Multiplication

What’s the best way to multiply two numbers?

Multiplication

Input 2 non-negative numbers, x and y (n digits each)

Output the product x · y

$$\begin{array}{r} 5678 \\ \times\ 1234 \\ \hline \mathbf{7006652} \end{array}$$

Multiplication

$$\begin{array}{r} 45 \\ \text{x } 63 \\ \hline 135 \\ 2700 \\ \hline \mathbf{2835} \end{array}$$

Multiplication

Algorithm description (very very informal)

1. Compute partial products (using multiplication & “carries” for digit overflows)

2. Add all, properly shifted, partial products together

$$\begin{array}{r} 45 \\ \text{x}\ 63 \\ \hline 135 \\ 2700 \\ \hline \mathbf{2835} \end{array}$$

Multiplication

45123456678093420581217332421

x 63782384198347750652091236423

Multiplication

$$\overbrace{\begin{array}{r} 45123456678093420581217332421 \\ \text{x}\ 63782384198347750652091236423 \\ \hline \end{array}}^{\mathbf{n}\ \text{digits}}$$

How efficient is this algorithm?

How many single-digit operations are required?

Multiplication

$$\begin{array}{r} \overbrace{\quad 45123456678093420581217332421}^{\mathbf{n}\text{ digits}} \\ \text{x } 63782384198347750652091236423 \\ \hline \end{array}$$

How efficient is this algorithm ?

How many single-digit operations *in the worst case* ?

n partial products: ~$2n^2$ ops

at most n multiplications & n additions per partial product

adding n partial products: ~$2n^2$ ops

a bunch of additions & "carries"

Multiplication

$$\begin{array}{r} \overbrace{\phantom{\text{x }}45123456678093420581217332421}^{\mathbf{n}\text{ digits}} \\ \text{x } 63782384198347750652091236423 \\ \hline \end{array}$$

How efficient is this algorithm ?

How many single-digit operations *in the worst case* ?

n partial products: $\sim 2n^2$ ops

at most n multiplications & n additions per partial product

adding n partial products: $\sim 2n^2$ ops

a bunch of additions & “carries”

$\sim 4n^2$ operations in the worst case

Do better?
What does “better” mean?

Is 1000000n operations better than $4n^2$?

Is $0.000001n^3$ operations better than $4n^2$?

Is $3n^2$ operations better than $4n^2$?

What does “better” mean?

Is 1000000n operations better than $4n^2$?

Is $0.000001n^3$ operations better than $4n^2$?

Is $3n^2$ operations better than $4n^2$?

The answers for the first two depend on what value n is...

$1000000n < 4n^2$ only when n exceeds a certain value (in this case, 250000)

These constant multipliers are too environment-dependent...

An operation could be faster/slower depending on the machine, so $3n^2$ ops on a slow machine might not be “better” than $4n^2$ ops on a faster machine

What does “better” mean?

Asymptotic Analysis

What does “better” mean?

Asymptotic Analysis

Some guiding principles

- we care about how the running time/number of operations *scales* with the size of the input (i.e. the runtime’s *rate of growth*),
- we want some measure of runtime that’s independent of hardware, programming language, memory layout, etc.

What does “better” mean?

Asymptotic Analysis

Some guiding principles

- we care about how the running time/number of operations *scales* with the size of the input (i.e. the runtime’s *rate of growth*),
- we want some measure of runtime that’s independent of hardware, programming language, memory layout, etc.
    - Details like hardware / language / memory / compiler / etc. are important to real world engineers. We want to reason about high-level algorithmic approaches rather than lower-level details.

Asymptotic Analysis (the high level idea)

We'll express the asymptotic runtime of an algorithm using

BIG-O notation

We say Multiplication "runs in time $O(n^2)$"

- Informally, this means that the runtime "*scales like*" n^2
- Discuss the formal definition of Big-O later

Asymptotic Analysis (the high level idea)

We'll Express the asymptotic runtime of an algorithm using

BIG-O notation

- We would say Multiplication "runs in time $O(n^2)$"
  - Informally, this means that the runtime "scales like" n^2
  - We'll discuss the formal definition of Big-O later

THE POINT OF ASYMPTOTIC NOTATION

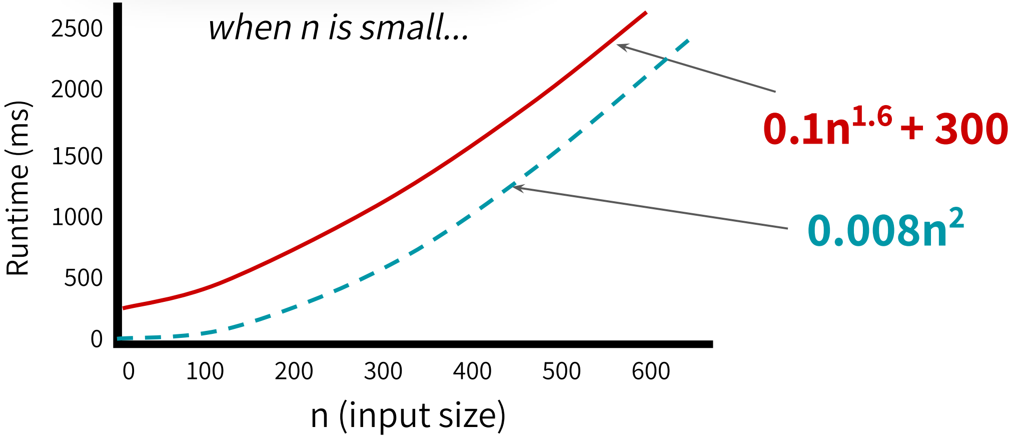
suppress constant factors and lower-order terms

too system dependent *irrelevant for large inputs*

Asymptotic Analysis (the high level idea)

THE POINT OF ASYMPTOTIC NOTATION

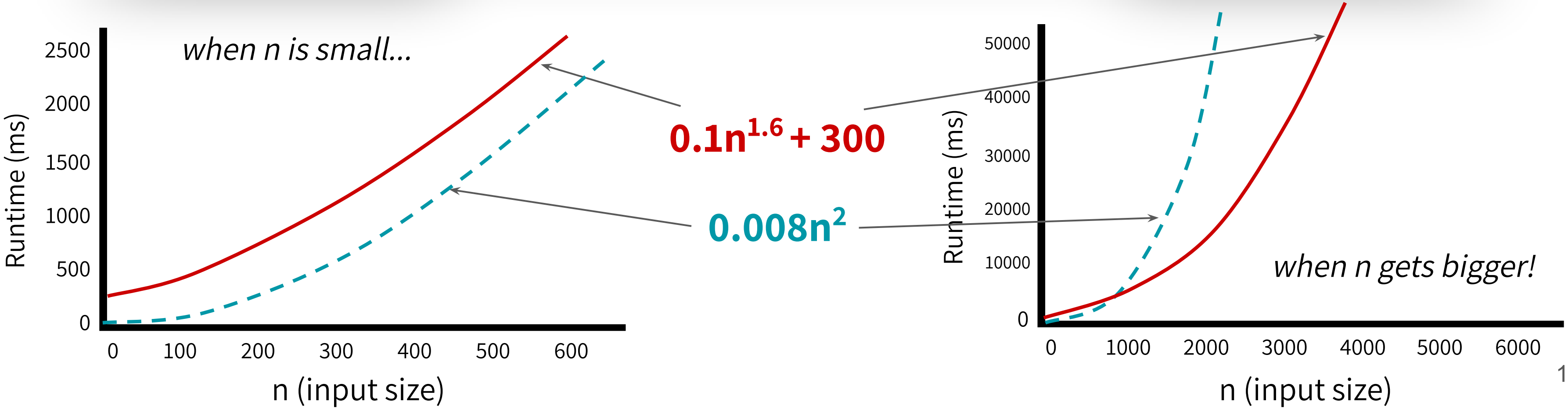
$$\textbf{suppress } \underbrace{\textbf{constant factors}}_{\textit{too system dependent}} \textbf{ and } \underbrace{\textbf{lower-order terms}}_{\textit{irrelevant for large inputs}}$$



Asymptotic Analysis (the high level idea)

THE POINT OF ASYMPTOTIC NOTATION

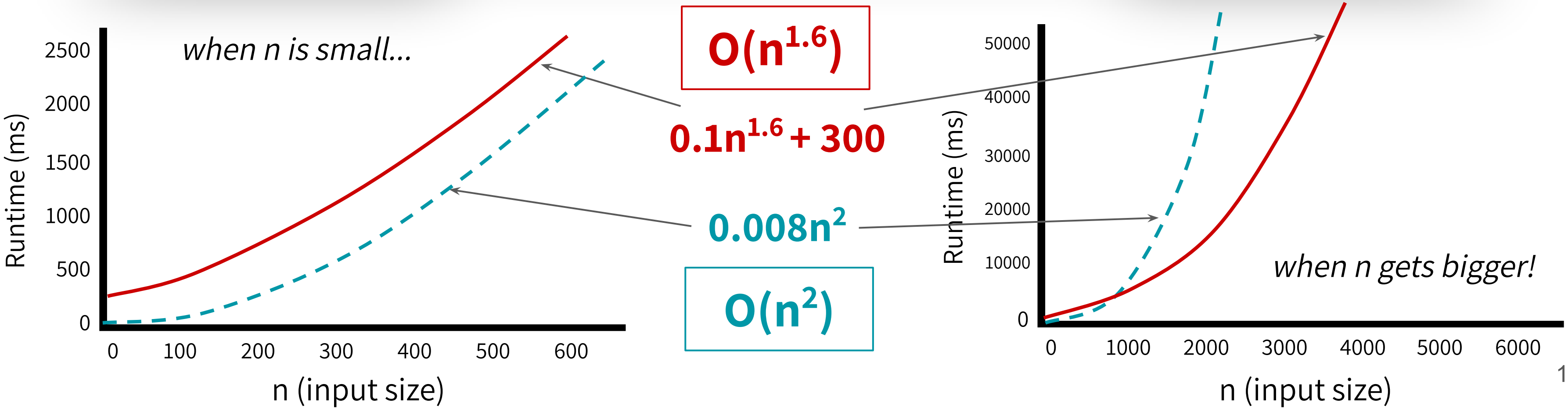
suppress $\underbrace{\text{constant factors}}_{\text{too system dependent}}$ and $\underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$



Asymptotic Analysis (the high level idea)

THE POINT OF ASYMPTOTIC NOTATION

suppress $\underbrace{\textbf{constant factors}}_{\text{too system dependent}}$ **and** $\underbrace{\textbf{lower-order terms}}_{\text{irrelevant for large inputs}}$



$O(n^{1.6})$

$0.1n^{1.6} + 300$

$0.008n^2$

$O(n^2)$

Asymptotic analysis (the high level idea)

To compare algorithm runtimes in this class, we compare their Big-O runtimes

a runtime of $O(n^2)$ is considered “better” than a runtime of $O(n^3)$
a runtime of $O(n^{1.6})$ is considered “better” than a runtime of $O(n^2)$
a runtime of $O(1/n)$ is considered “better” than $O(1)$

Asymptotic Analysis (the high level idea)

To compare algorithm runtimes in this class, we compare their Big-O runtimes

a runtime of $O(n^2)$ is considered "better" than a runtime of $O(n^3)$
a runtime of $O(n^{1.6})$ is considered "better" than a runtime of $O(n^2)$
a runtime of $O(1/n)$ is considered "better" than $O(1)$

Can we multiply n-digit integers faster than $O(n^2)$?

Asymptotic Analysis

Big-O notation, and Big-Ω and Big-Θ

A note on runtime analysis

There are a few different ways to analyze the runtime of an algorithm

Worst-case analysis
What is the runtime of the algorithm on the *worst* possible input?

Best-case analysis
What is the runtime of the algorithm on the *best* possible input?

Average-case analysis
What is the runtime of the algorithm on the *average* input?

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

Write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say “T(n) is O(f(n))”?

English Definition

Visual Perspective

Mathematical Definition

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

Write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say “T(n) is O(f(n))”?

In English

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

Visual Perspective

Mathematical Definition

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

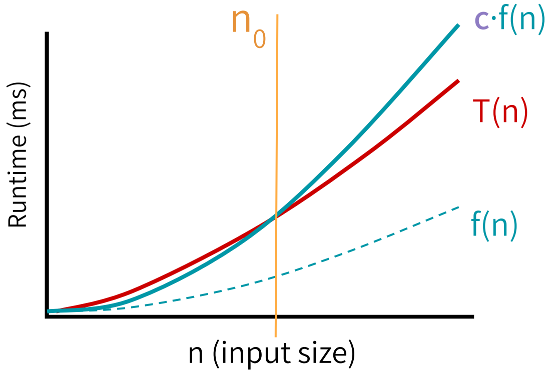
Write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say "T(n) is O(f(n))"?

In English

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

In Pictures



Mathematical Definition

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

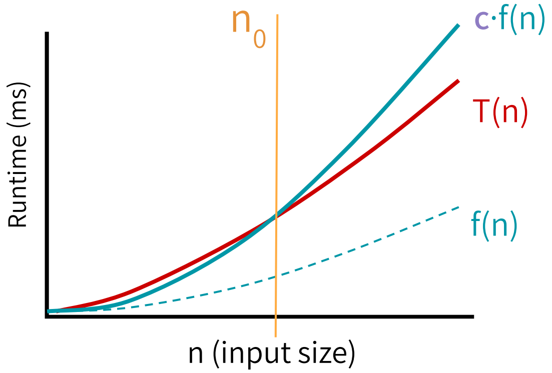
Write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say "T(n) is O(f(n))"?

In English

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

In Pictures



In Math

T(n) = O(f(n))
if and only if
there exists positive
constants
c and n_0 such that *for all*
$n \geq n_0$

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

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

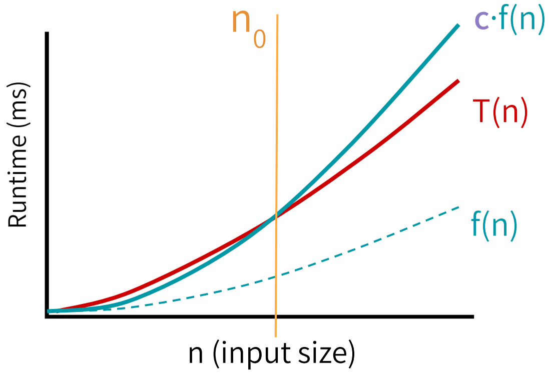
Write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say "T(n) is O(f(n))"?

In English

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

In Pictures



In Math

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$
$$T(n) \leq c \cdot f(n)$$

Big-O Notation

Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

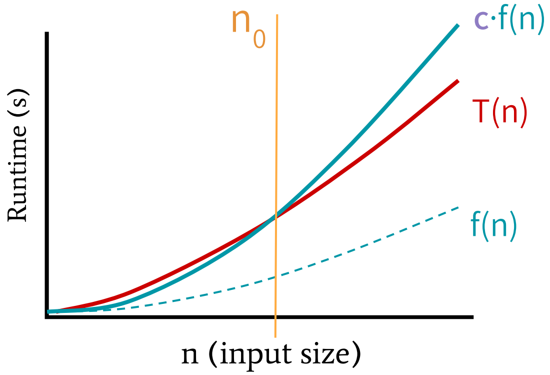
Write $T(n)$ to denote the worst case runtime of an algorithm

What do we mean when we say “$T(n)$ is $O(f(n))$”?

In English

$T(n) = O(f(n))$ if and only if $T(n)$ is *eventually* **upper bounded** by a constant multiple of $f(n)$

In Pictures



In Math

$$T(n) = O(f(n))$$

“for all”

“if and only if” $\Leftrightarrow$

$$\exists\ c\ ,\ n_0 > 0\ \text{ s.t. }\ \forall\ n \geq n_0\ ,$$

$$T(n) \leq c \cdot f(n)$$ “such that”

“there exists”

Proving BIG-O Bounds

If ever asked to formally prove that T(n) is O(f(n)), use the math definition

$$T(n) = O(f(n))$$

$$\Leftrightarrow$$

$$\exists\ c\ ,\ n_0 > 0\ \text{ s.t. }\ \forall\ n \geq n_0\ ,$$

$$\mathbf{T(n) \leq c \cdot f(n)}$$

must be constants, i.e. c & n_0 cannot depend on n

To **prove** T(n) = O(f(n)), you need to announce your c & n_0 up front

Play around with the expressions to find appropriate choices of c & n_0 (positive constants)

Then you can write the proof.

Proving BIG-O Bounds

If ever asked to formally prove that T(n) is O(f(n)), use the math definition

$$T(n) = O(f(n))$$

$$\Leftrightarrow$$

$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$

$$\mathbf{T(n) \leq c \cdot f(n)}$$

must be constants, i.e. c & n_0 cannot depend on n

To **prove** T(n) = O(f(n)), you need to announce your c & n_0 up front

Play around with the expressions to find appropriate choices of c & n_0 (positive constants)

Then you can write the proof.

Let c = __ and n_0 = __.

We will show that **$T(n) \leq c \cdot f(n)$** for all $n \geq n_0$.

Proving BIG-O Bounds

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \ge n_0,$$
$$\mathbf{T(n) \le c \cdot f(n)}$$

Example: Prove that $3n^2 + 5n = O(n^2)$

Let $c = 4$ and $n_0 = 5$. We will now show that $3n^2 + 5n \le c \cdot n^2$ for all $n \ge n_0$.
We know that for any $n \ge n_0$, we have:

$$5 \le n$$
$$5n \le n^2$$
$$3n^2 + 5n \le 4n^2$$

Using our choice of c and n_0, we have successfully shown that $3n^2 + 5n \le c \cdot n^2$ for all $n \ge n_0$. From the definition of Big-O, this proves that $3n^2 + 5n = O(n^2)$. ■

Disproving BIG-O Bounds

If you're ever asked to formally disprove that T(n) is O(f(n)), use **proof by contradiction**

Disproving BIG-O Bounds

If you're ever asked to formally disprove that T(n) is O(f(n)), use **proof by contradiction**

For sake of contradiction, assume that T(n) is O(f(n)).

In other words, assume there does indeed exist a choice of c & n_0 s.t. $\forall$ $n \geq n_0$, $\mathbf{T(n) \leq c \cdot f(n)}$

Disproving BIG-O Bounds

If you're ever asked to formally disprove that T(n) is O(f(n)), use **proof by contradiction**

For sake of contradiction, assume that T(n) is O(f(n)).

In other words, assume there does indeed exist a choice of c & n_0 s.t. $\forall\ n \geq n_0$, **$T(n) \leq c \cdot f(n)$**

↓

Treating c & n_0 as variables, derive a contradiction!

Disproving BIG-O Bounds

If you're ever asked to formally disprove that T(n) is O(f(n)), use **proof by contradiction**

For sake of contradiction, assume that T(n) is O(f(n)).

In other words, assume there does indeed exist a choice of c & n_0 s.t. $\forall\ n \geq n_0$, $T(n) \leq c \cdot f(n)$

↓

Treating c & n_0 as variables, derive a contradiction!

↓

Conclude that the original assumption must be false, so T(n) is *not* O(f(n)).

Disproving BIG-O Bounds

$T(n) = O(f(n))$
$\Leftrightarrow$
$\exists\ c, n_0 > 0$ s.t. $\forall\ n \geq n_0$, $T(n) \leq c \cdot f(n)$

Prove that $3n^2 + 5n$ is *not* $O(n)$.

For sake of contradiction, assume that $3n^2 + 5n$ is $O(n)$. This means that there exists positive constants c & n_0 such that $3n^2 + 5n \leq c \cdot n$ for all $n \geq n_0$.
Then, we would have the following:

$$3n^2 + 5n \leq c \cdot n$$
$$3n + 5 \leq c$$
$$n \leq (c - 5)/3$$

However, since $(c - 5)/3$ is a constant, we've arrived at a contradiction since n cannot be bounded above by a constant for all $n \geq n_0$. For instance, consider $n = n_0 + c$: we see that $n \geq n_0$, but $n > (c - 5)/3$.

Thus, our original assumption was incorrect, which means that $3n^2 + 5n$ is *not* $O(n)$.

BIG-O Examples

$\log_2 n + 15 = O(\log_2 n)$

Polynomials

Say $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree $k \geq 1$.

Then

i. $p(n) = O(n^k)$

ii. $p(n)$ is **not** $O(n^{k-1})$

$6n^3 + n \log_2 n = O(n^3)$

$25 = O(1)$

any constant = O(1)

BIG-O Examples

lower order terms don’t matter!

$$\log_2 n + 15 = O(\log_2 n)$$

Polynomials

Say $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$ is a polynomial of degree $k \geq 1$.

Then:

i. $p(n) = O(n^k)$

ii. $p(n)$ is **not** $O(n^{k-1})$

constant multipliers & lower order terms don’t matter

$$6n^3 + n \log_2 n = O(n^3)$$

$$25 = O(1)$$

any constant = $O(1)$

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say “T(n) is Ω(f(n))”?

English Definition

Pictorial Definition

Mathematical Definition

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say “T(n) is Ω(f(n))”?

English

T(n) = Ω(f(n)) if and only if T(n) is eventually **lower bounded** by a constant multiple of f(n)

Pictorial Definition

Mathematical Definition

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

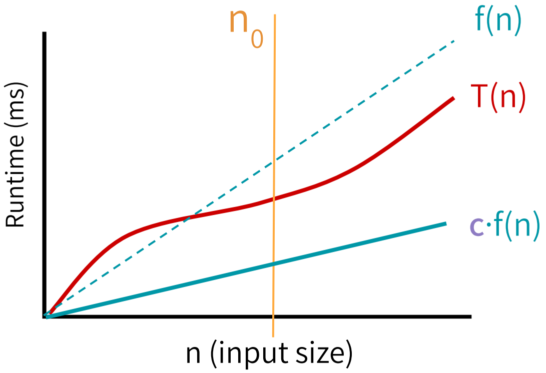
write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say "T(n) is Ω(f(n))"?

In English

T(n) = Ω(f(n)) if and only if T(n) is eventually **lower bounded** by a constant multiple of f(n)

In Pictures



Mathematical Definition

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

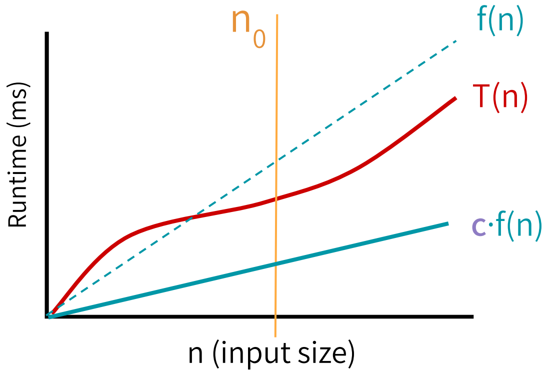
write T(n) to denote the worst case runtime of an algorithm

What do we mean when we say "T(n) is Ω(f(n))"?

In English

T(n) = Ω(f(n)) if and only if T(n) is eventually lower bounded by a constant multiple of f(n)

In Pictures



Maths

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

$$\Leftrightarrow$$

$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$

$$T(n) \geq c \cdot f(n)$$

inequality switched directions!

Big-Θ Notation

We say “T(n) is Θ(f(n))” if and only if both

$$T(n) = O(f(n))$$

&

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

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

$$\Leftrightarrow$$

$$\exists\ c_1, c_2, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$

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

Asymptotic Notation

BOUND	DEFINITION	REPRESENTS
$T(n) = O(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \ge n_0,$ **$T(n) \le c$ □ $f(n)$**	upper bound
$T(n) = \Omega(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \ge n_0,$ **$T(n) \ge c$ □ $f(n)$**	lower bound
$T(n) = \Theta(f(n))$	$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$	tight bound