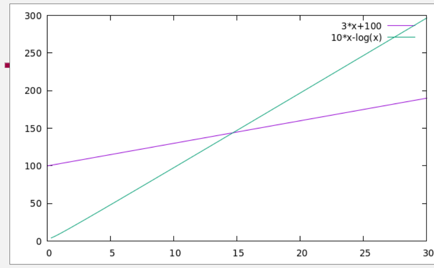
Understanding Definitions

For each of the following pairs of functions, indicate whether f = O(g), f = Ω(g), or f = Θ(g).

a. $f(n) = 3n + 100, g(n) = 10n - \log n$ [pairA]
b. $f(n) = (\log\ n)^2 + 5n\ \log\ n, g(n) = 2n$ [pairB]
c. $f(n) = 3n^2 + n^3, g(n) = 3^n - 5n^3$ [pairC]

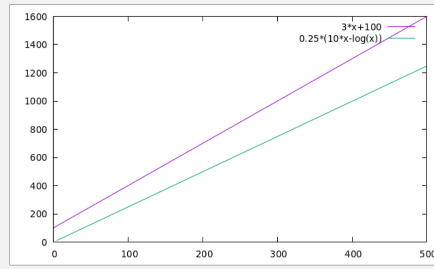
O gives an *upper bound* on a function's growth rate
Ω gives a *lower bound* on a function's growth rate
Θ gives a *tight bound* on a function's growth rate

notation	meaning	definition
f(n) = O(g(n))	c g(n) is an upper bound on f(n)	there exists c > 0 and n_0 > 0 such that f(n) ≤ c g(n) for all n ≥ n_0
f(n) = Ω(g(n))	c g(n) is an lower bound on f(n)	there exists c > 0 and n_0 > 0 such that f(n) ≥ c g(n) for all n ≥ n_0
f(n) = Θ(g(n))	c_1 g(n) is an upper bound on f(n) c_2 g(n) is an lower bound on f(n)	there exists c_1 > 0, c_2 > 0, and n_0 > 0 such that f(n) ≤ c_1 g(n) and f(n) ≥ c_2 g(n) for all n ≥ n_0



3n+100 = O(10n-log n)

because
3n+100 ≤ c(10n-log n)
for c = 1 and n > 15

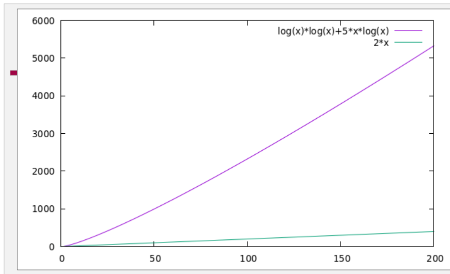


3n+100 = Ω(10n-log n)

because
3n+100 ≥ c(10n-log n)
for c = 0.25 and n > 0

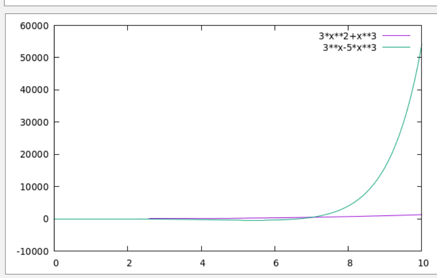
thus
3n+100 = Θ(10n-log n)

because
3n+100 ≤ c_1(10n-log n) and
3n+100 ≥ c_2(10n-log n) for
c_1 = 1, c_2 = 0.25, and n > 15



$(\log n)^2 + 5n \log n = \Omega(2n)$

because
$(\log n)^2 + 5n \log n \geq 2n$
for c = 1 and n > 5



$3n^2+n^3 = O(3^n-5n^3)$

because
$3n^2+n^3 \leq c(3^n-5n^3)$
for c = 1 and n > 8

O, Ω, Θ vs Best and Worst Cases

The big-Oh notation compares growth rates of functions – comparing shapes of curves.

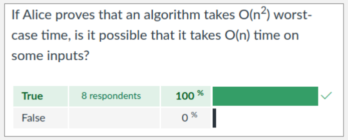
- f(n) = O(g(n)) says that f(n) grows no faster than g(n)
  - g(n) is an upper bound on the growth rate
- f(n) = Ω(g(n)) says that f(n) grows no slower than g(n)
  - g(n) is a lower bound on the growth rate
- f(n) = Θ(g(n)) says that f(n) grows at the same rate as g(n)
  - g(n) is a tight bound on the growth rate

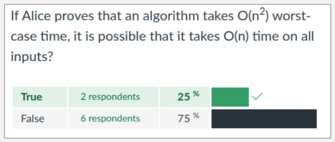
The best (or worst) case is the specific input instance that yields the fastest (or slowest) running time over all possible input instances of a given size – comparing the actual number of steps required.

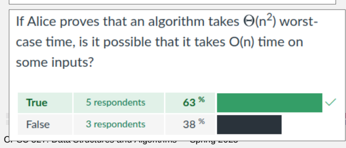
- no input instance will take longer than the worst case for that size, or take less time than the best case for that size

Understanding Terminology and Concepts

- in all cases, the answer is “yes” – why?







- worst-case means nothing is slower, but faster is possible
  - e.g. insertion sort
- O is an upper bound, so $f(n) = O(n^2)$ says that f(n) doesn't grow any faster than n^2, but it doesn't preclude it growing slower i.e. $n = O(n^2)$ though typically we want to give the tightest bound we can
- Θ means that the worst case won't actually turn out to be better than n^2, but the worst case is the slowest input of a given size and others (e.g. best case) may be better

O, Ω, Θ vs Best and Worst Cases

Saying that the worst-case behavior is **$O(n^2)$** means –

- some inputs could be O(n) because the worst case is the slowest instance for a given size
- all inputs could be O(n) because n grows no faster than n^2, though one generally tries to give the tightest O possible

Saying that the worst-case behavior is **$\Theta(n^2)$** means –

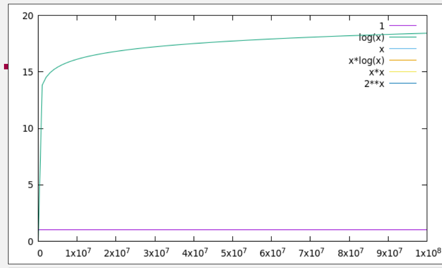
- some inputs could be O(n) because the worst case is the slowest instance for a given size
- not all inputs could be O(n) because then the worst case instances would also be O(n) and n does not grow at the same rate as n^2

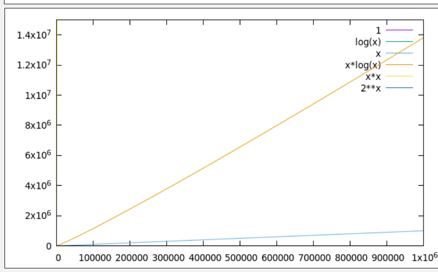
O, Ω, or Θ?

- give as tight as bound as possible
- use Θ if you can
  - e.g. mergesort is Θ(n log n)
  - e.g. insertion sort is best case Θ(n) and worst case $\Theta(n^2)$
- can use O if best case running time grows more slowly than the worst case (or Ω if worst case running time grows faster than the best case) but you don't want to distinguish – only worst (or best) case is important
  - e.g. insertion sort is $O(n^2)$
  - e.g. insertion sort is Ω(n)
- can use O (or Ω) if you can't establish a tight bound
  - you don’t know if the best case is better or if the worst case is worse

Implications for Algorithm Design

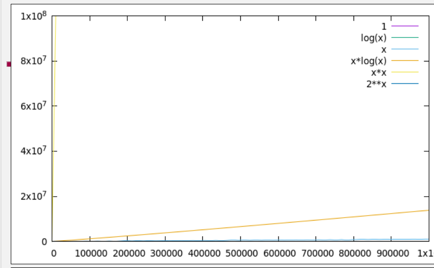
Θ	**fast computer**	**1000x faster**
1	n is irrelevant	n is irrelevant
log n	any n is fine	any n is fine
n **n log n**	still practical for n = 1,000,000	still practical for n = 1,000,000,000
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	usable up to n = 300,000 hopeless for n > 30,000,000
2^n	impractical for n > 40	impractical for n > 50
n!	useless for n ≥ 20	useless for n ≥ 22

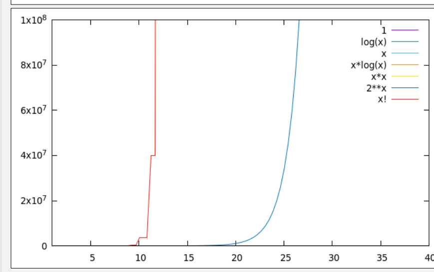




Θ	fast computer
1	n is irrelevant
log n	any n is fine

n	still practical for n = 1,000,000
n log n	





n	still practical for n = 1,000,000
n log n	
n^2	usable up to n = 10,000 hopeless for n > 1,000,000

2^n	impractical for n > 40
n!	useless for n ≥ 20

Implications for Algorithm Design

Θ	running time on fast computer	characteristics of typical tasks with the specified running time
1	n is irrelevant	examine only a fixed number of things regardless of input size
log n	any n is fine	repeatedly eliminate a fraction of the search space
n	still practical for n = 1,000,000	examine each object a fixed number of times
n log n		divide-and-conquer with linear time per step mergesort, quicksort
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	examine all pairs insertion sort, selection sort
n^3		examine all triples
2^n	impractical for n > 40	enumerate all subsets
n!	useless for n ≥ 20	enumerate all permutations

Big-Oh From Algorithms

use the table on the previous slide

An array contains each of the numbers 1..n plus one duplicate value. Which value is duplicated?

- Algorithm A uses quicksort or mergsort to sort all of the numbers, then makes one pass through the array looking for adjacent slots with the same value.
- Algorithm B makes one pass through the array to sum the numbers, then uses the formula $\frac{n(n-1)}{2}$ to calculate the sum of the numbers 1..n and subtracts that from the sum of the array's value.
- Algorithm C makes one pass through the array and for each value, makes a pass through the rest of the array to see if another copy of that value is found i.e. each value in the array is compared to each other value to find the duplicate.

sort, then examine each object a fixed number of times → Θ(n log n) + Θ(n) = Θ(n log n)

examine each object a fixed number of times, then examine only a fixed number of things → Θ(n) + Θ(1) = Θ(n)

for each object, examine each object a fixed number of times → Θ(n) x Θ(n) = $\Theta(n^2)$

		B	A	C		
n	$\lg n$	n	$n \lg n$	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 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{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	4×10^{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		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

suitability for n = 25, 2500, 250,000, 250,000,000

Questions

How do you choose between multiple algorithms with suitable big-Ohs?

n	still practical for n = 1,000,000	examine each object a fixed number of times
n log n		divide-and-conquer with linear time per step mergesort, quicksort
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	examine all pairs insertion sort, selection sort

- if n = 1,000, all three of these are potentially suitable
- consider other factors
  - is there already a library implementation?
  - if you have to implement something, which is simpler to implement (and implement correctly)?
  - are there significant differences in memory usage?

Questions

O(n log n) is pretty practical – why couldn't you just use mergesort or quicksort for a very large array?

n	still practical for n = 1,000,000	examine each object a fixed number of times
n log n		divide-and-conquer with linear time per step mergesort, quicksort

- real systems have only a limited amount of memory
  - if the array is too large to fit into memory, it is kept on disk and parts are swapped into memory when needed
- if successive accesses are scattered throughout the array, the system spends all of its CPU time swapping things in and out of memory instead of actually sorting
  - the assumption that each memory access is one time step also breaks down
- need algorithms exhibiting *locality of access* to minimize swaps