

# DSC 40B

## *Theoretical Foundations II*

Lecture 2 | Part 1

**News**

# News

- ▶ Lab 01 posted on Gradescope
  - ▶ Due Friday @ 11:59 pm PST on Gradescope.
- ▶ Homework 01 will be posted at [dsc40b.com](https://dsc40b.com)
  - ▶ Due next Tuesday @ 11:59 pm PST on Gradescope.
  - ▶ LaTeX template available.

# Agenda

1. Analyzing nested loops.
2. What is  $\Theta$  notation, really?

# DSC 40B

## *Theoretical Foundations II*

Lecture 2 | Part 2

### **Nested Loops**

# Example 1: Interview Problem



## Example 1: Interview Problem

- ▶ Design an algorithm to solve the following problem...
- ▶ Given the heights of  $n$  people, what is the height of the tallest doctor you can make by stacking two of them?

## Exercise

- ▶ What is the time complexity of the brute force solution?
- ▶ **Bonus:** what is the **best possible** time complexity of any solution?

# The Brute Force Solution

- ▶ Loop through all possible (ordered) pairs.
  - ▶ How many are there?
- ▶ Check height of each.
- ▶ Keep the best.



Time/exec.    # of execs.

```
def tallest_doctor(heights):  
    max_height = -float('inf')  
    n = len(heights)  
    for i in range(n):  
        for j in range(n):  
            if i == j:  
                continue  
            height = heights[i] + heights[j]  
            if height > max_height:  
                max_height = height  
    return max_height
```

# Time Complexity

- ▶ Time complexity of this is  $\Theta(n^2)$ .
- ▶ **TODO:** Can we do better?
- ▶ Note: this algorithm considers each pair of people **twice**.
- ▶ We'll fix that in a moment.

## First: A shortcut

- ▶ Making a table is getting tedious.
- ▶ Usually, find a chunk that **dominates** time complexity; i.e., yields the leading term of  $T(n)$ .
- ▶ **Observation:** If each line takes constant time to execute once, the line that runs the most **dominates** the time complexity.

# Totalling Up

```
for i in range(n):  
    for j in range(n):  
        height = heights[i] + heights[j] # <- count execs.
```

- ▶ On outer iter. # 1, inner body runs \_\_\_\_\_ times.
- ▶ On outer iter. # 2, inner body runs \_\_\_\_\_ times.
- ▶ On outer iter. #  $\alpha$ , inner body runs \_\_\_\_\_ times.
- ▶ The outer loop runs \_\_\_\_\_ times.
- ▶ Total number of executions: \_\_\_\_\_

```
def f(n):  
    for i in range(3*n**3 + 5*n**2 - 100):  
        for j in range(n**5, n**6):  
            print(i, j)
```

## Example 2: The Median

- ▶ **Given:** real numbers  $x_1, \dots, x_n$ .
- ▶ **Compute:**  $h$  minimizing the **total absolute loss**

$$R(h) = \sum_{i=1} |x_i - h|$$

## Example 2: The Median

- ▶ **Solution:** the **median**.
- ▶ That is, a **middle** number.
- ▶ But how do we actually **compute** a median?

# A Strategy

- ▶ **Recall:** one of  $x_1, \dots, x_n$  must be a median.
- ▶ **Idea:** compute  $R(x_1), R(x_2), \dots, R(x_n)$ , return  $x_i$  that gives the smallest result.

$$R(h) = \sum_{i=1}^n |x_i - h|$$

- ▶ Basically a **brute force** approach.



## Exercise

- ▶ What is the time complexity of this brute force approach?
- ▶ How long will it take to run on an input of size 10,000?

```
def median(numbers):  
    min_h = None  
    min_value = float('inf')  
    for h in numbers:  
        total_abs_loss = 0  
        for x in numbers:  
            total_abs_loss += abs(x - h)  
        if total_abs_loss < min_value:  
            min_value = total_abs_loss  
            min_h = h  
    return min_h
```

# The Median

- ▶ The brute force approach has  $\Theta(n^2)$  time complexity.
- ▶ **TODO:** Is there a better algorithm?

# The Median

- ▶ The brute force approach has  $\Theta(n^2)$  time complexity.
- ▶ **TODO:** Is there a better algorithm?
  - ▶ It turns out, you can find the median in *linear* time.<sup>1</sup>

---

<sup>1</sup>Well, *expected* time.

```
In [8]: numbers = list(range(10_000))
```

```
In [9]: %time median(numbers)
```

```
CPU times: user 7.26 s, sys: 0 ns, total: 7.26 s
```

```
Wall time: 7.26 s
```

```
Out[9]: 4999
```

```
In [10]: %time mystery_median(numbers)
```

```
CPU times: user 4.3 ms, sys: 2 µs, total: 4.3 ms
```

```
Wall time: 4.3 ms
```

```
Out[10]: 4999
```

# Careful!

- ▶ Not every nested loop has  $\Theta(n^2)$  time complexity!

```
def foo(n):  
    for x in range(n):  
        for y in range(10):  
            print(x + y)
```

# DSC 40B

## *Theoretical Foundations II*

Lecture 2 | Part 3

### **Dependent Nested Loops**

## Example 3: Tallest Doctor, Again

- ▶ Our previous algorithm for the tallest doctor computed height for each *ordered* pair of people.
  - ▶  $i = 3$  and  $j = 7$  is the same as  $i = 7$  and  $j = 3$
- ▶ **Idea:** consider each *unordered* pair only once:

```
for i in range(n):  
    for j in range(i + 1, n):
```

- ▶ What is the time complexity?



# Pictorially

```
for i in range(4):  
    for j in range(4):  
        print(i, j)
```

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

# Pictorially

```
for i in range(4):  
    for j in range(i + 1, 4):  
        print(i, j)
```

(0,1) (0,2) (0,3)  
 (1,2) (1,3)  
 (2,3)

```
1 def tallest_doctor_2(heights):
2     max_height = -float('inf')
3     n = len(heights)
4     for i in range(n):
5         for j in range(i + 1, n):
6             height = heights[i] + heights[j]
7             if height > max_height:
8                 max_height = height
```

- ▶ **Goal:** How many times does line 6 run in total?
- ▶ Now inner nested loop **depends** on outer nested loop.

# Independent

```
for i in range(n):  
    for j in range(n):  
        ...
```

- ▶ Inner loop doesn't depend on outer loop iteration #.
- ▶ Just multiply: inner body executed  $n \times n = n^2$  times.

# Dependent

```
for i in range(n):  
    for j in range(i, n):  
        ...
```

- ▶ Inner loop depends on outer loop iteration #.
- ▶ Can't just multiply: inner body executed ??? times.

# Dependent Nested Loops

```
for i in range(n):  
    for j in range(i + 1, n):  
        height = heights[i] + heights[j]
```

- ▶ Idea: find formula  $f(\alpha)$  for “number of iterations of inner loop during outer iteration  $\alpha$ ”<sup>2</sup>

- ▶ Then total:  $\sum_{\alpha=1}^n f(\alpha)$

---

<sup>2</sup>Why  $\alpha$  and not  $i$ ? Python starts counting at 0, math starts at 1. Using  $i$  would be confusing – does it start at 0 or 1?

```
for i in range(n):  
    for j in range(i + 1, n):  
        height = heights[i] + heights[j]
```

- ▶ On outer iter. # 1, inner body runs \_\_\_\_\_ times.
- ▶ On outer iter. # 2, inner body runs \_\_\_\_\_ times.
- ▶ On outer iter. #  $\alpha$ , inner body runs \_\_\_\_\_ times.
- ▶ The outer loop runs \_\_\_\_\_ times.

# Totalling Up

- ▶ On outer iteration  $\alpha$ , inner body runs  $n - \alpha$  times.
  - ▶ That is,  $f(\alpha) = n - \alpha$
- ▶ There are  $n$  outer iterations.
- ▶ So we need to calculate:

$$\sum_{\alpha=1}^n f(\alpha) = \sum_{\alpha=1}^n (n - \alpha)$$

$$\sum_{\alpha=1}^n (n - \alpha)$$

=

$$\underbrace{(n-1)}_{\text{1st outer iter}} + \underbrace{(n-2)}_{\text{2nd outer iter}} + \dots + \underbrace{(n-\alpha)}_{\text{kth outer iter}} + \dots + \underbrace{(n-(n-1))}_{\text{(n-1)th outer iter}} + \underbrace{(n-n)}_{\text{nth outer iter}}$$

=

$$1 + 2 + 3 + \dots + (n-3) + (n-2) + (n-1)$$

=



## Aside: Arithmetic Sums

- ▶  $1 + 2 + 3 + \dots + (n-1) + n$  is an **arithmetic sum**.
- ▶ Formula for total:  $n(n + 1)/2$ .
- ▶ You should memorize it!

# Time Complexity

- ▶ `tallest_doctor_2` has  $\Theta(n^2)$  time complexity
- ▶ Same as original `tallest_doctor`!
- ▶ Should we have been able to guess this? Why?

## Reason 1: Number of Pairs

- ▶ We're doing constant work for each unordered pair.
- ▶ Recall from 40A: number of pairs of  $n$  objects is

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

- ▶ So  $\Theta(n^2)$



## Main Idea

If the loops are dependent, you'll usually need to write down a summation, evaluate.

## Main Idea

Halving the work (or thirding, quartering, etc.) doesn't change the time complexity.

## Exercise

Design a linear time algorithm for this problem.

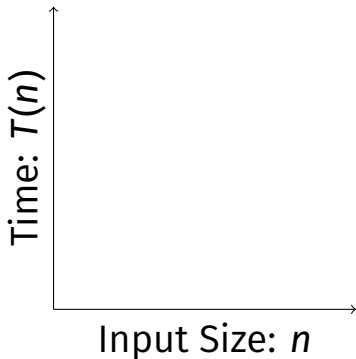
# DSC 40B

## *Theoretical Foundations II*

Lecture 2 | Part 4

**Growth Rates**

# 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  $\approx 500$  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.,  $\approx 14$  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  $n^\alpha$  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 2 | Part 5

**Big Theta, Formalized**

# So Far

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

# Now

- ▶ A deeper look at **asymptotic notation**:
- ▶ What does  $\Theta(\cdot)$  mean, exactly?
- ▶ Related notations:  $O(\cdot)$  and  $\Omega(\cdot)$ .
- ▶ How these notations save us work.

# 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”  $g(n)$ .

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

# Theta Notation Examples

►  $4n^2 + 3n - 20 = \Theta(n^2)$

►  $3n + \sin(4\pi n) = \Theta(n)$

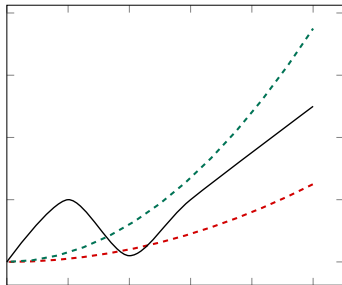
►  $2^n + 100n = \Theta(2^n)$



## Definition

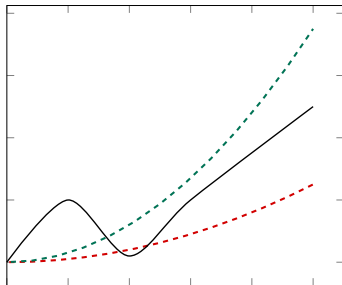
We write  $f(n) = \Theta(g(n))$  if there are positive constants  $N$ ,  $c_1$  and  $c_2$  such that for all  $n \geq N$ :

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



## 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.

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad (n \geq N)$$

- ▶ 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:

# Example

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

► Lower bound:



# Example

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

► All together:

# Upper-Bounding Tips

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

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

- ▶ “Drop” **negative** terms

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

# 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,  $B$ .

## Exercise

Let  $f(n) = [3n + (n \sin(\pi n) + 3)]n$ . Which one of the following is true?

- ▶  $f = \Theta(n)$
- ▶  $f = \Theta(n^2)$
- ▶  $f = \Theta(n \sin(\pi n))$