Big-Oh

asymptotic growth rate or *order*

compare two functions, but...

ignore constant factors, small inputs

asymptotic growth rate or *order*

compare two functions, but...

ignore constant factors, small inputs

example: $f(n) = 1\,000\,000 \cdot n^2$; $g(n) = 2^n$

g grows faster — eventually much bigger than f

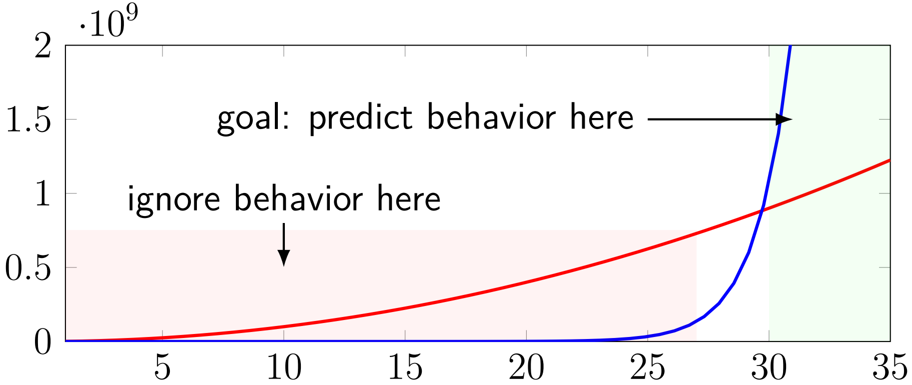
asymptotic growth rate or *order*

compare two functions, but...

ignore constant factors, small inputs

example: $f(n) = 1\,000\,000 \cdot n^2$; $g(n) = 2^n$

g grows faster — eventually much bigger than f



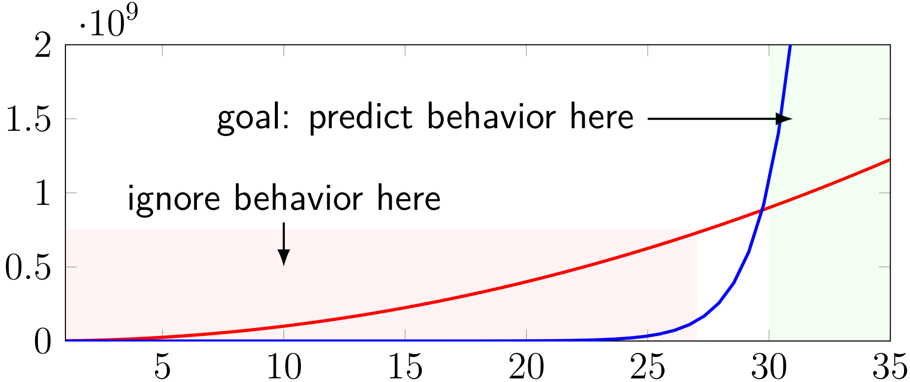
asymptotic growth rate or *order*

compare two functions, but...

ignore constant factors, small inputs

example: $f(n) = 1\,000\,000 \cdot n^2$; $g(n) = 2^n$

g grows faster — eventually much bigger than f



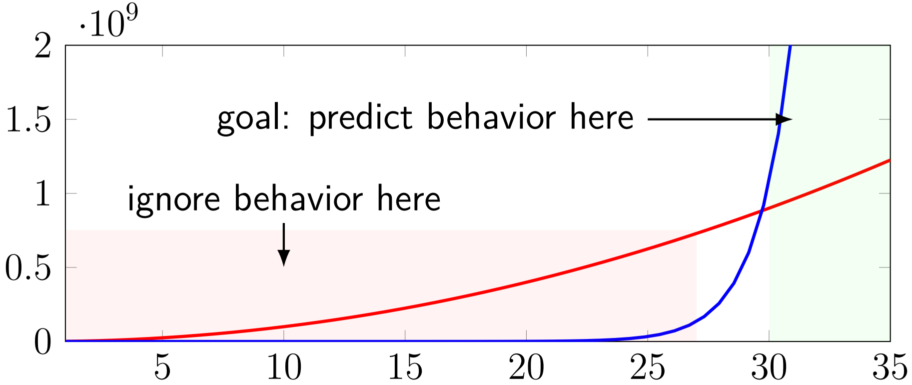
asymptotic growth rate or *order*

compare two functions, but...

ignore constant factors, small inputs

example: $f(n) = 1\,000\,000 \cdot n^2$; $g(n) = 2^n$

g grows faster — eventually much bigger than f



preview: what functions?

example: comparing sorting algorithms

runtime $= f$(size of input)

e.g. seconds to sort $= f$(number of elements in list)

e.g. # operations to sort $= f$(number of elements in list)

space $= f$(size of input)

e.g. number of bytes of memory $= f$(number of elements in list)

theory, not empirical

yes, you can make *guesses* about big-oh behavior from measurements

but, no, graphs $\neq$ big-oh comparison

- what happens further to the right?
- might not have tested big enough

want to write down formula

theory, not empirical

yes, you can make *guesses* about big-oh behavior from measurements

but, no, graphs $\neq$ big-oh comparison

- what happens further to the right?
- might not have tested big enough

want to write down formula

example: summing a list of n items:

- exactly n addition operations
- *assume* each one takes k unit of time
- runtime $= f(n) = kn$

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete
`Vector`	87.818	0.004	63.202	24.612 s
`ArrayList`	87.192	0.010	62.470	24.712 s
`LinkedList`	263.776	0.006	196.550	67.439 s
`HashSet`	0.029	0.022	0.003	0.004 s
`TreeSet`	0.134	0.110	0.017	0.007 s
`Vector`, sorted	2.642	0.009	0.024	2.609 s

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete	
`Vector`	87.818	0.004	63.202	24.612	s
`ArrayList`	87.192	0.010	62.470	24.712	s
`LinkedList`	263.776	0.006	196.550	67.439	s
`HashSet`	0.029	0.022	0.003	0.004	s
`TreeSet`	0.134	0.110	0.017	0.007	s
`Vector`, sorted	2.642	0.009	0.024	2.609	s

some runtimes get really big as size gets large...

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete
`Vector`	87.818	0.004	63.202	24.612 s
`ArrayList`	87.192	0.010	62.470	24.712 s
`LinkedList`	263.776	0.006	196.550	67.439 s
`HashSet`	0.029	0.022	0.003	0.004 s
`TreeSet`	0.134	0.110	0.017	0.007 s
`Vector`, sorted	2.642	0.009	0.024	2.609 s

others seem to remain manageable

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete
`Vector`	87.818	0.004	63.202	24.612 s
`ArrayList`	87.192	0.010	62.470	24.712 s
`LinkedList`	263.776	0.006	196.550	67.439 s
`HashSet`	0.029	0.022	0.003	0.004 s
`TreeSet`	0.134	0.110	0.017	0.007 s
`Vector`, sorted	2.642	0.009	0.024	2.609 s

problem: growth rate of runtimes with list size

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete	
`Vector`	87.818	0.004	63.202	24.612	s
`ArrayList`	87.192	0.010	62.470	24.712	s
`LinkedList`	263.776	0.006	196.550	67.439	s
`HashSet`	0.029	0.022	0.003	0.004	s
`TreeSet`	0.134	0.110	0.017	0.007	s
`Vector`, sorted	2.642	0.009	0.024	2.609	s

for `Vector` (unsorted), `ArrayList`, `LinkedList`...
$\#$ operations grows like n where n is list size

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete
`Vector`	87.818	0.004	63.202	24.612 s
`ArrayList`	87.192	0.010	62.470	24.712 s
`LinkedList`	263.776	0.006	196.550	67.439 s
`HashSet`	0.029	0.022	0.003	0.004 s
`TreeSet`	0.134	0.110	0.017	0.007 s
`Vector`, sorted	2.642	0.009	0.024	2.609 s

for `HashSet`...
operations per search/remove is constant (sort of)

recall: comparing list data structures

List benchmark (from intro slides) w/ 100000 elements

Data structure	Total	Insert	Search	Delete
`Vector`	87.818	0.004	63.202	24.612 s
`ArrayList`	87.192	0.010	62.470	24.712 s
`LinkedList`	263.776	0.006	196.550	67.439 s
`HashSet`	0.029	0.022	0.003	0.004 s
`TreeSet`	0.134	0.110	0.017	0.007 s
`Vector`, sorted	2.642	0.009	0.024	2.609 s

for `TreeSet`, sorted `Vector`...
$\#$ operations per search grows like $\log(n)$ where n is list size

why asymptotic analysis?

"can my program work when data gets big?"

- website gets thousands of new users?
- text editor opening 1MB book? 1 GB log file?
- music player sees $1\,000$ song collection? $50\,000$?
- text search on 100 petabyte copy of the text of the web?

why asymptotic analysis?

"can my program work when data gets big?"

website gets thousands of new users?

text editor opening 1MB book? 1 GB log file?

music player sees $1\,000$ song collection? $50\,000$?

text search on 100 petabyte copy of the text of the web?

if asymptotic analysis says "no"

can find out before implementing algorithm

won't be fixed by, e.g., buying a faster CPU

sets of functions

define sets of functions based on an example f

$\Omega(f)$: grow no slower than f ("$\geq f$")

$O(f)$: grow no faster than f ("$\leq f$")

$\Theta(f) = \Omega(f) \cap O(f)$: grow as fast as f ("$= f$")

sets of functions

define sets of functions based on an example f

$\Omega(f)$: grow no slower than f ("$\geq f$")

$O(f)$: grow no faster than f ("$\leq f$")

$\Theta(f) = \Omega(f) \cap O(f)$: grow as fast as f ("$= f$")

examples:

$n^3 \in \Omega(n^2)$
$100n \in O(n^2)$
$10n^2 + n \in \Theta(n^2)$ — ignore constant factor, etc.
and $10n^2 + n \in O(n^2)$ and $10n^2 + n \in \Omega(n^2)$

what are we measuring

$$
\begin{aligned}
f(n) &= \text{worst case running time} \\
n &= \text{input size — as a positive integer}
\end{aligned}
$$

what are we measuring

$$f(n) = \text{worst case running time}$$
$$n = \text{input size — as a positive integer}$$

will comapre f to another function $g(n)$

example: $f(n) \in O(g(n))$ (or $f \in O(g)$)
informally: "f is big-oh of g"

example $f(n) \notin \Omega(g(n))$ or ($g \notin \Omega(g)$)
informally: "f' is not big-omega of g"

what are we measuring

$$
\begin{aligned}
f(n) &= \text{worst case running time} \\
n &= \text{input size — as a positive integer}
\end{aligned}
$$

will comapre f to another function $g(n)$

example: $f(n) \in O(g(n))$ (or $f \in O(g)$)

informally: "f is big-oh of g"

example $f(n) \notin \Omega(g(n))$ or $(g \notin \Omega(g))$

informally: "f' is not big-omega of g"

worst case?

this class: almost always worst cases

intuition: detect if program will *ever* take "forever"

worst case?

this class: almost always worst cases

intuition: detect if program will *ever* take "forever"

example: iterating through an array until we find a value

- best case: look at one value, it's the one we want
- worst case: look at every value, none of them are what we want

worst case?

this class: almost always worst cases

intuition: detect if program will *ever* take “forever”

example: iterating through an array until we find a value

best case: look at one value, it’s the one we want

worst case: look at every value, none of them are what we want

$f(n)$ is run time of *slowest* input of size n

formal definitions

$f(n) \in O(g(n))$:

there exists $c > 0$ and $n_0 > 0$ such that

for all $n > n_0$, $f(n) \leq c \cdot g(n)$

formal definitions

$f(n) \in O(g(n))$:

there exists $c > 0$ and $n_0 > 0$ such that
for all $n > n_0$, $f(n) \leq c \cdot g(n)$

$f(n) \in \Omega(g(n))$:

there exists $c > 0$ and $n_0 > 0$ such that
for all $n > n_0$, $f(n) \geq c \cdot g(n)$

formal definitions

$f(n) \in O(g(n))$:
there exists $c > 0$ and $n_0 > 0$ such that
for all $n > n_0$, $f(n) \leq c \cdot g(n)$

$f(n) \in \Omega(g(n))$:
there exists $c > 0$ and $n_0 > 0$ such that
for all $n > n_0$, $f(n) \geq c \cdot g(n)$

$f(n) \in \Theta(g(n))$:
$f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

formal definition example (1)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $n \in O(n^2)$:

formal definition example (1)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $n \in O(n^2)$:
choose $c = 1$, $n_0 = 2$
for $n > 2 = n_0$: $n \leq c \cdot n^2 = n^2$
Yes!

formal definition example (2)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $10n \in O(n)$?

formal definition example (2)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $10n \in O(n)$?

choose $c = 11$, $n_0 = 2$
for $n > 2 = n_0$: $f(n) = 10n \leq c \cdot g(n) = 11n$
Yes!

formal definition example (2)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $10n \in O(n)$?

choose $c = 11$, $n_0 = 2$
for $n > 2 = n_0$: $f(n) = 10n \leq c \cdot g(n) = 11n$
Yes!

don't need to choose smallest possible c

negating formal definitions

$f \in O(g)$: there exists $c, n_0 > 0$ so for all $n > n_0$: $f(n) \leq cg(n)$

$f \notin O(g)$:

there does not exist $c, n_0 > 0$ so for all $n > n_0$: $f(n) \leq cg(n)$
for all c, n_0, there exists $n > n_0$: $f(n) > cg(n)$

formal definition example (3)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $n^2 \in O(n)$?

formal definition example (3)

$f(n) \in O(g(n))$ if and only if

there exists $c > 0$ and $n_0 > 0$ such that

$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $n^2 \in O(n)$?

no — consider any $c, n_0 > 0$

consider $n_{bad} = (c + 100)(n_0 + 100) > n_0$

$n_{bad}^2 = (c + 100)^2(n_0 + 100)^2 > c(c + 100)(n_0 + 100) = cn_{bad}$

so can't find c, n_0 that sastisfy definition

(i.e. $f(n) = n_{bad}^2 \not\leq c \cdot g(n_{bad}) = cn_{bad}$)

formal definition example (3)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

Is $n^2 \in O(n)$?

no — consider any $c, n_0 > 0$
consider $n_{bad} = (c + 100)(n_0 + 100) > n_0$
$n_{bad}^2 = (c + 100)^2 (n_0 + 100)^2 > c(c + 100)(n_0 + 100) = cn_{bad}$
so can't find c, n_0 that sastisfy definition
(i.e. $f(n) = n_{bad}^2 \not\leq c \cdot g(n_{bad}) = cn_{bad}$)

alternative

$n_{bad} = \max\{c + 100, n_0 + 1\} > n_0$

formal definition example (4)

$f(n) \in O(g(n))$ if and only if
there exists $c > 0$ and $n_0 > 0$ such that
$f(n) \leq c \cdot g(n)$ for all $n > n_0$

consider: $f(n) = 100 \cdot n^2 + n$, $g(n) = n^2$:
choose $c = 200$, $n_0 = 2$
observe for $n > 2$: $100n^2 + n \leq 101n^2$
for $n > 2 = n_0$: $f(n) = 100n^2 + n \leq 101n^2 \leq c \cdot g(n) = 200n^2$

big-oh proofs generally

if proving yes case:

- look at inequality
- *choose* a large enough c and n_0 that it's definitely true
- don't bother finding smallest c, n_0 that work

if proving no case:

- game: given c, n_0 find counter example
- general idea: *choose* $n > n_0$ using a formula based on c
- show that this n never satisfies the inequality
- don't bother showing it's true for all $n' > n$
- don't bother finding smallest n that works

aside: forall/exists

$\forall n > 0$: for all $n > 0$

$\exists n < 0$: there exists an $n < 0$

definition consequences

If $f \in O(h)$ and $g \notin O(h)$, which are true?

1. $\forall m > 0,\ f(m) < g(m)$

   for all m, f is less than g

2. $\exists m > 0,\ f(m) < g(m)$

   there exists an m, so f is less than g

3. $\exists m_0 > 0, \forall m > m_0,\ f(m) < g(m)$

   there exists an m_0, so for all m larger, f is less than g

4. 1 and 2

5. 2 and 3

6. 1 and 2 and 3

definition consequences

If $f \in O(h)$ and $g \notin O(h)$, which are true?

1. $\forall m > 0,\ f(m) < g(m)$
   for all m, f is less than g
2. $\exists m > 0,\ f(m) < g(m)$
   there exists an m, so f is less than g
3. $\exists m_0 > 0, \forall m > m_0,\ f(m) < g(m)$
   there exists an m_0, so for all m larger, f is less than g
4. 1 and 2
5. 2 and 3
6. 1 and 2 and 3

$f \in O(h), g \notin O(h) \;\not\Rightarrow\; \forall m. f(m) < g(m)$

counterexample — $f(n) = 5n$; $g(n) = n^3$; $h(n) = n^2$

$f \in O(h)$: $5n \leq cn^2$ for all $n > n_0$ with $c = 6$, $n_0 = 2$
$g \notin O(h)$: $n^3 \leq cn^2$? use $n \approx cn_0$ as counterexample

$m = 2$: $f(m) = 10 \not< g(m) = 8$

$f \in O(h), g \notin O(h) \not\Longrightarrow \forall m. f(m) < g(m)$

counterexample — $f(n) = 5n$; $g(n) = n^3$; $h(n) = n^2$

$f \in O(h)$: $5n \leq cn^2$ for all $n > n_0$ with $c = 6$, $n_0 = 2$
$g \notin O(h)$: $n^3 \leq cn^2$? use $n \approx cn_0$ as counterexample

$m = 2$: $f(m) = 10 \not< g(m) = 8$

intuition: big-oh ignores behavior for small n

$n^3 \notin O(n^2)$

big-Oh definition requires:

$$n^3 \le cn^2 \text{ for all } n > n_0$$

choose any $c > 1$ and $n_0 > 1$, then

$$n = cn_0 \text{ is a counterexample}$$

$$n^3 = c^3 n_0^3 = cn_0(cn_0)^2 > cn^2$$

contradicting the definition
(and for $c < 1$, use $n = n_0 + 1$, etc.)

$$f \in O(h), g \notin O(h) \implies \exists m. f(m) < g(m)$$

intuition: should be true for 'big enough' m

assume definition of big-Oh:

$f \in O(h)$: $\forall n > n_0 : \; f(n) \leq ch(n)$ (for a $n_0, c > 0$)
$g \notin O(h)$: $\exists n > n_0 : \; g(n) > ch(n)$ (for *any* $n_0, c > 0$)

assume f's n_0, c

use the n that must exist for g (from definition)

$f \in O(h), g \notin O(h) \implies ?\exists m_0 \forall m > m_0 . f(m) < g(m)$

intuitively, seems so g must grow faster than f — for big m:

$$f(m) < c_1 \cdot h(m)$$
$$g(m) < c_2 \cdot h(m)$$

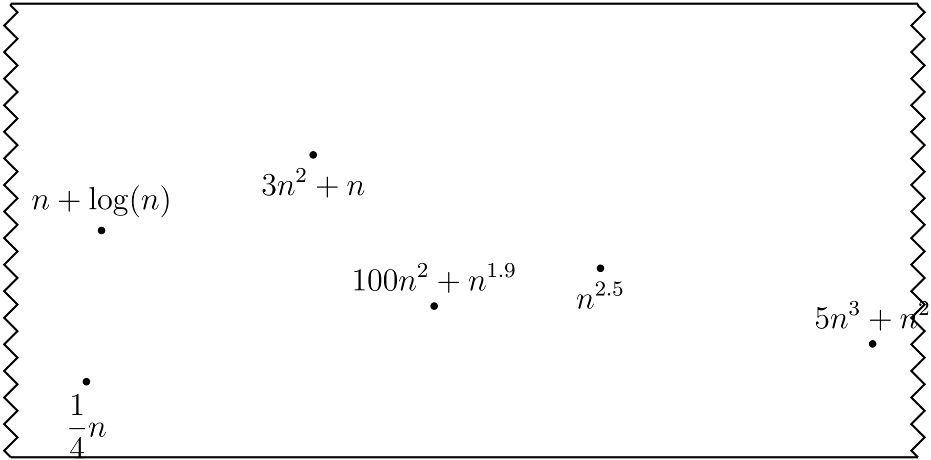
but some corner case counterexamples:

$$f(n) = n$$
$$g(n) = \begin{cases} 1 & n \text{ odd} \\ n^2 & n \text{ even} \end{cases}$$
$$h(n) = n$$

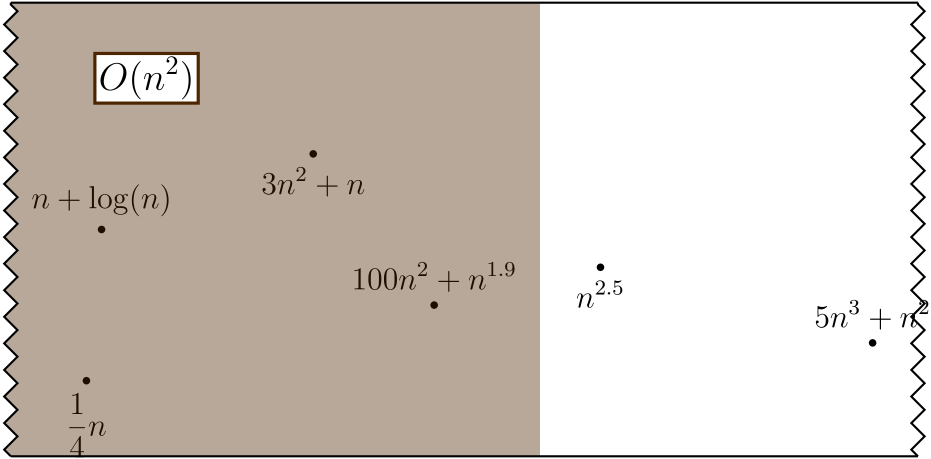
true with additional restriction:

f, g monotonic ($g(n) \leq g(n+1)$, etc.)

function hierarchy

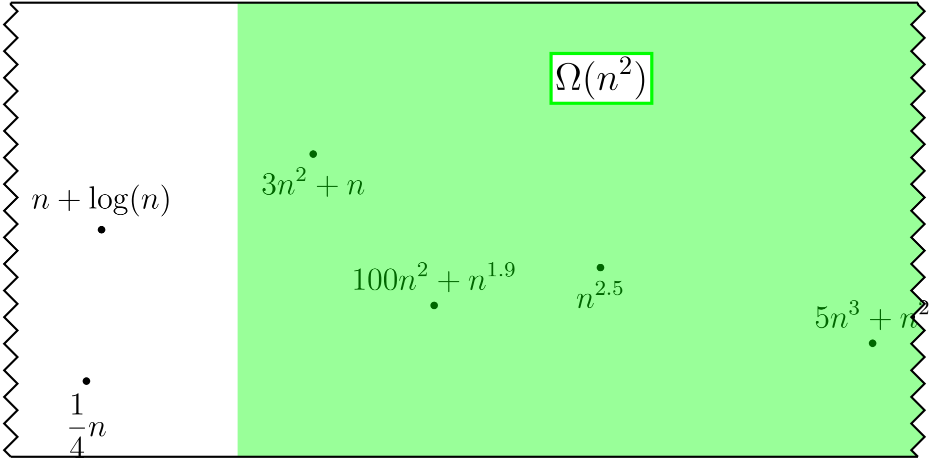


function hierarchy



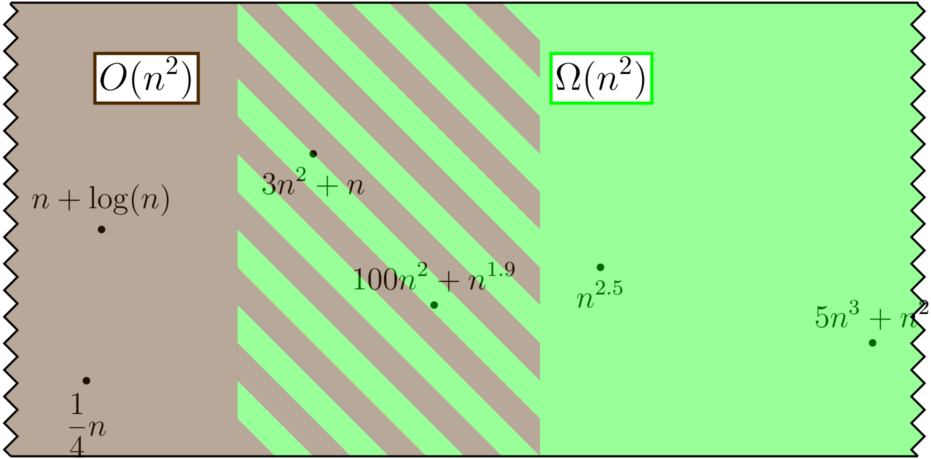
O — upper bound ("$\leq$")

function hierarchy



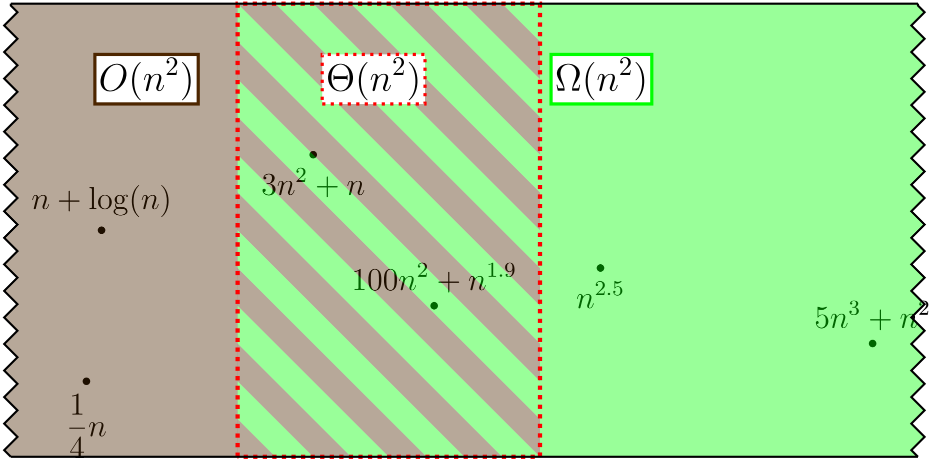
Ω — lower bound ("$\geq$")

function hierarchy



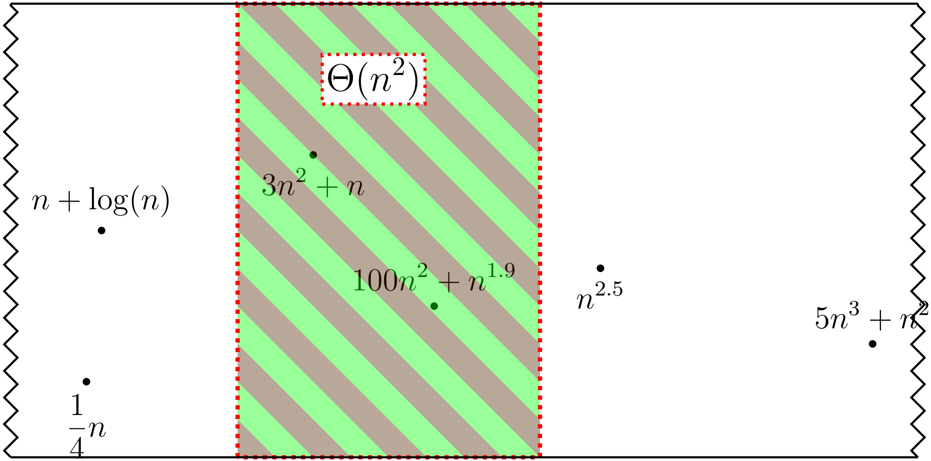
O and Ω overlap

function hierarchy



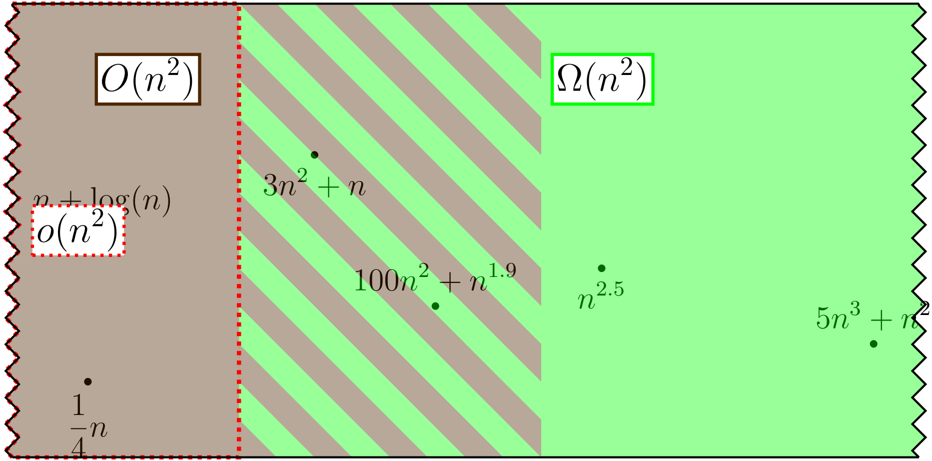
Θ — tight bound ("$=$") — O and Ω

function hierarchy



Θ — tight bound ("=") — O and Ω

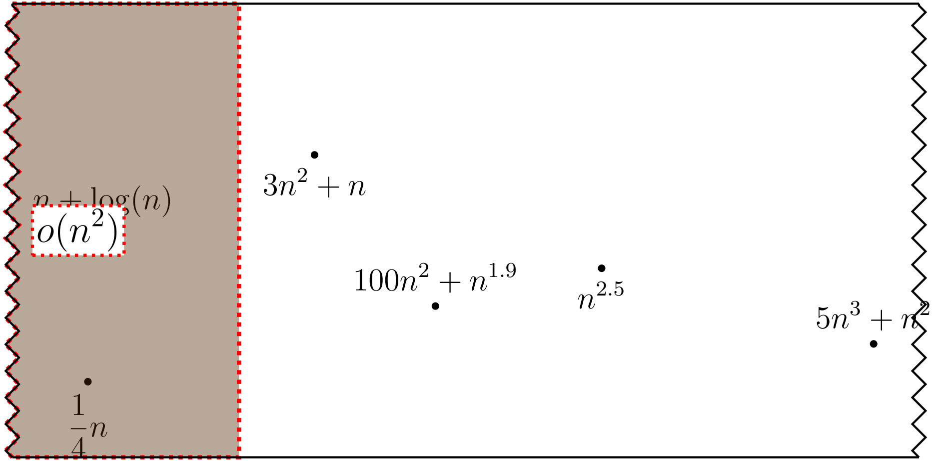
function hierarchy



$g \in o(f)$ ("little-oh")— strict upper bound

$f(n) < c \cdot g(n)$ (all c); (versus $O(f)$: $f(n) \leq c \cdot g(n)$)

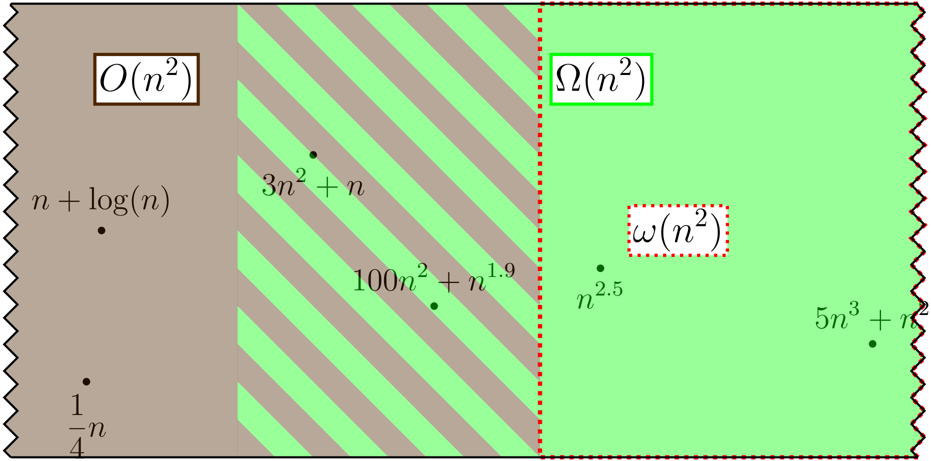
function hierarchy



$g \in o(f)$ ("little-oh")— strict upper bound

$f(n) < c \cdot g(n)$ (all c); (versus $O(f)$: $f(n) \leq c \cdot g(n)$)

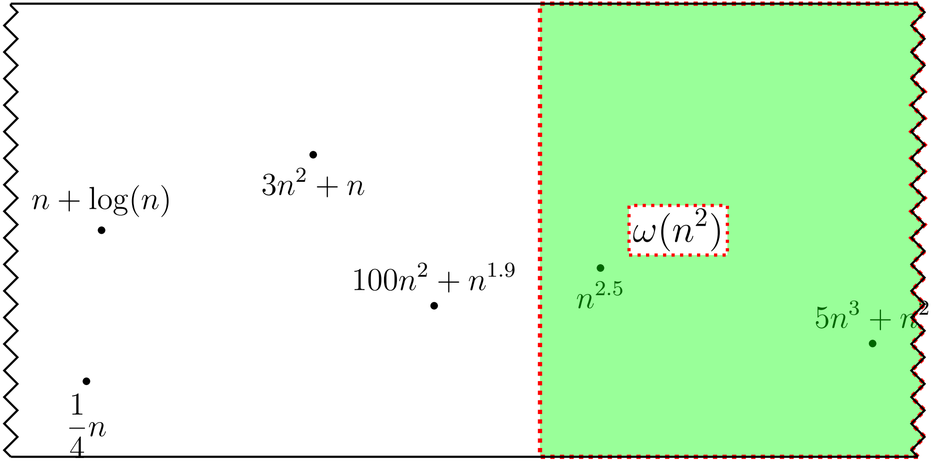
function hierarchy



$g \in \omega(f)$ — strict lower bound

$f(n) > c \cdot g(n)$ (all c); (versus $\Omega(f)$: $f(n) \geq c \cdot g(n)$)

function hierarchy



$g \in \omega(f)$ — strict lower bound

$f(n) > c \cdot g(n)$ (all c); (versus $\Omega(f)$: $f(n) \geq c \cdot g(n)$)

big-Oh variants

$O(f)$	asymptotically less than or equal to f
$o(f)$	asymptotically less than f
$\Omega(f)$	asymptotically greater than or equal to f
$\omega(f)$	asymptotically greater than f
$\Theta(f)$	asymptotically equal to f

limit-based definition

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = X$$

if only if...

$X < \infty$: $f \in O(g)$

$X > 0$: $f \in \Omega(g)$

$0 < X < \infty$: $f \in \Theta(g)$

$X = 0$: $f \in o(g)$

$X = \infty$ (and $\liminf$): $f \in \omega(g)$

limit-based definition

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = X$$

if only if...

$X < \infty$: $f \in O(g)$

$X > 0$: $f \in \Omega(g)$

$0 < X < \infty$: $f \in \Theta(g)$

$X = 0$: $f \in o(g)$

$X = \infty$ (and $\liminf$): $f \in \omega(g)$

lim sup?

$\limsup \frac{f(n)}{g(n)}$ — "limit superior"

equal to normal $\lim$ if it is defined

only care about upper bound

e.g. n^2 in $f(n) = \begin{cases} 1 & n \text{ odd} \\ n^2 & n \text{ even} \end{cases}$

usually glossed over (including in Bloomfield's/Floryan's slides from prior semesters)

some big-Oh properties (1)

for O and Ω and Θ:

$O(f+g) = O(\max(f,g))$

$f \in O(g)$ and $g \in O(h) \implies f \in O(h)$

also holds for o (little-oh), ω

$f \in O(f)$

some big-Oh properties (2)

$f \in O(g) \leftrightarrow g \in \Omega(f)$

$f \in \Theta(g) \leftrightarrow g \in \Theta(f)$

does *not* hold for O, Ω, etc.

Θ is an equivalence relation

reflexive, transitive, etc.

a note on =

informally, sometimes people write $5n^2 = O(n^2)$

not very precise — O is a *set of functions*

selected asymptotic relationships

for $k > 0$, $l > 0$, $c > 1$, $\epsilon > 0$:

$n^k \in o(c^{n^l})$ (polynomial always smaller than exponential)

$n^k \in o(n^k \log n)$ (adding $\log$ makes something bigger)

$\log_k(n) \in \Theta(\log_l(n))$ (all log bases are the same)

$n^k + cn^{k-1} \in \Theta(n^k)$ (only polynomial degree matters)

a note on logs

$$\log_k(n) = \frac{\log_l(n)}{\log_l(k)} = c \cdot \log_l(n)$$

therefore

$$\Theta(\log_k(n)) = \Theta(\log_l(n))$$

...so doesn't matter which base of log we mean

some names

$\Theta(1)$ — constant (some fixed maximum)

read kth element of array

$\Theta(\log n)$ — logarithmic

binary search a sorted array

$\Theta(n)$ — linear

searching an unsorted array

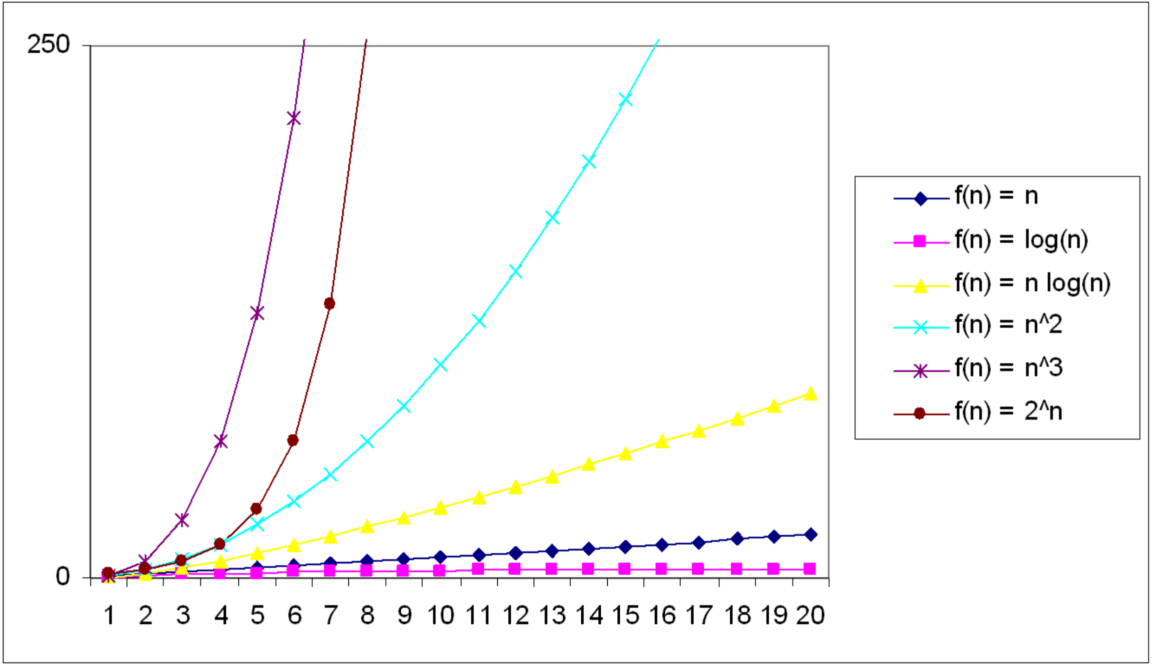
$\Theta(n \log n)$ — log-linear

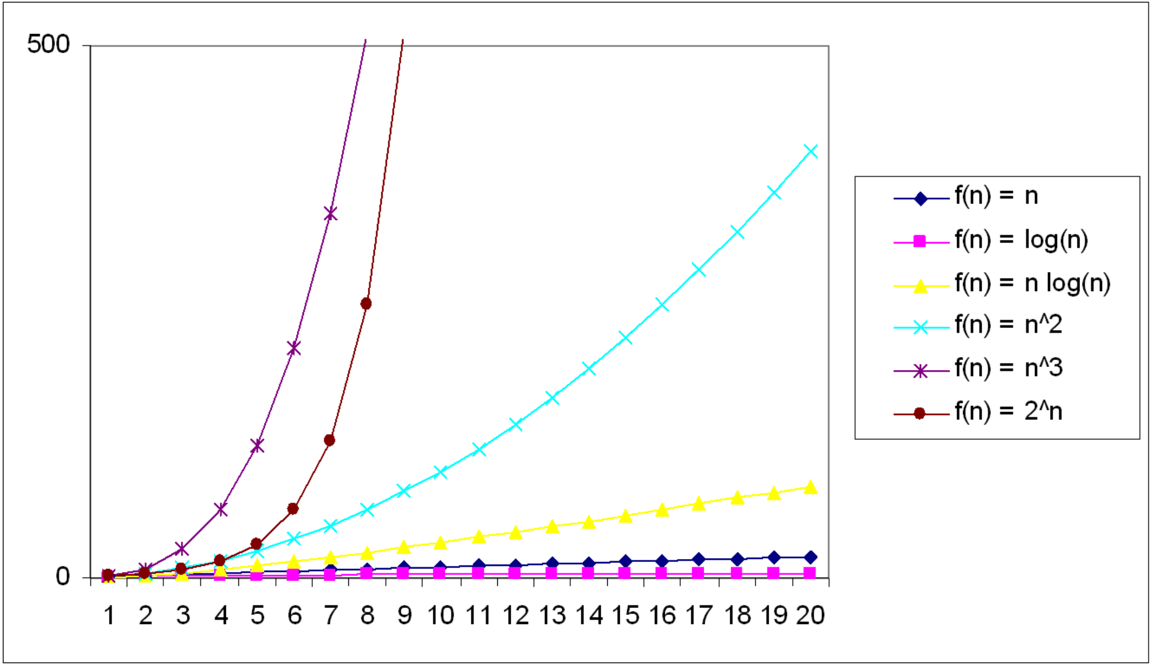
sorting an array by comparing elements

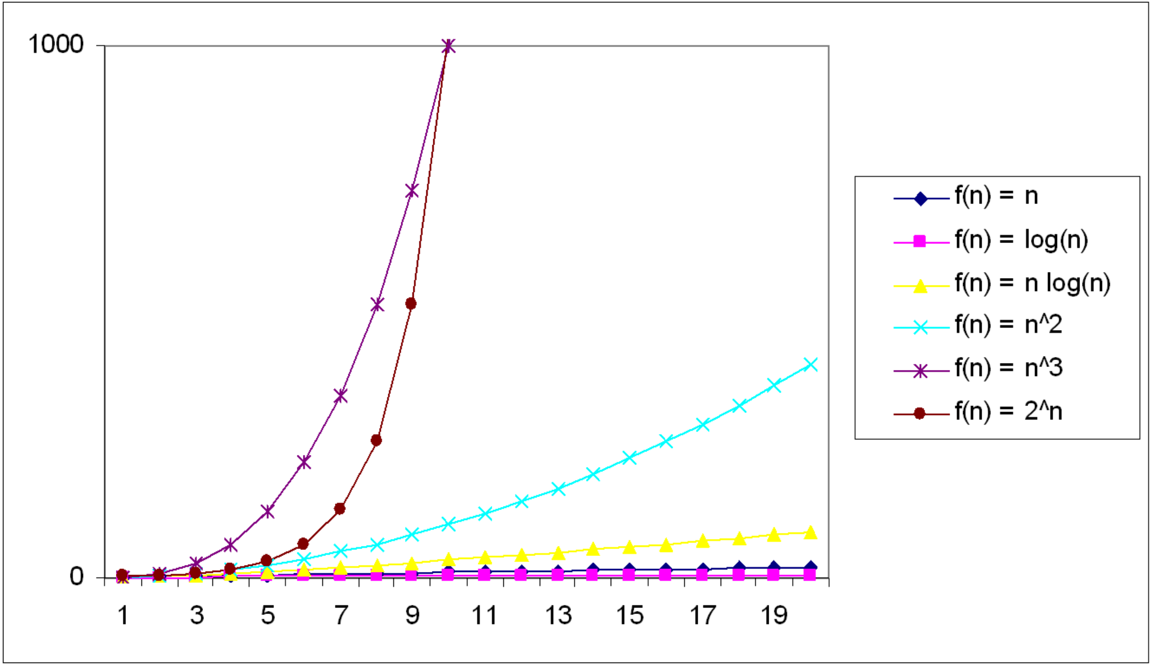
$\Theta(n^2)$ — quadratic

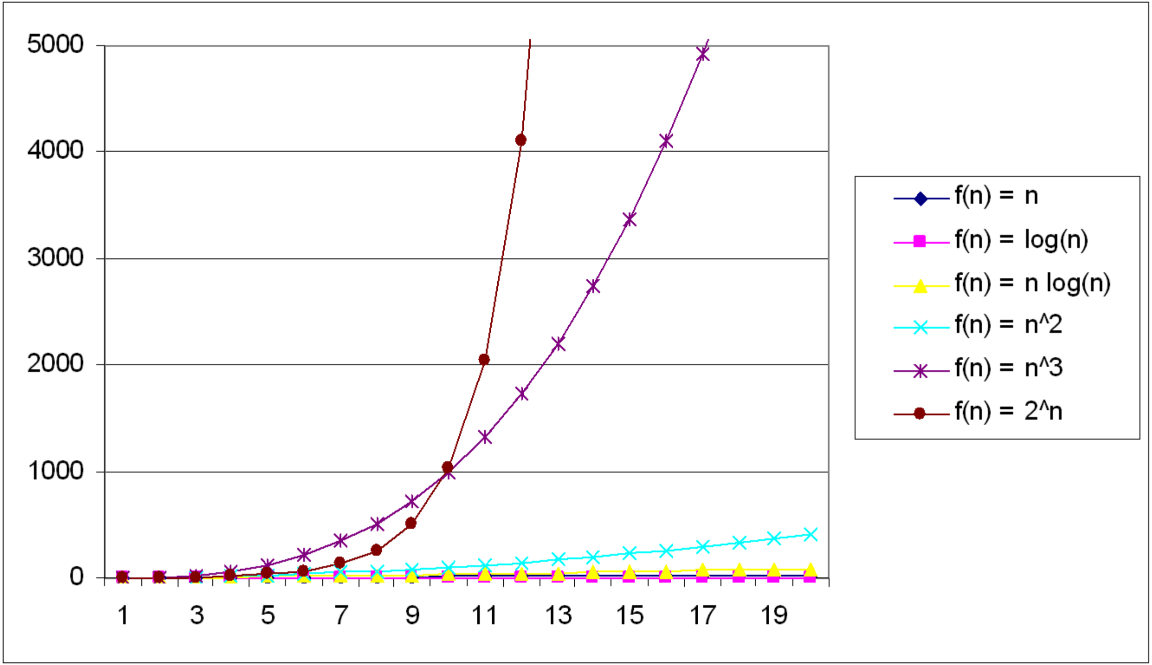
$\Theta(n^3)$ — cubic

$\Theta(2^n)$, $\Theta(c^n)$ — exponential

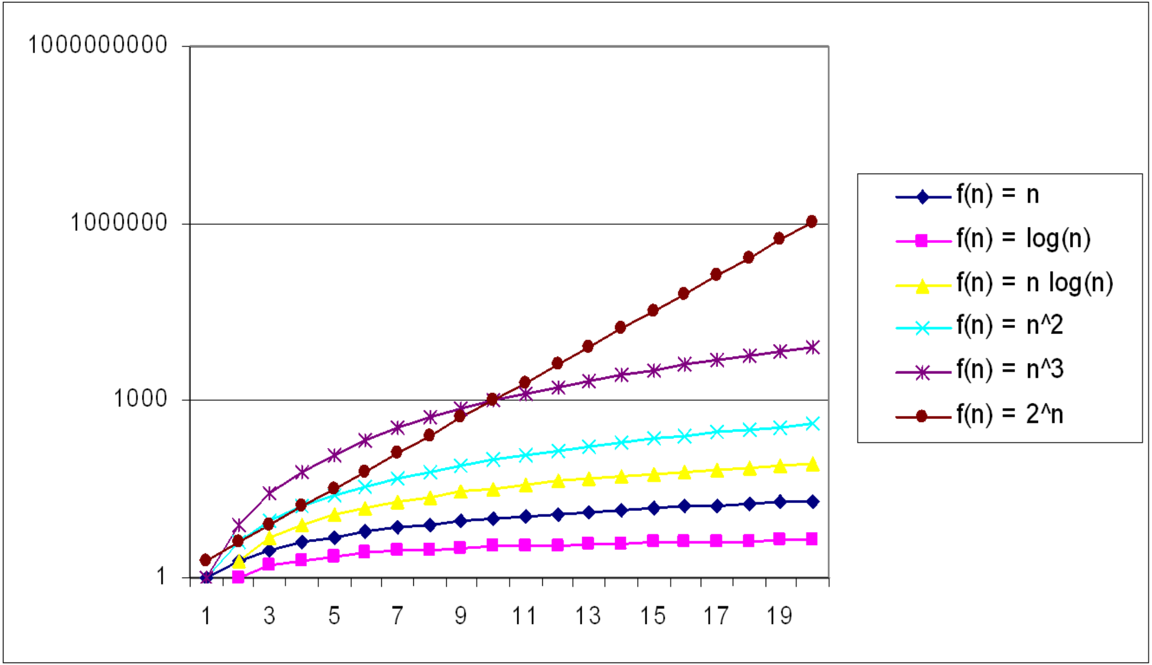

250
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
f(n) = n
f(n) = log(n)
f(n) = n log(n)
f(n) = n^2
f(n) = n^3
f(n) = 2^n


500
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
f(n) = n
f(n) = log(n)
f(n) = n log(n)
f(n) = n^2
f(n) = n^3
f(n) = 2^n


1000
0
1
3
5
7
9
11
13
15
17
19
f(n) = n
f(n) = log(n)
f(n) = n log(n)
f(n) = n^2
f(n) = n^3
f(n) = 2^n



5000
4000
3000
2000
1000
0
1
3
5
7
9
11
13
15
17
19
f(n) = n
f(n) = log(n)
f(n) = n log(n)
f(n) = n^2
f(n) = n^3
f(n) = 2^n



1000000000
1000000
1000
1
1
3
5
7
9
11
13
15
17
19
f(n) = n
f(n) = log(n)
f(n) = n log(n)
f(n) = n^2
f(n) = n^3
f(n) = 2^n

big-oh rules of thumb (1)

```
for (int i = 0; i < N; ++i)
    foo();
```

runtime $\in \Theta(N \times (\text{runtime of } \texttt{foo}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        bar();
```

runtime $\in \Theta(N \times (M \times \text{runtime of } \texttt{bar}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < i; ++j)
        foo();
```

runtime $\in \Theta\left(\sum_{i=0}^{N} i \times \text{runtime of } \texttt{foo}\right) = \Theta(N^2 \cdot \text{runtime of } \texttt{foo})$

big-oh rules of thumb (1)

```
for (int i = 0; i < N; ++i)
    foo();
```

runtime $\in \Theta(N \times (\text{runtime of } \texttt{foo}))$

time to increment i?
"constant factor"
ignored by Θ

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        bar();
```

runtime $\in \Theta(N \times (M \times \text{runtime of } \texttt{bar}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < i; ++j)
        foo();
```

$$\text{runtime} \in \Theta\left(\sum_{i=0}^{N} i \times \text{runtime of } \texttt{foo}\right) = \Theta(N^2 \cdot \text{runtime of } \texttt{foo})$$

big-oh rules of thumb (1)

nested loops — work inside out
find time of inner loop (“foo”)
multiply by iterations of outer loop

```
for (int i = 0; i < N; ++i)
    foo();
```

runtime $\in \Theta(N \times (\text{runtime of foo}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        bar();
```

runtime $\in \Theta(N \times (M \times \text{runtime of bar}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < i; ++j)
        foo();
```

runtime $\in \Theta\left(\sum_{i=0}^{N} i \times \text{runtime of foo}\right) = \Theta(N^2 \cdot \text{runtime of foo})$

big-oh rules of thumb (1)

```
for (int i = 0; i < N; ++i)
    foo();
```

runtime $\in \Theta(N \times (\text{runtime of foo}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        bar();
```

runtime $\in \Theta(N \times (M \times \text{runtime of bar}))$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < i; ++j)
        foo();
```

at least $N/2$ iterations with
at least $N/2$ calls to foo
$\implies N/2 \cdot N/2 = N^2/4$
also $\leq N \cdot N = N^2$ calls
$\implies$ # calls to foo is $\Theta(N^2)$

runtime $\in \Theta\left(\sum_{i=0}^{N} i \times \text{runtime of foo}\right) = \Theta(N^2 \cdot \text{runtime of foo})$

big-oh rules of thumb (2)

```
foo();
bar();
```

runtime = runtime of foo + runtime of bar
$\in \Theta(\max\{\texttt{foo}\text{ runtime}, \texttt{bar}\text{ runtime}\})$

```
if (quux()) {
    foo();
} else {
    bar();
}
```

runtime $\approx$ runtime of quux + $\max$(runtime of foo, runtime of bar)
(max because we measure the worst-case)

$\Theta(1)$: constant time

constant time ($\Theta(1)$ time) — runtime does not depend on input

accessing an array element

linked list insert/delete (at known end)

getting a vector's size

...

is that really constant time

is getting vector's size really constant time?

vector stores its size, but, for, e.g. $N = 2^{10000}$, the size itself is huge

our *usual* assumption:

- treat "sensible" integer arithmetic as constant time
- (anything we'd keep in a `long` or smaller variable in practice?)

can do other analysis, but uncommon

- e.g. "bit complexity" — number of single bit operations

$\Theta(\log n)$: logarithmic time

binary search of sorted array

search space cut in half each iteration — $\lceil \log_2 N \rceil$ iterations

balanced tree search/insert

height of tree (somehow) gaurenteed to be $\Theta(\log N)$

$\Theta(n)$: linear

constant # operations/element

printing a list

search in unsorted array

search in linked list

doubling the size of a vector

unbalanced binary search tree find/insert

$\Theta(n \log n)$: log-linear

fast comparison-based sorting

merge sort, heap sort, ...

quicksort *if pivot choices are good*

inserting n elements into a balanced tree

$\Theta(n^2)$: quadratic

slow comparison-based sorting

insertion sort, bubble sort, selection sort, ...

quicksort *if pivot choices are bad*

most doubly nested for loops that go up to n

$\Theta(2^{n^c})$, $c \geq 1$: exponential

n-bit solution; try every 2^n of the possiblities

crack a combination lock by trying every possiblity

finding the best move in an $N \times N$ Go game (with Japanese rules)

checking satisfiablity of Boolean expression*

the Traveling Salesman problem*

*known algorithms — maybe can do better?

more?

$\Theta(n^3)$ — find shortest paths between all pairs of n nodes on a fully-connected graph

approx. order $2^{n^{1/3}}$ — best known integer factorization algorithm