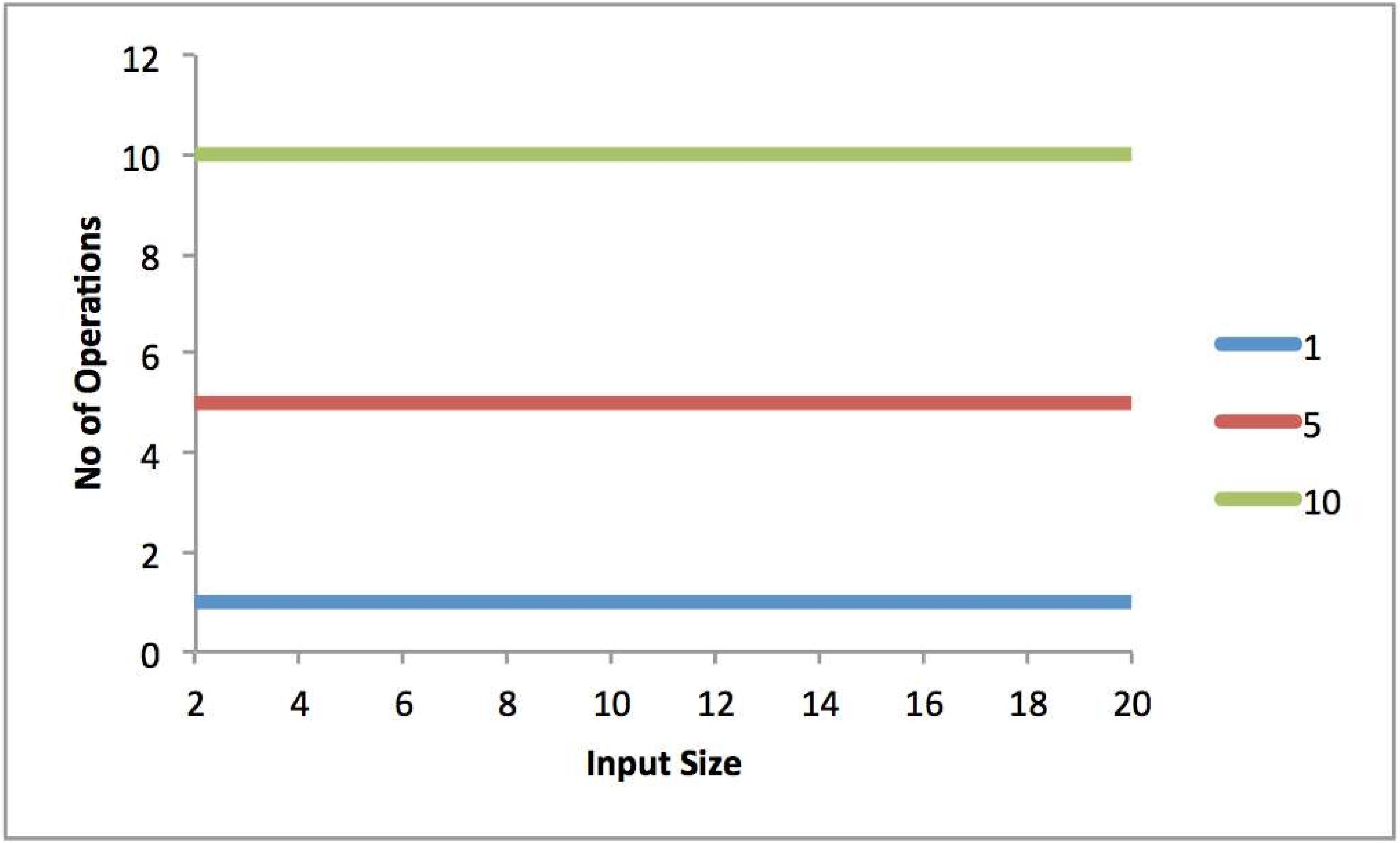
Big O Notation

- Capital letter O to specify an algorithm's order of complexity
  - $O(n^2)$ pronounced "oh of n-squared" or "big oh of n-squared"
  - represents the concept of "upper bound"

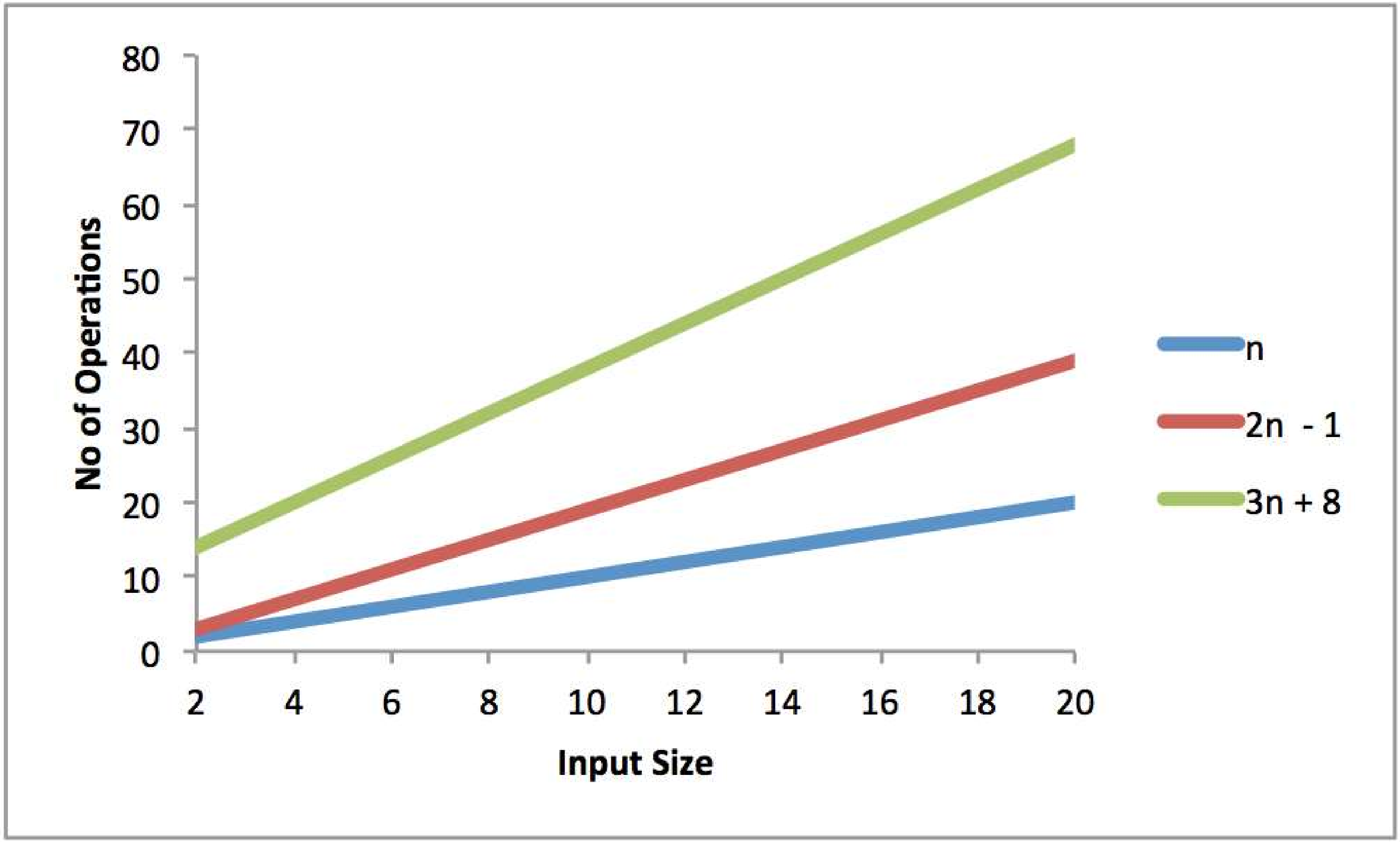
- E.g., $O(n^2)$ means an algorithm is of the order n^2
  - "for large n the number of operations will be roughly n^2"

- Algorithms with same order of complexity are *asymptotically* equal in efficiency

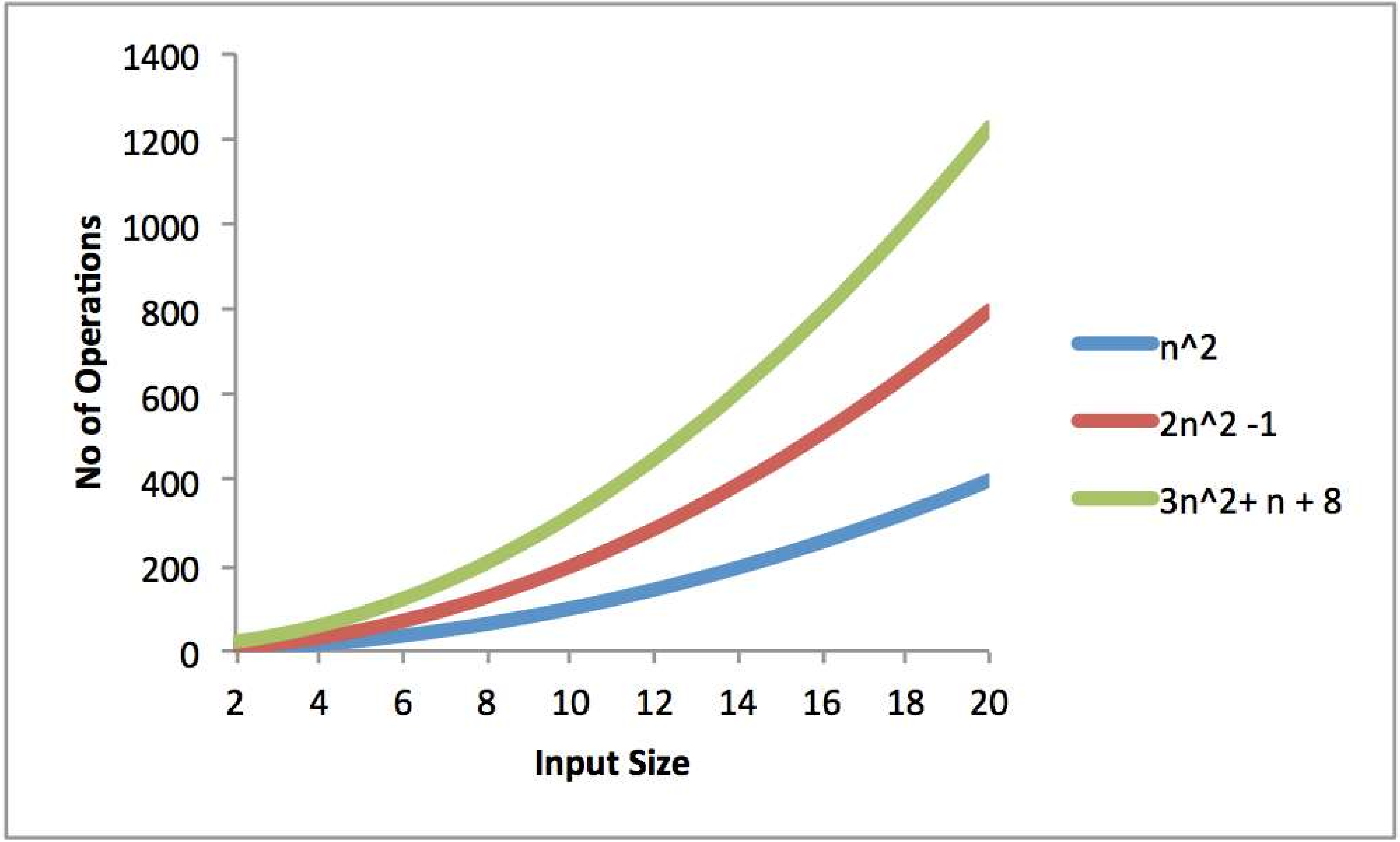
O(1) - Constant Time



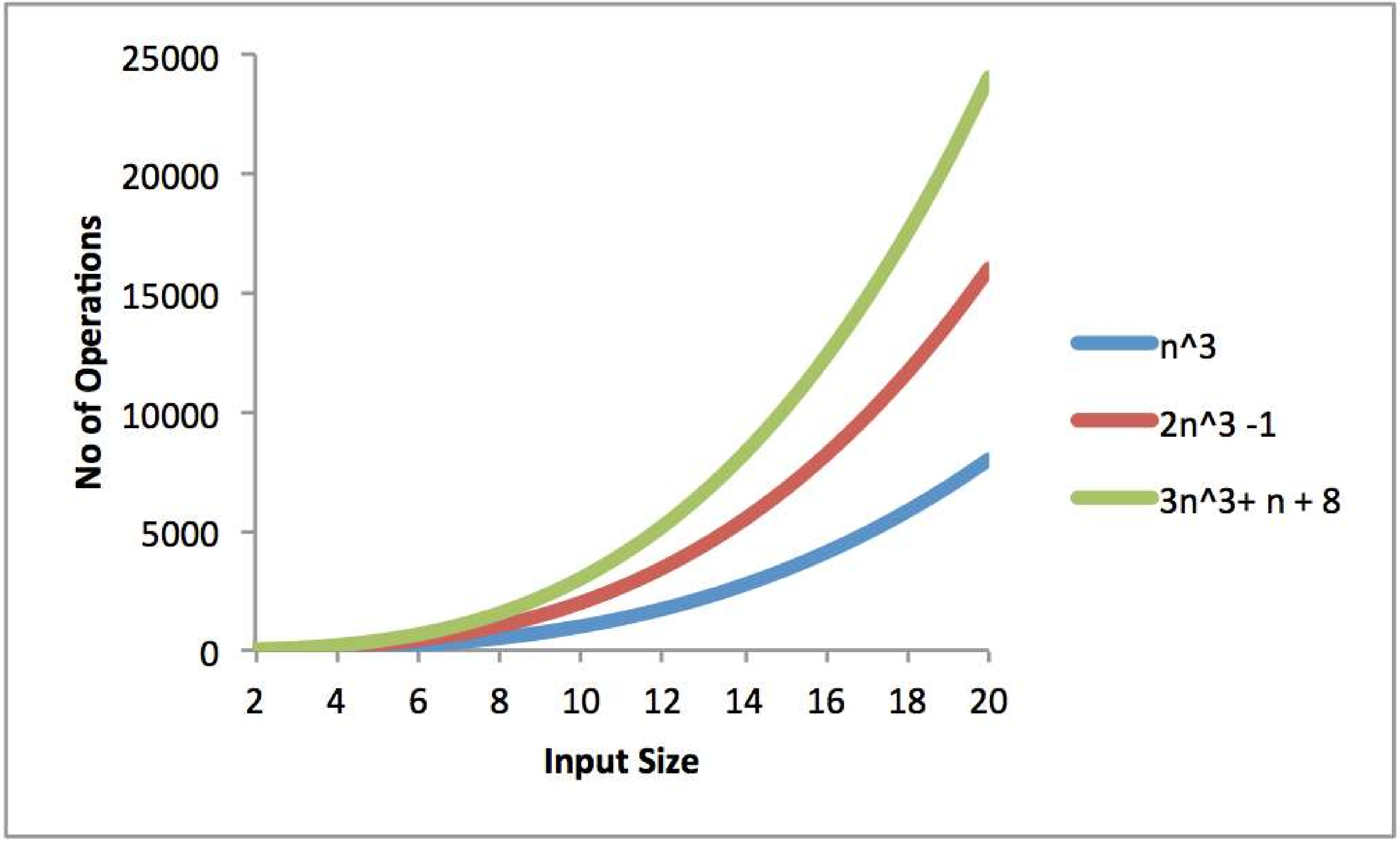
O(n) - Linear Time



O(n^2) - Quadratic Time



O(n^3) - Cubic Time



Increasing order of complexity

increasing order of complexity

	Big O	Remarks
Constant	O(1)	not affected by input size `n`
Logarithmic	O(log `n`)	we will see this during decomposition/recursion
Linear	O(`n`)	roughly proportional to input size `n`
Linearithmic	O(`n` log `n`)	we will see this during decomposition/recursion
Polynomial	O(n^k)	`k` is some constant, e.g., `k` = 1 is linear, `k` = 2 is quadratic, `k` = 3 is cubic
Exponential	O(k^n)	`k` is some constant
Factorial	O(`n!`)	often considered to be within exponential family

Thinking in terms of Growth

increasing order of complexity

Big O	Remarks
O(1)	when `n` doubles, the number of operations remains the same
O(log `n`)	when `n` doubles, the number of operations increase by 1 (for $\log_2 n$) when `n` x10, the number of operations increase by 1 (for $\log_{10} n$) (note: we shall see later that the base is not significant for Big O notation)
O(`n`)	when `n` doubles, the number of operations also doubles
O(n^2)	when `n` doubles, the number of operations also quadruples
O(2^n)	when `n` increases by 1, the number of operations doubles
O(`n!`)	when `n` increases by 1, the number of operations increases `n` times

Dominance rules for Big O notation

- Exponential dominates polynomial, which dominates logarithmic.

  e.g.:

  - If number of operations is $2^n + n^2$, complexity is $O(2^n)$
  - If number of operations is $n^2 + \log n$, complexity is $O(n^2)$

- Higher order dominates lower order

  e.g.:

  - If number of operations is $n^3 + n^2 + n$, the complexity is $O(n^3)$

- Ignore multiplicative constants in the highest-order term

  e.g.:

  - If number of operations is $3n^2$, the complexity is $O(n^2)$

Why keep only the dominant term?

Dominant term

- E.g. 1
  No of steps = $\boxed{\boxed{2^n} + n^2}$
  → $O(2^n)$

n	2^n	n^2	$2^n + n^2$	% of 2^n
1	2	1	3	67%
5	32	25	57	56%
10	1024	100	1124	91%
20	1048576	400	1048976	100%
30	1073741824	900	1073742724	100%
40	1.09951E+12	1600	1.09951E+12	100%
50	1.1259E+15	2500	1.1259E+15	100%
100	1.26765E+30	10000	1.26765E+30	100%
200	1.60694E+60	40000	1.60694E+60	100%
300	2.03704E+90	90000	2.03704E+90	100%
400	2.5822E+120	160000	2.5822E+120	100%
500	3.2734E+150	250000	3.2734E+150	100%
1000	1.0715E+301	1000000	1.0715E+301	100%

Why keep only the dominant term?

Dominant term

- E.g. 2
  No of steps = $\boxed{n^2} + \log n$
  → $O(n^2)$

n	n^2	log n	n^2 + log n	% of n^2
10	100	1.00	101.00	99%
20	400	1.30	401.30	100%
30	900	1.48	901.48	100%
40	1600	1.60	1601.60	100%
50	2500	1.70	2501.70	100%
100	10000	2.00	10002.00	100%
200	40000	2.30	40002.30	100%
300	90000	2.48	90002.48	100%
400	160000	2.60	160002.60	100%
500	250000	2.70	250002.70	100%
1000	1000000	3.00	1000003.00	100%
10000	100000000	4.00	100000004.00	100%
100000	10000000000	5.00	10000000005.00	100%

Why Drop Multiplicative Constant?

- When we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter.
- However, this means that two algorithms can have the **same** big-O time complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires $10 * N^2 + N$ time. For both algorithms, the time is $O(N^2)$, but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms do matter in terms of which algorithm is actually faster.
- However, constants do <u>not</u> matter in terms of how an algorithm "scales" (i.e. how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an algorithm that requires $10*N^2$ time, for **both** algorithms, if the problem size doubles, the actual time will quadruple.
- When two algorithms have **different** big-O time complexity, the constants and low-order terms only matter when the problem size is small. For example, even if there are large constants involved, a linear-time $O(n)$ algorithm will always eventually be faster than a quadratic-time $O(n^2)$ algorithm.

Extracted from http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html

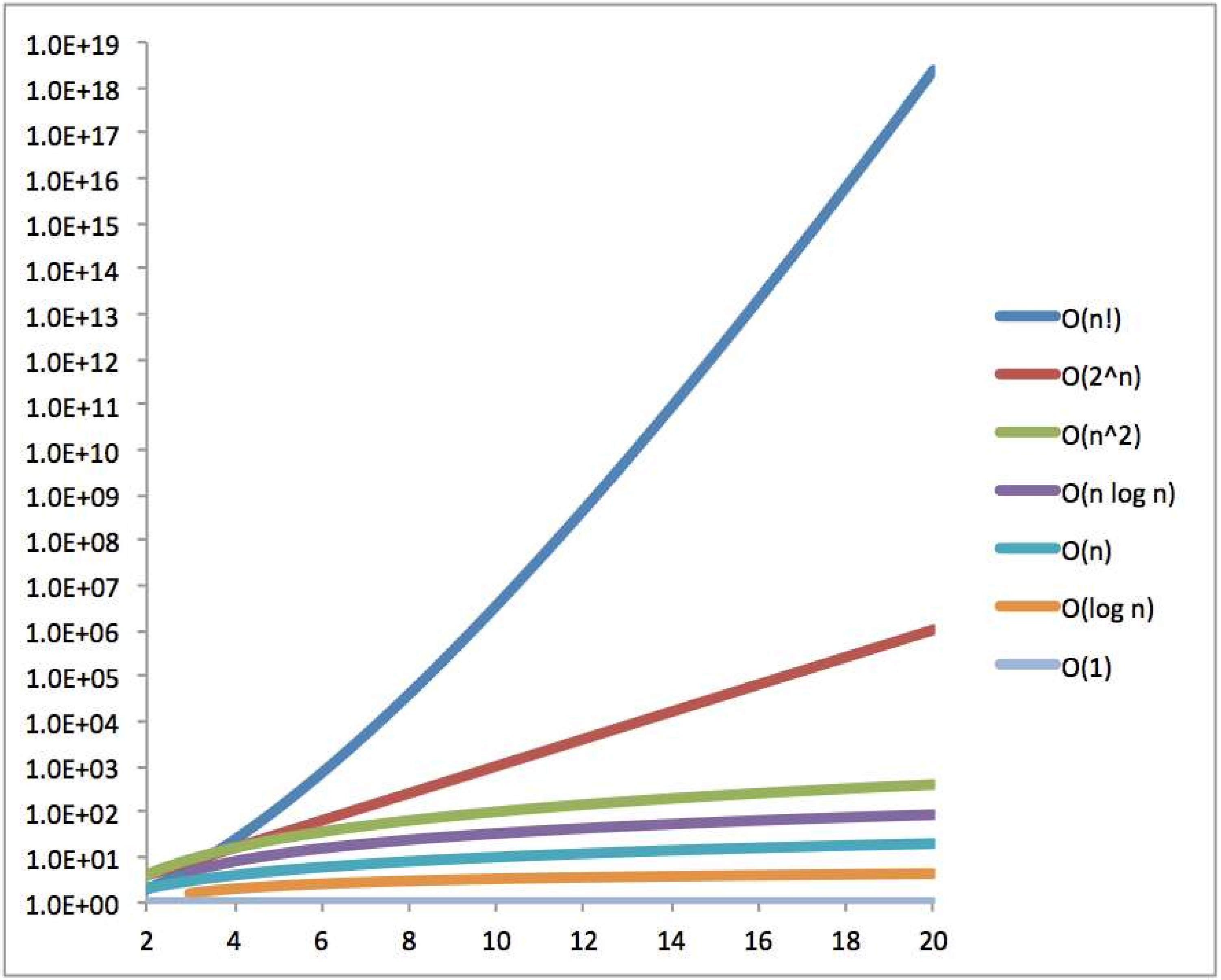
Steps towards Big O

- Characterize the worst case
  - Exercise some creativity
  - Understand different complexity classes of problems
- Count the number of operations in terms of the input size `n`
  - Use your Counting skills learnt in Week 1
- Reduce by dropping the less dominant terms
  - Use the guidelines given in this week’s lesson

Complexity helps to answer these questions

- How difficult is this problem?
  - More complex problems require more computations, and thus are more difficult to solve with a computer.
- Among algorithms that solve the same problem, which is better?
  - Algorithms with lower complexity are preferred.
- Is this problem solvable, or even solved?
  - An algorithm with polynomial complexity (or lower) is commonly considered “efficient”.
  - Some problems with exponential complexity can still be “solved” using heuristic algorithms (see future lesson).
  - Usually with some heuristic reasoning, most problems have solutions with low-degree polynomials such as n, $n \log n$, n^2, or n^3

Log Scale



factorial
exponential
polynomial
linearithmic
linear
logarithmic
constant

If each operation takes one nanosecond (10^{-9})

input size	log n	n	n log n	n^2	2^n	n!
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1 ms	77.1 yrs
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1 sec	8 x 10^{15} yrs
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 days	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	4 x 10^{13} yrs	
1,000	0.010 µs	1.00 µs	9.966 µs	1 ms		
10,000	0.013 µs	10 µs	130 µs	100 ms		
100,000	0.017 µs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 µs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 µs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 µs	0.10 sec	2.66 sec	115.7 days		

Figure 2.4 in Steven S. Skiena, "The Algorithm Design Manual," Springer, 2nd Edition, 2010.

Faster Algorithm vs. Faster Machine

Largest size of problem that can be solved in 1 sec

		Slower Machine (1 operation per μs)	1000x Faster Machine (1 operation per ns)
Slower Algorithm (Exponential)	$n!$	9.5	12.5
	2^n	20	30
Faster Algorithm (Polynomial)	n^3	100	1,000
	n^2	1,000	31,623
	n	1,000,000	1,000,000,000

Summary

- The concept of algorithms
- The concept of computational complexity
  - Efficiency is measured in terms of growth rate
- Big O to indicate the order of complexity in worst case scenarios
- Different orders of complexity
  - Constant, logarithmic, polynomial, exponential

In-class Exercises: What is the Big O?

	f(n)	Big O
(a)	$3n^3 - 27n^2 + 9n + 10$	
(b)	$n^2 - \log n + 9n$	
(c)	$n \log n + 9n$	
(d)	$2^n + n^2$	

In-class Exercises: What is the Big O?

(a) Given an array `a` of `n` numbers, where `n > 10`, find out which of the first `10` numbers is the largest.

(b) Given an array `a` of `n` numbers, find the smallest difference between any two numbers in the array `a`.

(c) There are `n` students in the class. Find 3 students with different last names.

(03) Complexity Part 2b (Solution to In-class Ex) Video (9 mins):

https://www.youtube.com/watch?v=nLtPWkZB3aA&list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&index=23

Road Map

Algorithm Design and Analysis

- Week 1: Counting, Programming
- Week 2: Programming
- Week 3: Complexity
- Next week → Week 4: Iteration & Decomposition
- Week 5: Recursion

Fundamental Data Structures

(Weeks 6 - 10)

Computational Intractability and Heuristic Reasoning

(Weeks 11 - 13)

Useful Formulas for your Tutorial

- $\log(n^x) = x * \log n$ ← note: this is different from $(\log n)^x$
- $\log a + \log b = \log(a * b)$
- $\log a - \log b = \log(a / b)$
- $\log_a x = \log_b x / \log_b a$ ← how to change the base

- This is an AP series: 1 + 2 + 3 + 4…. + n
  - sum of AP series = **N**/2 * (a + l), where
    - **N** is the number of terms
    - a is the first number in the series
    - l is the last number in the series
  - sum = n/2 * (1 + n)

Announcements

- Remember to do your labs **on time**.
- Remember to watch the videos for week 4 (Iteration & Decomposition) + attempt SCQ before next lesson
- Open consultation hours (Mon 9-10 a.m.) - you are welcome!