

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 1

Growth Rates

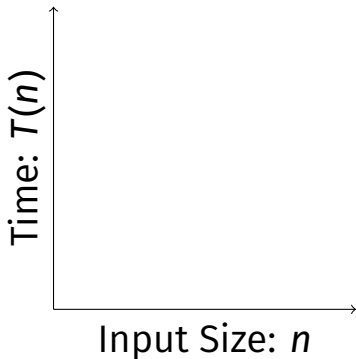
Today in DSC 40B...

- ▶ Formally define Θ , O , Ω notation.
- ▶ Some useful properties.
- ▶ The drawbacks of asymptotic time complexity.
- ▶ Best, worst case time complexities.

Also!

- ▶ There's a set of course notes.
- ▶ Also, a bank of practice problems.
- ▶ See `dsc40b.com`.

Linear vs. Quadratic Scaling



- ▶ $T(n) = \Theta(n)$ means " $T(n)$ grows like n "
- ▶ $T(n) = \Theta(n^2)$ means " $T(n)$ grows like n^2 "

Definition

An algorithm is said to run in **linear time** if $T(n) = \Theta(n)$.

Definition

An algorithm is said to run in **quadratic time** if $T(n) = \Theta(n^2)$.

Linear Growth

- ▶ If input size doubles, time roughly *doubles*.
- ▶ If code takes 5 seconds on 1,000 points...
- ▶ ...on 100,000 data points it takes ≈ 500 seconds.
- ▶ i.e., 8.3 minutes

Quadratic Growth

- ▶ If input size doubles, time roughly *quadruples*.
- ▶ If code takes 5 seconds on 1,000 points...
- ▶ ...on 100,000 points it takes $\approx 50,000$ seconds.
- ▶ i.e., ≈ 14 hours

In data science...

- ▶ Let's say we have a training set of 10,000 points.
- ▶ If model takes **quadratic** time to train, should expect to wait minutes to hours.
- ▶ If model takes **linear** time to train, should expect to wait seconds to minutes.
- ▶ These are rules of thumb only.

Exponential Growth

- ▶ Increasing input size by one *doubles* (triples, etc.) time taken.
- ▶ Grows very quickly!
- ▶ **Example:** brute force search of 2^n subsets.

```
for subset in all_subsets(things):  
    print(subset)
```

Logarithmic Growth

- ▶ To increase time taken by one unit, must *double* (triple, etc.) the input size.
- ▶ Grows very slowly!
- ▶ $\log n$ grows slower than n^α for *any* $\alpha > 0$
 - ▶ I.e., $\log n$ grows slower than n , \sqrt{n} , $n^{1/1,000}$, etc.

Exercise

What is the asymptotic time complexity of the code below as a function of n ?

```
i = 1
while i <= n
    i = i * 2
```


Solution

- Same general strategy as before: “how many times does loop body run?”

```
i = 1
while i <= n
    i = i * 2
```

<i>n</i>	# iters.
1	
2	
3	
4	
5	
6	
7	
8	

Common Growth Rates

- 
- ▶ $\Theta(1)$: **constant**
 - ▶ $\Theta(\log n)$: **logarithmic**
 - ▶ $\Theta(n)$: **linear**
 - ▶ $\Theta(n \log n)$: **linearithmic**
 - ▶ $\Theta(n^2)$: **quadratic**
 - ▶ $\Theta(n^3)$: **cubic**
 - ▶ $\Theta(2^n)$: **exponential**

Exercise

Which grows faster, $n!$ or 2^n ?

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 2

Big Theta, Formalized

So Far

- ▶ Time Complexity Analysis: a picture of how an algorithm **scales**.
- ▶ Can use Θ -notation to express time complexity.
- ▶ Allows us to **ignore** details in a rigorous way.
 - ▶ **Saves us work!**
 - ▶ **But what exactly can we ignore?**

Theta Notation, Informally

- ▶ $\Theta(\cdot)$ forgets constant factors, lower-order terms.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

Theta Notation, Informally

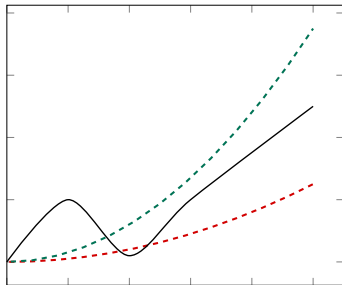
- ▶ $f(n) = \Theta(g(n))$ if $f(n)$ “grows like” $g(n)$.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

Definition

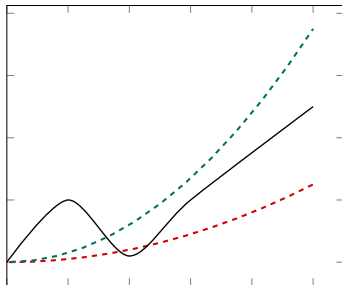
We write $f(n) = \Theta(g(n))$ if there are positive constants N , c_1 and c_2 such that for all $n \geq N$:

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



Main Idea

If $f(n) = \Theta(g(n))$, then when n is large f is “sandwiched” between copies of g .



Proving Big-Theta

- ▶ We can prove that $f(n) = \Theta(g(n))$ by finding these constants.

$$c_1g(n) \leq f(n) \leq c_2g(n) \quad (n \geq N)$$

- ▶ Requires an upper bound and a lower bound.

Strategy: Chains of Inequalities

- ▶ To show $f(n) \leq c_2 g(n)$, we show:

$$f(n) \leq (\text{something}) \leq (\text{another thing}) \leq \dots \leq c_2 g(n)$$

- ▶ At each step:
 - ▶ We can do anything to make value **larger**.
 - ▶ But the goal is to simplify it to look like $g(n)$.

Example

- ▶ Show that $4n^3 - 5n^2 + 50 = \Theta(n^3)$.
- ▶ Find constants c_1, c_2, N such that for all $n > N$:

$$c_1 n^3 \leq 4n^3 - 5n^2 + 50 \leq c_2 n^3$$

- ▶ They don't have to be the “best” constants! Many solutions!

Example

$$c_1 n^3 \leq 4n^3 - 5n^2 + 50 \leq c_2 n^3$$

- ▶ We want to make $4n^3 - 5n^2 + 50$ “look like” cn^3 .
- ▶ For the upper bound, can do anything that makes the function **larger**.
- ▶ For the lower bound, can do anything that makes the function **smaller**.

Example

$$c_1 n^3 \leq 4n^3 - 5n^2 + 50 \leq c_2 n^3$$

► Upper bound:

Upper-Bounding Tips

- ▶ “Promote” lower-order **positive** terms:

$$3n^3 + 5n \leq 3n^3 + 5n^3$$

- ▶ “Drop” **negative** terms

$$3n^3 - 5n \leq 3n^3$$

Example

$$c_1 n^3 \leq 4n^3 - 5n^2 + 50 \leq c_2 n^3$$

► Lower bound:

Lower-Bounding Tips

- ▶ “Drop” lower-order **positive** terms:

$$3n^3 + 5n \geq 3n^3$$

- ▶ “Promote and cancel” negative lower-order terms if possible:

$$4n^3 - 2n \geq 4n^3 - 2n^3 = 2n^3$$

Lower-Bounding Tips

- ▶ “Cancel” negative lower-order terms with big constants by “breaking off” a piece of high term.

$$\begin{aligned}4n^3 - 10n^2 &= (3n^3 + n^3) - 10n^2 \\ &= 3n^3 + (n^3 - 10n^2)\end{aligned}$$

$$n^3 - 10n^2 \geq 0 \text{ when } n^3 \geq 10n^2 \implies n \geq 10:$$

$$\geq 3n^3 + 0 \quad (n \geq 10)$$

Caution

- ▶ To upper bound a fraction A/B , you must:
 - ▶ Upper bound the numerator, A .
 - ▶ *Lower* bound the denominator, B .
- ▶ And to lower bound a fraction A/B , you must:
 - ▶ Lower bound the numerator, A .
 - ▶ *Upper* bound the denominator, B .

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 3

Big-Oh and Big-Omega

Other Bounds

- ▶ $f = \Theta(g)$ means that f is both **upper** and **lower** bounded by factors of g .
- ▶ Sometimes we only have (or care about) upper bound or lower bound.
- ▶ We have notation for that, too.

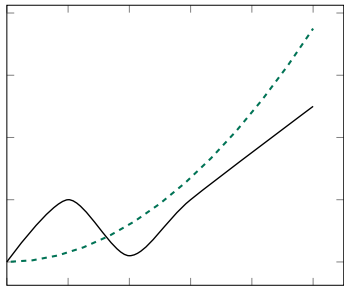
Big-O Notation, Informally

- ▶ Sometimes we only care about **upper bound**.
- ▶ $f(n) = O(g(n))$ if f “grows at most as fast” as g .
- ▶ Examples:
 - ▶ $4n^2 = O(n^{100})$
 - ▶ $4n^2 = O(n^3)$
 - ▶ $4n^2 = O(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = O(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

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



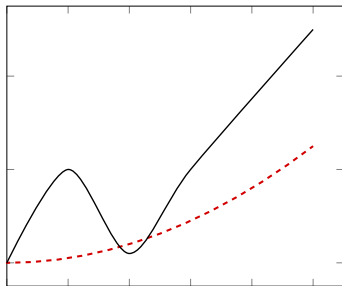
Big-Omega Notation

- ▶ Sometimes we only care about **lower bound**.
- ▶ Intuitively: $f(n) = \Omega(g(n))$ if f “grows at least as fast” as g .
- ▶ Examples:
 - ▶ $4n^{100} = \Omega(n^5)$
 - ▶ $4n^2 = \Omega(n)$
 - ▶ $4n^2 = \Omega(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = \Omega(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

$$c_1 \cdot g(n) \leq f(n)$$



FUN FACT

“Omega” in Greek literally means: big O.
So translated, “Big-Omega” means “big big O”.

Theta, Big-O, and Big-Omega

- ▶ If $f = \Theta(g)$ then $f = O(g)$ and $f = \Omega(g)$.
- ▶ If $f = O(g)$ and $f = \Omega(g)$ then $f = \Theta(g)$.
- ▶ Pictorially:
 - ▶ $\Theta \implies (O \text{ and } \Omega)$
 - ▶ $(O \text{ and } \Omega) \implies \Theta$

Analogyes

- ▶ Θ is kind of like $=$
- ▶ O is kind of like \leq
- ▶ Ω is kind of like \geq

Why?

- ▶ Laziness.
- ▶ Sometimes finding an upper or lower bound would take **too much work**, and/or we don't really care about it anyways.

Big-Oh

- ▶ Often used when another part of the code would dominate time complexity anyways.

Exercise

What is the time complexity of foo?

```
def foo(n):  
    for a in range(n**4):  
        print(a)  
  
    for i in range(n):  
        for j in range(i**2):  
            print(i + j)
```

Example: Big-Oh

```
def foo(n):  
    for a in range(n**4):  
        print(a)  
  
    for i in range(n):  
        for j in range(i**2):  
            print(i + j)
```

Big-Omega

- ▶ Often used when the time complexity will be so large that we don't care what it is, exactly.

Example: Big-Omega

```
best_separation = float('inf')
best_clustering = None

for clustering in all_clusterings(data):
    sep = calculate_separation(clustering)
    if sep < best_separation:
        best_separation = sep
        best_clustering = clustering

print(best_clustering)
```

Other Notations

- ▶ $f(n) = o(g(n))$ if f grows “much slower” than g .
 - ▶ Whatever c you choose, eventually $f < cg(n)$.
 - ▶ Example: $n^2 = o(n^3)$
- ▶ $f(n) = \omega(g(n))$ if f grows “much faster” than g .
 - ▶ Whatever c you choose, eventually $f > cg(n)$.
 - ▶ Example: $n^3 = \omega(n^2)$
- ▶ We won't really use these.

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 4

Properties

Properties

- ▶ We don't usually go back to the definition when using Θ .
- ▶ Instead, we use a few basic **properties**.

Properties of Θ

1. **Symmetry:** If $f = \Theta(g)$, then $g = \Theta(f)$.
2. **Transitivity:** If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.
3. **Reflexivity:** $f = \Theta(f)$

Exercise

Which of the following properties are true?

- ▶ T or F: If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- ▶ T or F: If $f = \Omega(h)$ and $g = \Omega(h)$, then $f = \Omega(g)$.
- ▶ T or F: If $f_1 = \Theta(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = \Theta(g_1 + g_2)$.
- ▶ T or F: If $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2)$, then $f_1 \times f_2 = \Theta(g_1 \times g_2)$.

Proving/Disproving Properties

- ▶ Start by trying to disprove.
- ▶ Easiest way: find a counterexample.
- ▶ Example: If $f = \Omega(h)$ and $g = \Omega(h)$, then $f = \Omega(g)$.
 - ▶ **False!** Let $f = n^3$, $g = n^5$, and $h = n^2$.

Proving the Property

- ▶ If you can't disprove, maybe it is true.
- ▶ Example:
 - ▶ Suppose $f_1 = O(g_1)$ and $f_2 = O(g_2)$.
 - ▶ Prove that $f_1 \times f_2 = O(g_1 \times g_2)$.

Step 1: State the assumption

- ▶ We know that $f_1 = O(g_1)$ and $f_2 = O(g_2)$.
- ▶ So there are constants c_1, c_2, N_1, N_2 so that for all $n \geq N$:

$$f_1(n) \leq c_1 g_1(n) \quad (n \geq N_1)$$

$$f_2(n) \leq c_2 g_2(n) \quad (n \geq N_2)$$

Step 2: Use the assumption

- ▶ Chain of inequalities, starting with $f_1 \times f_2$, ending with $\leq cg_1 \times g_2$.
- ▶ Using the following piece of information:

$$f_1(n) \leq c_1 g_1(n) \quad (n \geq N_1)$$

$$f_2(n) \leq c_2 g_2(n) \quad (n \geq N_2)$$

Analyzing Code

- ▶ The properties of Θ (and O and Ω) are useful when analyzing code.
- ▶ We can analyze pieces, put together the results.

Sums of Theta

- ▶ **Property:** If $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2)$, then $f_1 + f_2 = \Theta(g_1 + g_2)$
- ▶ Used when analyzing **sequential** code.

Example

```
def foo(n):  
    bar(n)  
    baz(n)
```

- ▶ Say bar takes $\Theta(n^3)$, baz takes $\Theta(n^4)$.
- ▶ foo takes $\Theta(n^4 + n^3) = \Theta(n^4)$.
- ▶ baz is the **bottleneck**.

Products of Theta

- ▶ **Property:** If $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2)$, then

$$f_1 \cdot f_2 = \Theta(g_1 \cdot g_2)$$

- ▶ Useful when analyzing nested **loops**.

Example

```
def foo(n):  
    for i in range(3*n + 4, 5n**2 - 2*n + 5):  
        for j in range(500*n, n**3):  
            print(i, j)
```

Careful!

- ▶ If inner loop index depends on outer loop, you have to be more careful.

```
def foo(n):  
    for i in range(n):  
        for j in range(i):  
            print(i, j)
```

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 5

Asymptotic Notation Practicalities

In this part...

- ▶ Other ways asymptotic notation is used.
- ▶ Asymptotic notation *faux pas*.
- ▶ Downsides of asymptotic notation.

Not Just for Time Complexity!

- ▶ We most often see asymptotic notation used to express time complexity.
- ▶ But it can be used to express any type of growth!

Example: Combinatorics

- ▶ Recall: $\binom{n}{k}$ is number of ways of choosing k things from a set of n .
- ▶ How fast does this grow with n ? For fixed k :

$$\binom{n}{k} = \Theta(n^k)$$

- ▶ Example: the number of ways of choosing 3 things out of n is $\Theta(n^3)$.

Example: Central Limit Theorem

- ▶ Recall: the CLT says that the sample mean has a normal distribution with standard deviation $\sigma_{\text{pop}}/\sqrt{n}$
- ▶ The **error** in the sample mean is: $O(1/\sqrt{n})$

Faux Pas

- ▶ Asymptotic notation can be used improperly.
 - ▶ Might be technically correct, but defeats the purpose.
- ▶ Don't do these in, e.g., interviews!

Faux Pas #1

- ▶ Don't include constants, lower-order terms in the notation.
- ▶ **Bad:** $3n^2 + 2n + 5 = \Theta(3n^2)$.
- ▶ **Good:** $3n^2 + 2n + 5 = \Theta(n^2)$.
- ▶ It isn't *wrong* to do so, just defeats the purpose.

Faux Pas #2

- ▶ Don't include base in logarithm.
- ▶ **Bad:** $\Theta(\log_2 n)$
- ▶ **Good:** $\Theta(\log n)$
- ▶ Why? $\log_2 n = c \cdot \log_3 n = c' \log_4 n = \dots$

Faux Pas #3

- ▶ Don't misinterpret meaning of $\Theta(\cdot)$.
- ▶ $f(n) = \Theta(n^3)$ does **not** mean that there are constants so that $f(n) = c_3n^3 + c_2n^2 + c_1n + c_0$.

Faux Pas #4

- ▶ Time complexity is not a **complete** measure of efficiency.
- ▶ $\Theta(n)$ is not always “better” than $\Theta(n^2)$.
- ▶ Why?

Faux Pas #4

- ▶ **Why?** Asymptotic notation “hides the constants”.
- ▶ $T_1(n) = 1,000,000n = \Theta(n)$
- ▶ $T_2(n) = 0.000001n^2 = \Theta(n^2)$
- ▶ But $T_1(n)$ is **worse** for all but really large n .

Main Idea

Time complexity is not the **only** way to measure efficiency, and it can be misleading.

Sometimes even a $\Theta(2^n)$ algorithm is better than a $\Theta(n)$ algorithm, if the data size is small.

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 6

The Movie Problem

The Movie Problem



The Movie Problem

- ▶ **Given:** an array `movies` of movie durations, and the flight duration `t`
- ▶ **Find:** two movies whose durations add to `t`.
 - ▶ If no two movies sum to `t`, return **None**.

Exercise

Design a brute force solution to the problem. What is its time complexity?

```
def find_movies(movies, t):  
    n = len(movies)  
    for i in range(n):  
        for j in range(i + 1, n):  
            if movies[i] + movies[j] == t:  
                return (i, j)  
    return None
```

Time Complexity

- ▶ It looks like there is a **best** case and **worst** case.
- ▶ How do we formalize this?

For the future...

- ▶ Can you come up with a better algorithm?
- ▶ What is the *best possible* time complexity?

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 7

Best and Worst Cases

Example 1: mean

```
def mean(arr):  
    total = 0  
    for x in arr:  
        total += x  
    return total / len(arr)
```

Time Complexity of mean

- ▶ Linear time, $\Theta(n)$.
- ▶ Depends **only** on the array's **size**, n , not on its actual elements.

Example 2: Linear Search

- ▶ **Given:** an array `arr` of numbers and a target `t`.
- ▶ **Find:** the index of `t` in `arr`, or **None** if it is missing.

```
def linear_search(arr, t):  
    for i, x in enumerate(arr):  
        if x == t:  
            return i  
    return None
```

Exercise

What is the time complexity of `linear_search`?

```
def linear_search(arr, t):  
    for i, x in enumerate(arr):  
        if x == t:  
            return i  
    return None
```

Observation

- ▶ It looks like there are *two* extreme cases...

The Best Case

- ▶ When the target, t , is the very first element.
- ▶ The loop exits after one iteration.
- ▶ $\Theta(1)$ time?

The **Worst** Case

- ▶ When the target, t , is not in the array at all.
- ▶ The loop exits after n iterations.
- ▶ $\Theta(n)$ time?

Time Complexity

- ▶ `linear_search` can take vastly different amounts of time on two inputs of the **same size**.
 - ▶ Depends on **actual elements** as well as size.
- ▶ It has no single, overall time complexity.
- ▶ Instead we'll report **best** and **worst** case time complexities.

Best Case Time Complexity

- ▶ How does the time taken in the **best case** grow as the input gets larger?

Definition

Define $T_{\text{best}}(n)$ to be the **least** time taken by the algorithm on any input of size n .

The asymptotic growth of $T_{\text{best}}(n)$ is the algorithm's **best case time complexity**.

Best Case

- ▶ In `linear_search`'s **best case**, $T_{\text{best}}(n) = c$, no matter how large the array is.
- ▶ The **best case time complexity** is $\Theta(1)$.

Worst Case Time Complexity

- ▶ How does the time taken in the **worst case** grow as the input gets larger?

Definition

Define $T_{\text{worst}}(n)$ to be the **most** time taken by the algorithm on any input of size n .

The asymptotic growth of $T_{\text{worst}}(n)$ is the algorithm's **worst case time complexity**.

Worst Case

- ▶ In the worst case, `linear_search` iterates through the entire array.
- ▶ The **worst case time complexity** is $\Theta(n)$.

Exercise

What are the best case and worst case time complexities of find_movies?

```
def find_movies(movies, t):  
    n = len(movies)  
    for i in range(n):  
        for j in range(i + 1, n):  
            if movies[i] + movies[j] == t:  
                return (i, j)  
    return None
```


Best Case

- ▶ Best case occurs when movie 1 and movie 2 add to the target.
- ▶ Takes constant time, independent of number of movies.
- ▶ Best case time complexity: $\Theta(1)$.

Worst Case

- ▶ Worst case occurs when no two movies add to target.
- ▶ Has to loop over all $\Theta(n^2)$ pairs.
- ▶ Worst case time complexity: $\Theta(n^2)$.

Caution!

- ▶ The best case is never: “the input is of size one”.
- ▶ The best case is about the **structure** of the input, not its **size**.
- ▶ Not always constant time! Example: sorting.

Note

- ▶ An algorithm like `linear_search` doesn't have **one single** time complexity.
- ▶ An algorithm like `mean` does, since the best and worst case time complexities coincide.

Main Idea

Reporting **best** and **worst** case time complexities gives us a richer of the performance of the algorithm.

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 8

Appendix: About Notation

A Common Mistake

- ▶ You'll sometimes see people equate $O(\cdot)$ with **worst case** and $\Omega(\cdot)$ with **best case**.
- ▶ This isn't right!

Why?

- ▶ $O(\cdot)$ expresses ignorance about a lower bound.
 - ▶ $O(\cdot)$ is like \leq
- ▶ $\Omega(\cdot)$ expresses ignorance about an upper bound.
 - ▶ $\Omega(\cdot)$ is like \geq
- ▶ Having both bounds is actually important here.

Example

- ▶ Suppose we said: “the worst case time complexity of `find_movies` is $O(n^2)$.”
- ▶ Technically true, but not precise.
- ▶ This is like saying: “I **don't know** how bad it actually is, but it can't be worse than quadratic.”
 - ▶ It could still be linear!”
- ▶ **Better:** the worst case time complexity is $\Theta(n^2)$.

Example

- ▶ Suppose we said: “the best case time complexity of `find_movies` is $\Omega(1)$.”
- ▶ This is like saying: “I **don't know** how good it actually is, but it can't be better than constant.”
 - ▶ It could be linear!
- ▶ **Correct:** the best case time complexity is $\Theta(1)$.

Put Another Way...

- ▶ It isn't **technically wrong** to say worst case for `find_movies` is $O(n^2)$...
- ▶ ...but it isn't **technically wrong** to say it is $O(n^{100})$, either!

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 9

Appendix: Asymptotic Notation and Limits

Limits and Θ , O , Ω

- ▶ You might prefer to use limits when reasoning about asymptotic notation.
- ▶ **Warning!** There are some tricky subtleties.
- ▶ Be able to “fall back” to the formal definitions.

Theta and Limits

- ▶ **Claim:** If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$, then $f(n) = \Theta(g(n))$.
- ▶ Example: $4n^3 - 5n^2 + 50$.

Warning!

- ▶ Converse **isn't true**: if $f(n) = \Theta(g(n))$, it need not be that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$.
- ▶ The limit can be **undefined**.
- ▶ Example: $5 + \sin(n) = \Theta(1)$, but the limit d.n.e.

Big-O and Limits

- ▶ If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) = O(g(n))$.
- ▶ Namely, the limit can be zero. e.g., $n = O(n^2)$.

Big-O and Limits

- ▶ If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) = O(g(n))$.
- ▶ Namely, the limit can be zero. e.g., $n = O(n^2)$.
- ▶ **Warning!** Converse not true. Limit may not exist.

Big-Omega and Limits

- ▶ If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$, then $f(n) = \Omega(g(n))$.
- ▶ Namely, the limit can be ∞ . e.g., $n^2 = \Omega(n)$.

Big-Omega and Limits

- ▶ If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$, then $f(n) = \Omega(g(n))$.
- ▶ Namely, the limit can be ∞ . e.g., $n^2 = \Omega(n)$.
- ▶ **Warning!** Converse not true. Limit may not exist.

Good to Know

- ▶ $\log_b n$ grows slower than n^p , as long as $p > 0$.
- ▶ Example:

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{n^{0.000001}} = 0$$