Algorithm analysis

Algorithm analysis

We want to predict the performance of our algorithms.

Predicting time in seconds isn't feasible.

- What if you use a different programming language?
- What if you used a better compiler?
- What if you just bought a faster computer?

Algorithm analysis

We want to predict the performance of our algorithms.

Predicting time in seconds isn't feasible.

- What if you use a different programming language?
- What if you used a better compiler?
- What if you just bought a faster computer?
- (And, honestly, it's too difficult.)

Usually, we don't care about exact timings (real-time systems excepted).

Primitive operations

Instead of estimating time in seconds, we estimate the number of primitive operations:

- variable assignments
- arithmetic operations
- essentially, CPU instructions

We still call this “running time”.

Primitive operations

Instead of estimating time in seconds, we estimate the number of primitive operations:

- variable assignments
- arithmetic operations
- essentially, CPU instructions

We still call this "running time".

But what if you buy a computer with a different CPU instruction set?

It doesn't matter, because we'll abstract away from this level of detail.

Scalability

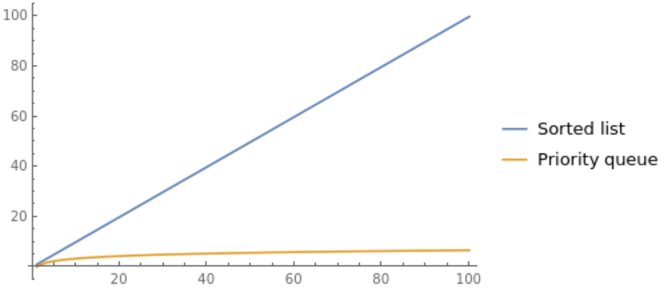
We want to know how algorithms' performance scales as the input gets large:

- Express running times as functions of input size, n.
- e.g., "Inserting into a sorted list takes time proportional to n."
- e.g., "... into a priority queue takes time proportional to $\log_2(n)$."

Scalability

We want to know how algorithms' performance scales as the input gets large:

- Express running times as functions of input size, n.
- e.g., "Inserting into a sorted list takes time proportional to n."
- e.g., "... into a priority queue takes time proportional to $\log_2(n)$."



Asymptotic behaviour

We're interested in **asymptotic** performance: large inputs, as $n \to \infty$.

> Suppress constant factors and lower-order terms.
>
> — Tim Roughgarden, *Algorithms Illuminated, Part I*.

Ignoring constant factors

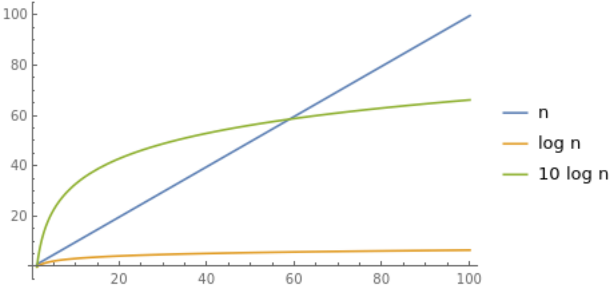
"The number of steps to insert an item into a sorted list is proportional to the list's length" – What's the constant of proportionality?

- Depends on hardware, compiler, etc.
- Exactly what we're trying to abstract away.
- It's probably a small number (like 10, not a million) so it doesn't matter much.

Ignoring constant factors

"The number of steps to insert an item into a sorted list is proportional to the list's length" – What's the constant of proportionality?

- Depends on hardware, compiler, etc.
- Exactly what we're trying to abstract away.
- It's probably a small number (like 10, not a million) so it doesn't matter much.



Can we really ignore constant factors?

Mostly.

When choosing between fundamentally different approaches

- difference between the “underlying functions” is the big deal.
- e.g., linked list vs priority queue.

Can we really ignore constant factors?

Mostly.

When choosing between fundamentally different approaches

- difference between the “underlying functions” is the big deal.
- e.g., linked list vs priority queue.

When Choosing between similar approaches:

- constant factors can be important
- e.g., implementing stacks as arrays vs lists.

Can we really ignore constant factors?

Mostly.

When choosing between fundamentally different approaches

- difference between the "underlying functions" is the big deal.
- e.g., linked list vs priority queue.

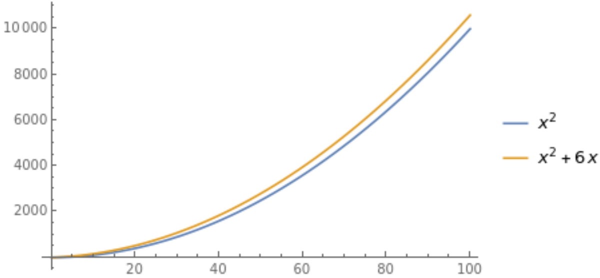
When Choosing between similar approaches:

- constant factors can be important
- e.g., implementing stacks as arrays vs lists.

When writing code: twice as fast is twice as fast.

Ignoring lower-order terms

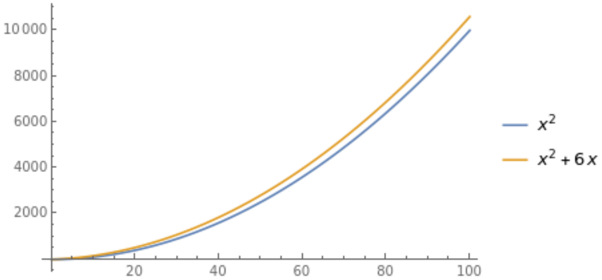
Lower-order terms are ones that become insignificant as $n \to \infty$.



- The two curves are basically the same.

Ignoring lower-order terms

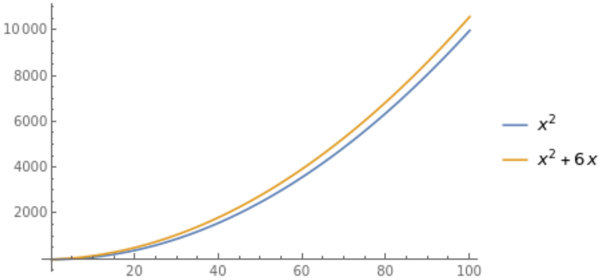
Lower-order terms are ones that become insignificant as $n \to \infty$.



- The two curves are basically the same.
- For $x = 10$, $6x$ is 40% of $x^2 + 6x$.
- For $x = 100$, it's 6%; for $x = 1000$, it's 0.6%.

Ignoring lower-order terms

Lower-order terms are ones that become insignificant as $n \to \infty$.



- The two curves are basically the same.
- For $x = 10$, $6x$ is 40% of $x^2 + 6x$.
- For $x = 100$, it's 6%; for $x = 1000$, it's 0.6%.
- For $x > 6$, $x^2 + 6x < 2x^2$ and we already agreed to ignore constant factors.

Can we really ignore lower-order terms?

Usually, yes.

- For small inputs, any algorithm will do.
- It's large inputs that need us to be smart.
- Lower-order terms are negligible for large inputs.

Asymptotic notation

We want a system for comparing functions while ignoring constant factors and lower-order terms.

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = O(g(n))$ if there are constants n_0 and c such that, for all $n \geq n_0$, $f(n) \leq c\, g(n)$.

Asymptotic notation

We want a system for comparing functions while ignoring constant factors and lower-order terms.

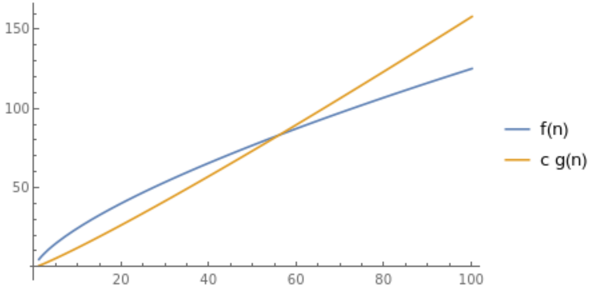
Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = O(g(n))$ if there are constants n_0 and c such that, for all $n \geq n_0$, $f(n) \leq c\, g(n)$.

We say "f is big-O of g".

"Constant" means that n_0 and c cannot depend on n in any way.

Big-O, visually



Think of $f = O(g)$ as "f is sort-of less than g."

Big-O examples (1)

$n^2 + 6n = O(n^2)$.

Big-O examples (1)

$n^2 + 6n = O(n^2)$.

For all $n \geq 6$, $6n \leq n^2$ so $n^2 + 6n \leq 2n^2$.

Big-O examples (1)

$n^2 + 6n = O(n^2)$.

For all $n \geq 6$, $6n \leq n^2$ so $n^2 + 6n \leq 2n^2$.

So take $n_0 = 6$ and $c = 2$.

Big-O examples (2)

$2n^2 + 6n = O(n^2)$.

Big-O examples (2)

$2n^2 + 6n = O(n^2)$.

For all $n \geq 6$, $6n \leq n^2$ so $2n^2 + 6n \leq 3n^2$.

Big-O examples (2)

$2n^2 + 6n = O(n^2)$.

For all $n \geq 6$, $6n \leq n^2$ so $2n^2 + 6n \leq 3n^2$.

So take $n_0 = 6$ and $c = 3$.

Big-O examples (3)

For any constants k, and $a_0, \ldots, a_k$,
let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$.

E.g., $p(n) = 5n^4 - 16n^3 + 0n^2 + 3n - 1$.

Claim: $p(n) = O(n^k)$.

Big-O examples (3)

For any constants k, and $a_0, \ldots, a_k$,
let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$.

E.g., $p(n) = 5n^4 - 16n^3 + 0n^2 + 3n - 1$.

Claim: $p(n) = O(n^k)$.

First, we have $p(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0|$.

Big-O examples (3)

For any constants k, and $a_0, \ldots, a_k$,
let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$.

E.g., $p(n) = 5n^4 - 16n^3 + 0n^2 + 3n - 1$.

Claim: $p(n) = O(n^k)$.

First, we have $p(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0|$.

For all $n \geq 1$, $1 \leq n \leq n^2 \leq \cdots \leq n^{k-1} \leq n^k$.

Big-O examples (3)

For any constants k, and $a_0, \ldots, a_k$,
let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$.

E.g., $p(n) = 5n^4 - 16n^3 + 0n^2 + 3n - 1$.

Claim: $p(n) = O(n^k)$.

First, we have $p(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0|$.

For all $n \geq 1$, $1 \leq n \leq n^2 \leq \cdots \leq n^{k-1} \leq n^k$.

Therefore, $p(n) \leq (|a_k| + \cdots + |a_0|) n^k$.

Big-O examples (3)

For any constants k, and $a_0, \ldots, a_k$,
let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$.

E.g., $p(n) = 5n^4 - 16n^3 + 0n^2 + 3n - 1$.

Claim: $p(n) = O(n^k)$.

First, we have $p(n) \leq |a_k|n^k + |a_{k-1}|n^{k-1} + \cdots + |a_1|n + |a_0|$.

For all $n \geq 1$, $1 \leq n \leq n^2 \leq \cdots \leq n^{k-1} \leq n^k$.

Therefore, $p(n) \leq (|a_k| + \cdots + |a_0|)n^k$.

So take $n_0 = 1$ and $c = |a_k| + \cdots + |a_0|$.

Big-O examples (3)

For any constants k, and $a_0, \ldots, a_k$,
let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$.

E.g., $p(n) = 5n^4 - 16n^3 + 0n^2 + 3n - 1$.

Claim: $p(n) = O(n^k)$.

First, we have $p(n) \leq |a_k|n^k + |a_{k-1}|n^{k-1} + \cdots + |a_1|n + |a_0|$.

For all $n \geq 1$, $1 \leq n \leq n^2 \leq \cdots \leq n^{k-1} \leq n^k$.

Therefore, $p(n) \leq (|a_k| + \cdots + |a_0|)n^k$.

So take $n_0 = 1$ and $c = |a_k| + \cdots + |a_0|$.

Summary. Any polynomial is big-O of its leading term.

Big-O is ignoring constant factors and lower-order terms.

Big-O examples (4)

$n^{k+1} \neq O(n^k)$.

Big-O examples (4)

$n^{k+1} \neq O(n^k)$.

For all c and all $n \geq c$, $n^{k+1} \geq c\, n^k$.

So we cannot find c and n_0 such that $n^{k+1} \leq c\, n^k$ for all $n \geq n_0$.

Properties of big-O

Big-O behaves a lot like $\leq$ on numbers:

- $f(n) = O(f(n))$ for all functions f.
- $f(n) + g(n) = O(\max\{f(n), g(n)\})$.
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(\log_b n)} = n$.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(\log_b n)} = n$.
- $\log_a n = (\log_a b) \log_b n = O(\log_b n)$.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(\log_b n)} = n$.
- $\log_a n = (\log_a b) \log_b n = O(\log_b n)$.
- Big-O doesn't care about base; we can write "$O(\log n)$" without specifying the base.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(\log_b n)} = n$.
- $\log_a n = (\log_a b)\,\log_b n = O(\log_b n)$.
- Big-O doesn't care about base; we can write "$O(\log n)$" without specifying the base.
- $\log(m\,n) = (\log m) + (\log n)$ so $\log(n^2) = 2\log n = O(\log n)$.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(log_b n)} = n$.
- $\log_a n = (\log_a b) \log_b n = O(\log_b n)$.
- Big-O doesn't care about base; we can write "$O(\log n)$" without specifying the base.
- $\log(m\,n) = (\log m) + (\log n)$ so $\log(n^2) = 2 \log n = O(\log n)$.
- But $(\log n)^2 \neq O(\log n)$, just as $n^2 \neq O(n)$.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(log_b n)} = n$.
- $\log_a n = (\log_a b) \log_b n = O(\log_b n)$.
- Big-O doesn't care about base; we can write "$O(\log n)$" without specifying the base.
- $\log(m\,n) = (\log m) + (\log n)$ so $\log(n^2) = 2 \log n = O(\log n)$.
- But $(\log n)^2 \neq O(\log n)$, just as $n^2 \neq O(n)$.
- $\log n = O(n^k)$ for any $k > 0$, even non-integer k.

Big-O and logarithms

- The base-b logarithm of n is $\log_b n$, the unique number such that $b^{(log_b n)} = n$.
- $\log_a n = (\log_a b) \log_b n = O(\log_b n)$.
- Big-O doesn't care about base; we can write "$O(\log n)$" without specifying the base.
- $\log(m\,n) = (\log m) + (\log n)$ so $\log(n^2) = 2\log n = O(\log n)$.
- But $(\log n)^2 \neq O(\log n)$, just as $n^2 \neq O(n)$.
- $\log n = O(n^k)$ for any $k > 0$, even non-integer k.
- E.g., $\log n = O(\sqrt{n})$, since $\sqrt{n} = n^{1/2}$.

Big-O and constants

Let $f(n) = k$ be any constant function.

For all $n \geq 1$, $f(n) \leq k \cdot 1$, so $f(n) = O(1)$.

$f(n) = O(1)$ if, and only if, there is a constant c such that $f(n) \leq c$ for all $n \geq 1$.

Big-O cheat sheet

Common functions ordered by big-O ("sort-of less than"):

$$k, \ldots, \log(\log n), \log n, \sqrt{n} = n^{1/2}, n,$$
$$n, n \log n, n^{1.0001}, n^2, n^3, \ldots, 2^n, 2^{n^2}, \ldots$$

Big-O's friends (1)

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = \Omega(g(n))$ if there are constants n_0 and c such that, for all $n \geq n_0$, $f(n) \geq c\, g(n)$.

We say "f is big-Omega of g."

Big-O's friends (1)

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = \Omega(g(n))$ if there are constants n_0 and c such that, for all $n \geq n_0$, $f(n) \geq c\, g(n)$.

We say "f is big-Omega of g."

Equivalently, $f(n) = \Omega(g(n))$ if, and only if, $g(n) = O(f(n))$.

Big-O's friends (1)

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = \Omega(g(n))$ if there are constants n_0 and c such that, for all $n \geq n_0$, $f(n) \geq c\, g(n)$.

We say "f is big-Omega of g."

Equivalently, $f(n) = \Omega(g(n))$ if, and only if, $g(n) = O(f(n))$.

So big-Ω is "kind-of greater than".

Big-O's friends (2)

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = \Theta(g(n))$ if there are constants n_0, c and d such that, for all $n \geq n_0$, $c\,g(n) \leq f(n) \leq d\,g(n)$.

We say "f is big-Theta of g."

Big-O's friends (2)

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = \Theta(g(n))$ if there are constants n_0, c and d such that, for all $n \geq n_0$, $c\,g(n) \leq f(n) \leq d\,g(n)$.

We say "f is big-Theta of g."

Equivalently, $f(n) = \Theta(g(n))$ if, and only if, $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Big-O's friends (2)

Definition

Let $f(n)$ and $g(n)$ be functions $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$.
We write $f(n) = \Theta(g(n))$ if there are constants n_0, c and d such that, for all $n \geq n_0$, $c\, g(n) \leq f(n) \leq d\, g(n)$.

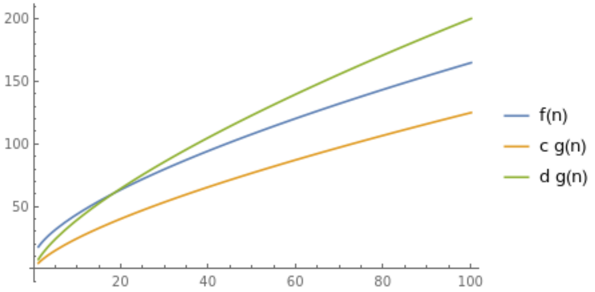
We say "f is big-Theta of g."

Equivalently, $f(n) = \Theta(g(n))$ if, and only if, $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

So big-Θ is "approximately proportional to".

Big-Θ, visually

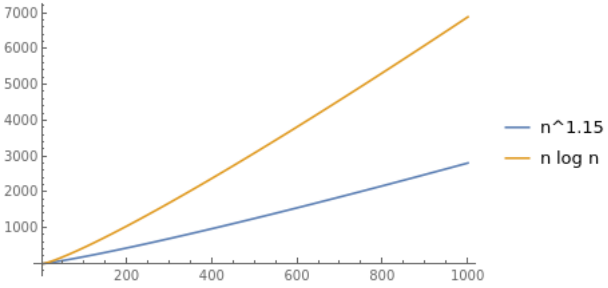
Here, $f(n) = \Theta(g(n))$.



Big-O by plotting graphs

Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

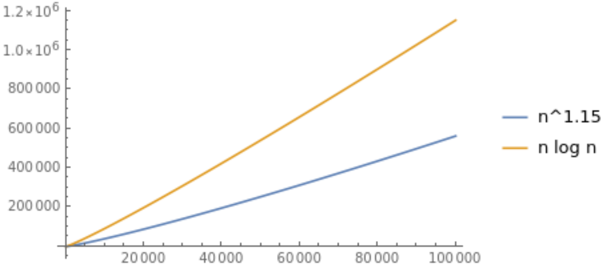
Works for most functions you'll come across in this course, but be careful!



Big-O by plotting graphs

Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

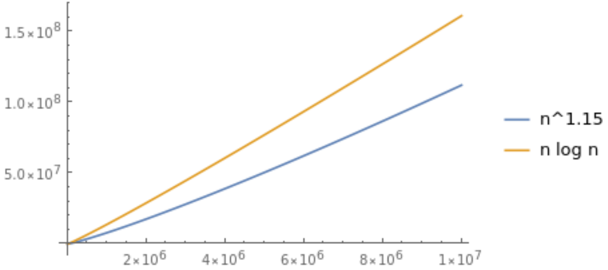
Works for most functions you'll come across in this course, but be careful!



Big-O by plotting graphs

Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

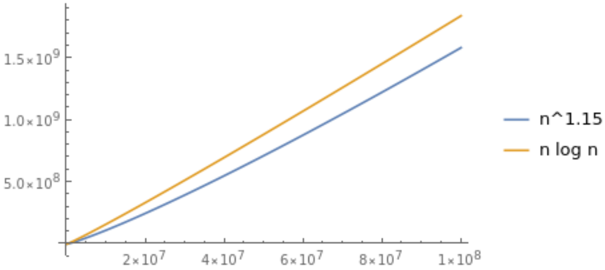
Works for most functions you'll come across in this course, but be careful!



Big-O by plotting graphs

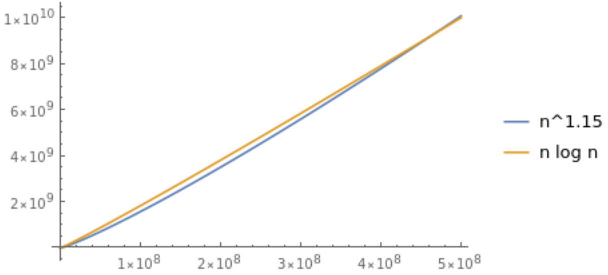
Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

Works for most functions you'll come across in this course, but be careful!



Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

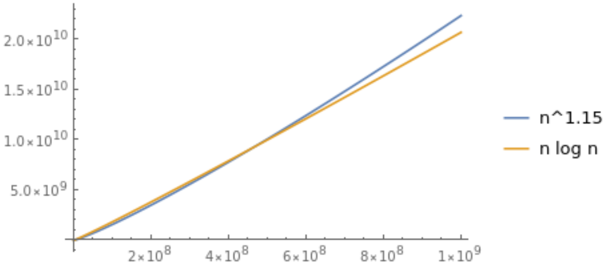
Works for most functions you'll come across in this course, but be careful!



Big-O by plotting graphs

Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

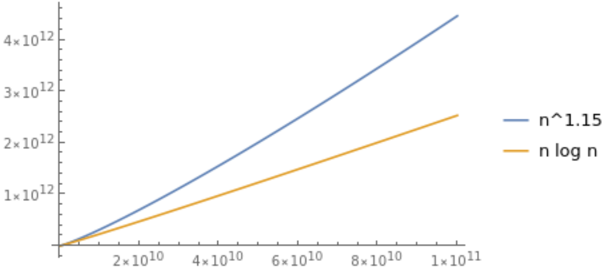
Works for most functions you'll come across in this course, but be careful!



Big-O by plotting graphs

Plotting graphs of $f(n)$ and $g(n)$ is often a good way to see if $f(n) = O(g(n))$.

Works for most functions you'll come across in this course, but be careful!



Tight bounds

Asymptotic notation allows us to bound one function in terms of another.

A bound is **tight** if it can't be simplified or made more precise.

Examples:

- $\pi < 4$ and $\pi > 3$ are tight integer bounds on pi.
- $\pi \leq 4$ and $\pi > 0$ are not tight – but true and may be useful!
- $n^2 + 3n + 4 = O(n^2)$ is tight.
- $n^2 + 3n + 4 = O(n^2 + n)$ and $= O(n^3)$ are not tight.

Pet peeve

There is no such thing as "the big-O of a function", as in "What is the big-O of $n \log n + 3n$?"

This is like asking "What is the integer bigger than π?"

Pet peeve

There is no such thing as "the big-O of a function", as in "What is the big-O of $n \log n + 3n$?"

This is like asking "What is the integer bigger than π?"

There are infinitely many correct answers:

- $4 > \pi$, $5 > \pi$, $76 > \pi$, ...
- $n \log n + 3n = O(n \log n),\ O(n^2),\ O\left(2^{2^n}\right)$, ...

Pet peeve

There is no such thing as "the big-O of a function", as in "What is the big-O of $n \log n + 3n$?"

This is like asking "What is the integer bigger than π?"

There are infinitely many correct answers:

- $4 > \pi$, $5 > \pi$, $76 > \pi$, ...
- $n \log n + 3n = O(n \log n),\ O(n^2),\ O\big(2^{2^n}\big)$, ...

Instead, ask "What is a tight big-O bound for $n \log n + 3n$?"

Not my pet peeve

Some people object vehemently to writing, e.g., "$3n + 4 = O(n)$" and insist on "$3n + 4 \in O(n)$".

Their point:

- $3n + 4$ is a function
- formally, $O(n)$ is a set of functions
- the two things cannot be equal because "they have different types".

Not my pet peeve

Some people object vehemently to writing, e.g., "$3n + 4 = O(n)$" and insist on "$3n + 4 \in O(n)$".

Their point:

- $3n + 4$ is a function
- formally, $O(n)$ is a set of functions
- the two things cannot be equal because "they have different types".

This is true but most people write $=$.

Asymptotic notation – summary

"Suppress constant factors and lower-order terms."

Asymptotic notation – summary

"Suppress constant factors and lower-order terms."

$f(n) = \Theta(g(n))$

- f is approximately proportional to g;
- For all large n, $c\,g(n) \le f(n) \le d\,g(n)$.

Asymptotic notation – summary

"Suppress constant factors and lower-order terms."

$f(n) = \Theta(g(n))$

- f is approximately proportional to g;
- For all large n, $c\,g(n) \leq f(n) \leq d\,g(n)$.

$f(n) = O(g(n))$

- f is at most approximately proportional to g;
- For all large n, $f(n) \leq c\,g(n)$.

Asymptotic notation – summary

"Suppress constant factors and lower-order terms."

$f(n) = \Theta(g(n))$

- f is approximately proportional to g;
- For all large n, $c\,g(n) \leq f(n) \leq d\,g(n)$.

$f(n) = O(g(n))$

- f is at most approximately proportional to g;
- For all large n, $f(n) \leq c\,g(n)$.

$f(n) = \Omega(g(n))$

- f is at least approximately proportional to g;
- For all large n, $f(n) \geq c\,g(n)$.

Practical program analysis

Consequences of running time bounds

If your program's input has length n and it runs in time...

- $\Theta(n)$ then doubling n doubles the time taken.

Consequences of running time bounds

If your program's input has length n and it runs in time...

- $\Theta(n)$ then doubling n doubles the time taken.
- $\Theta(n^2)$ then doubling n quadruples the time taken.
- $\Theta(n^k)$ then doubling n multiplies the time by 2^k.

Consequences of running time bounds

If your program's input has length n and it runs in time...

- $\Theta(n)$ then doubling n doubles the time taken.
- $\Theta(n^2)$ then doubling n quadruples the time taken.
- $\Theta(n^k)$ then doubling n multiplies the time by 2^k.
- $\Theta(\log n)$ then doubling n adds one time unit.

Consequences of running time bounds

If your program's input has length n and it runs in time...

- $\Theta(n)$ then doubling n doubles the time taken.
- $\Theta(n^2)$ then doubling n quadruples the time taken.
- $\Theta(n^k)$ then doubling n multiplies the time by 2^k.
- $\Theta(\log n)$ then doubling n adds one time unit.
- $\Theta(n \log n)$ then doubling n slightly more than doubles the time.

Consequences of running time bounds

If your program's input has length n and it runs in time...

- $\Theta(n)$ then doubling n doubles the time taken.
- $\Theta(n^2)$ then doubling n quadruples the time taken.
- $\Theta(n^k)$ then doubling n multiplies the time by 2^k.
- $\Theta(\log n)$ then doubling n adds one time unit.
- $\Theta(n \log n)$ then doubling n slightly more than doubles the time.
- $\Theta(2^n)$ then adding 1 to n doubles the time. (Eek!)

Analysis of Java programs (1)

Simple statements take time $O(1)$

- assignments
- evaluating expressions without method calls
- jumps (e.g., `break`, skipping past an `if` when the condition is false)

Analysis of Java programs (1)

Simple statements take time $O(1)$

- assignments
- evaluating expressions without method calls
- jumps (e.g., `break`, skipping past an `if` when the condition is false)

Method calls:

- Analyze the method to find it runs in time $O(f(n))$.
- Jumping to the method and back takes time $O(1)$.
- Total is $O(1 + f(n)) = O(f(n))$;

Analysis of Java programs (2)

```
if ([condition]) {
    [block 1]
} else {
    [block 2]
}
```

Analysis of Java programs (2)

```
if ([condition]) {
    [block 1]
} else {
    [block 2]
}
```

- Analyze the condition and two blocks to find they run in time $O(f_{\mathrm{c}})$, $O(g_1)$ and $O(g_2)$.

Analysis of Java programs (2)

```
if ([condition]) {
    [block 1]
} else {
    [block 2]
}
```

- Analyze the condition and two blocks to find they run in time $O(f_c)$, $O(g_1)$ and $O(g_2)$.
- Total is $O(1 + f_c + \max\{g_1, g_2\}) = O(\max\{f_c, g_1, g_2\})$.

Analysis of Java programs (3)

```
while ([condition]) {
    [block]
}
```

Analysis of Java programs (3)

```
while ([condition]) {
    [block]
}
```

- Analyze `[condition]` and `[block]` to find they run in time $O(f_{\mathrm{c}})$, $O(g)$.

Analysis of Java programs (3)

```
while ([condition]) {
    [block]
}
```

- Analyze `[condition]` and `[block]` to find they run in time $O(f_{\mathrm{c}})$, $O(g)$.
- Analyze the loop to find it runs $O(t(n))$ times.

Analysis of Java programs (3)

```
while ([condition]) {
    [block]
}
```

- Analyze `[condition]` and `[block]` to find they run in time $O(f_{\mathrm{c}})$, $O(g)$.
- Analyze the loop to find it runs $O(t(n))$ times.
- Total is $O(1 + (1 + g(n))t(n)) = O(g(n)t(n))$.

Analysis of Java programs (4)

```
for ([initializer]; [condition]; [increment]) {
    [block]
}
```

is equivalent to

```
[initializer]
while ([condition]) {
    [block]
    [increment]
}
```

Analysis of Java programs (5)

A specific, common case of `for` loops:

```
for (int i = 0; i < n; i++) {
    [block]
}
```

- We know the loop runs n times.
- Total cost is n times the cost of executing `[block]`.

Example analysis

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < n; j++)
3          if (A[i] == B[j])
4              duplicates++;
```

- Line 4 runs in time $O(1)$ (simple statement).

```
1 for (int i = 0; i < n; i++)
2     for (int j = 0; j < n; j++)
3         if (A[i] == B[j])
4             duplicates++;
```

- Line 4 runs in time $O(1)$ (simple statement).
- Lines 3–4 run in time $O(1)$ (if with simple condition and simple statement).

Example analysis

```
1 for (int i = 0; i < n; i++)
2     for (int j = 0; j < n; j++)
3         if (A[i] == B[j])
4             duplicates++;
```

- Line 4 runs in time $O(1)$ (simple statement).
- Lines 3–4 run in time $O(1)$ (if with simple condition and simple statement).
- `for (j...)` runs lines 3–4 n times: takes time $O(n \times 1) = O(n)$.

Example analysis

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < n; j++)
3          if (A[i] == B[j])
4              duplicates++;
```

- Line 4 runs in time $O(1)$ (simple statement).
- Lines 3–4 run in time $O(1)$ (if with simple condition and simple statement).
- `for (j...)` runs lines 3–4 n times: takes time $O(n \times 1) = O(n)$.
- `for (i...)` runs lines 2–4 n times: takes time $O(n \times n) = O(n^2)$.