DSC 40B

Lecture 2 : Timing, Counting Operations, Nested Loop

Mic!

Announcements

Announcements

- Lab01 is posted on Gradescope.
  - Due Monday, 11:59pm on Gradescope
- Homework 1 is posted on [DSC40b.com](DSC40b.com)
  - Due Wed, 11:59pm on Gradescope
  - Hand-written, submit pdf
- First discussion is today at 4pm, same room

https://webclicker.web.app/

ZNSOLY

Steps:

1. Go to a link above
2. Code: ZNSOLY
3. Make sure to use your UCSD email address (i.e., @ucsd.edu)
4. Use quest/public wifi please.
5. Answer the questions when I active the poll.
6. Do not worry if it does not work today. The first class does not count. We will figure it out eventually.

Measuring Efficiency by Timing

Efficiency

- Speed matters, especially with large data sets.
- An algorithm is only useful if it runs **fast enough.**
  - That depends on the size of your data set.
- How do we measure the efficiency of code?
- How do we know if a method will be fast enough?

Scenario

- You’re building a least squares regression model to predict a patient’s blood oxygen level.
- You’ve trained it on 1,000 people.
- You have a full data set of 100,000 people.
- How long will it take? How does it **scale**?

Example: Scaling

- Your code takes 5 seconds on 1,000 points.
- How long will it take on 100,000 data points?
- 5 seconds × 100 = 500 seconds?
- More? Less?

Coming Up

- We'll answer this in coming lectures.
- Today: start with simpler algorithms for the mean, median.

Approach #1: Timing

- How do we measure the efficiency of code?
- Simple: time it!
- Useful Jupyter tools: `time` and `timeit`
    - Magic functions

```
[4]: numbers = range(1000)

[5]: %%time
     sum (numbers)

     CPU times: user 30 µs, sys: 0 ns, total: 30 µs
     Wall time: 34.3 µs

[5]: 499500

[6]: %%timeit
     sum(numbers)

     9.96 µs ± 3.79 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Disadvantages of Timing

1. Time depends on the computer.

Disadvantages of Timing

1. Time depends on the computer.

2. Depends on the particular input, too.

Disadvantages of Timing

1. Time depends on the computer.
2. Depends on the particular input, too.
3. One timing doesn't tell us how algorithm **scales**.

Measuring Efficiency by Counting Operations

Approach #2: Time Complexity Analysis

- Determine efficiency of code **without** running it.
- **Idea**: find a formula for time taken as a function of input size.

Advantages of Time Complexity

1. Doesn't depend on the computer.

2. Reveals which inputs are "hard", which are "easy".

3. Tells us how algorithm scales.

Exercise

Write a function **mean** which takes in a NumPy array of floats and outputs their mean (without a built-in function).

Solution

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

Time Complexity Analysis

- How long does it take mean to run on an array of size n?
  - Call this $\boldsymbol{T(n)}$.
- We want a **formula** for $T(n)$.

Counting Basic Operations

- Assume certain basic operations (like adding two numbers) take a constant amount of time.
    - `x + y` doesn't take more time if numbers is bigger.
    - So `x + y` takes "constant time"
    - Compare to `sum(numbers)`. Not a basic operation.
- **Idea**: Count the number of **basic** operations. This is a measure of time.

Exercise

- What is the complexity for each operation?
    - accessing an element: `arr[i]`
    - asking for the length: `len(arr)`
    - finding the max: `max(arr)`

A: O(1)

B: O(log n)

C: O(n)

D: Something else

Exercise

- What is the complexity for each operation?
  - accessing an element: `arr[i]` `-> Constant`
  - asking for the length: `len(arr)` `-> Constant`
  - finding the max: `max(arr)`   `-> Linear`

Basic Operations with Arrays

- We'll assume that these operations on NumPy arrays take constant time.
    - accessing an element: `arr[i]`
    - asking for the length: `len(arr)`

Example

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

Time/exec.	# of execs.

Example

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

Time/exec.	# of execs.
c_1	

Example

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

Time/exec.	# of execs.
c_1	1
?	

Example

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

Time/exec.	# of execs.
c_1	1
c_2	?

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
?	

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
?	

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
c_5	?

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
?	
c_5	1

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
c_4	?
c_5	1

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
?	
c_4	n
c_5	1

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
c_3	?
c_4	n
c_5	1

Example

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

Time/exec.	# of execs.
c_1	1
c_2	1
c_3	n+1
c_4	n
c_5	1

Example

$$T(n) = C_1 + C_2 + C_5 + n\,C_4 + (n+1)\,C_3$$

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

Time/exec.	# of execs.
C_1	1
C_2	1
C_3	n+1
C_4	n
C_5	1

Example: mean

- **Total time**:

  $T(n) = c_3(n + 1) + c_4 n + (c_1 + c_2 + c_5)$

  $= (c_3 + c_4)n + (c_1 + c_2 + c_3 + c_5)$

- "Forgetting" constants, lower-order terms with "Big-Theta": $T(n) = \Theta(n)$.
- $\Theta(n)$ is the **time complexity** of the algorithm.

Main Idea

Forgetting constant, lower order terms allows us to focus on how the algorithm **scales**, *independent* of which computer we run it on.

Careful!

Not always the case that a single line of code takes constant time per execution!

Example

```
def mean_2(numbers):
    total = sum(numbers)
    n = len(numbers)
    return total / n
```

Time/exec.	# of execs.
?	?

Example

```python
def mean_2(numbers):
    total = sum(numbers)
    n = len(numbers)
    return total / n
```

Time/exec.	# of execs.
?	?
c_3	1

Example

```
def mean_2(numbers):
    total = sum(numbers)
    n = len(numbers)
    return total / n
```

Time/exec.	# of execs.
?	?
c_2	1
c_3	1

Example

```python
def mean_2(numbers):
    total = sum(numbers)
    n = len(numbers)
    return total / n
```

Time/exec.	# of execs.
c_1 n	1
c_2	1
c_3	1

Example: mean_2

- **Total time:**

$$T(n) = c_1 n + (c_2 + c_3)$$

- “Forgetting” constants, lower-order terms with “Big-Theta”:

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

Exercise

- Write an algorithm for finding the **maximum** of an array of n numbers.
- What is its time complexity?

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1
c_2	n +1
?	?

```
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1
c_2	n +1
?	?
c_3	1

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1
c_2	n +1
c_4	n
?	?
c_3	1

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1
c_2	n +1
c_4	n
c_5	?
c_3	1

slidesmania.com

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1
c_2	n +1
c_4	n
c_5	<= n
c_3	1

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

Time/exec.	# of execs.
c_1	1
c_2	n +1
c_4	n
c_5	<= n
c_3	1

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

Main Idea

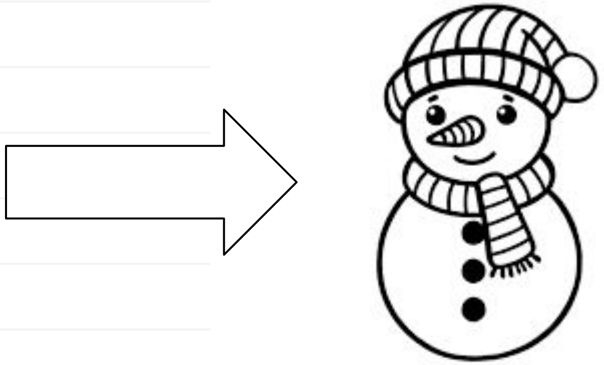
Using Big-Theta allows us not to worry about **exactly** how many times each line runs.

Remaining Questions

- What if the code is more complex?
  - For example, nested loops.
- What is this notation anyways?

Analyzing nested loops

Nested Loops. Example 1: Interview Problem



Given the diameters of *n* snowballs, what is the **tallest** snowman you can make using **exactly two** snowballs?

Exercise

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

A: Constant

B: logarithmic

C: Linear

D: Quadratic

E: Something else

The Brute Force Solution

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

How many ordered pairs?

How many ordered pairs?

- $N * (N - 1) = N^2 - N \sim N^2$

```python
def tallest_snowman(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
```

```
def tallest_snowman(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/exec.	# of execs.
c	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	c	
`    n = len(heights)`	c	
`    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`		

```python
def tallest_snowman(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
```

Line	Time/exec.	# of execs.
`max_height = -float('inf')`	c	
`n = len(heights)`	c	
`for i in range(n):`	c	
`for j in range(n):`		
`if i == j:`		
`continue`		
`height = heights[i] + heights[j]`		
`if height > max_height:`		
`max_height = height`		
`return max_height`		

Code	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	c	
`    n = len(heights)`	c	
`    for i in range(n):`	c	
`        for j in range(n):`	c	
`            if i == j:`	c	
`                continue`	c	
`            height = heights[i] + heights[j]`	c	
`            if height > max_height:`	c	
`                max_height = height`	c	
`    return max_height`	c	

Code	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`max_height = -float('inf')`	c	1
`n = len(heights)`	c	1
`for i in range(n):`	c	
`for j in range(n):`	c	
`if i == j:`	c	
`continue`	c	
`height = heights[i] + heights[j]`	c	
`if height > max_height:`	c	
`max_height = height`	c	
`return max_height`	c	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	C	1
`    n = len(heights)`	C	1
`    for i in range(n):`	C	n + 1
`        for j in range(n):`	C	
`            if i == j:`	C	
`                continue`	C	
`            height = heights[i] + heights[j]`	C	
`            if height > max_height:`	C	
`                max_height = height`	C	
`    return max_height`	C	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`max_height = -float('inf')`	C	1
`n = len(heights)`	C	1
`for i in range(n):`	C	n + 1
`for j in range(n):`	C	
`if i == j:`	C	n * n
`continue`	C	
`height = heights[i] + heights[j]`	C	
`if height > max_height:`	C	
`max_height = height`	C	
`return max_height`	C	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	c	1
`    n = len(heights)`	c	1
`    for i in range(n):`	c	n + 1
`        for j in range(n):`	c	
`            if i == j:`	c	n * n
`                continue`	c	n
`            height = heights[i] + heights[j]`	c	
`            if height > max_height:`	c	
`                max_height = height`	c	
`    return max_height`	c	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`max_height = -float('inf')`	c	1
`n = len(heights)`	c	1
`for i in range(n):`	c	n + 1
`for j in range(n):`	c	
`if i == j:`	c	n * n
`continue`	c	n
`height = heights[i] + heights[j]`	c	n*n - n
`if height > max_height:`	c	
`max_height = height`	c	
`return max_height`	c	

Code	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`max_height = -float('inf')`	C	1
`n = len(heights)`	C	1
`for i in range(n):`	C	n + 1
`for j in range(n):`	C	
`if i == j:`	C	n * n
`continue`	C	n
`height = heights[i] + heights[j]`	C	n*n - n
`if height > max_height:`	C	n*n - n
`max_height = height`	C	
`return max_height`	C	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`max_height = -float('inf')`	C	1
`n = len(heights)`	C	1
`for i in range(n):`	C	n + 1
`for j in range(n):`	C	
`if i == j:`	C	n * n
`continue`	C	n
`height = heights[i] + heights[j]`	C	n*n - n
`if height > max_height:`	C	n*n - n
`max_height = height`	C	<= n*n - n
`return max_height`	C	

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	C	1
`    n = len(heights)`	C	1
`    for i in range(n):`	C	n + 1
`        for j in range(n):`	C	
`            if i == j:`	C	n * n
`                continue`	C	n
`            height = heights[i] + heights[j]`	C	n*n - n
`            if height > max_height:`	C	n*n - n
`                max_height = height`	C	<= n*n - n
`    return max_height`	C	1

Code	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	C	1
`    n = len(heights)`	C	1
`    for i in range(n):`	C	n + 1
`        for j in range(n):`	C	?
`            if i == j:`	C	n * n
`                continue`	C	n
`            height = heights[i] + heights[j]`	C	n*n - n
`            if height > max_height:`	C	n*n - n
`                max_height = height`	C	<= n*n - n
`    return max_height`	C	1

A: n

B: n + 1

C: n * n

D: Som.else

	Time/exec.	# of execs.
`def tallest_snowman(heights):`		
`    max_height = -float('inf')`	C	1
`    n = len(heights)`	C	1
`    for i in range(n):`	C	n + 1
`        for j in range(n):`	C	n(n+1)
`            if i == j:`	C	n * n
`                continue`	C	n
`            height = heights[i] + heights[j]`	C	n*n - n
`            if height > max_height:`	C	n*n - n
`                max_height = height`	C	<= n*n - n
`    return max_height`	C	1

Time Complexity

- Time complexity of this is $\Theta(n^2)$.
- TODO: Can we do better?
- **Note**: this algorithm considers each pair of snowballs **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 $T(n)$.

A Shortcut

- Assume each line takes constant time to execute *once*.
- To determine the overall time complexity:
    1. Find the line that is execute most.
    2. Count how many times it is executed.

Shortcut for the Brute Force Solution

```
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.

Shortcut for the Brute Force Solution

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

- On outer iter. # 1, inner body runs ____n____ times.

Shortcut for the Brute Force Solution

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

- On outer iter. # 1, inner body runs ______n__________ times.
- On outer iter. # 2, inner body runs ____________________times.
- On outer iter. # α, inner body runs ____________________times.
- The outer loop runs _____________ times.
- Total number of executions: ____________________

Shortcut for the Brute Force Solution

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

- On outer iter. # 1, inner body runs ______n______ times.
- On outer iter. # 2, inner body runs ______n______times.
- On outer iter. # α, inner body runs ____________________times.
- The outer loop runs ____________ times.
- Total number of executions: ____________________

Shortcut for the Brute Force Solution

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

- On outer iter. # 1, inner body runs ______n______ times.
- On outer iter. # 2, inner body runs ______n______times.
- On outer iter. # α, inner body runs ______n______times.
- The outer loop runs __n______ times.
- Total number of executions: ____________________

Shortcut for the Brute Force Solution

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

- On outer iter. # 1, inner body runs ____n____ times.
- On outer iter. # 2, inner body runs ____n____times.
- On outer iter. # α, inner body runs ____n____times.
- The outer loop runs ___n______ times.
- Total number of executions: $\underbrace{n + n + \ldots + n}_{n} = n^2$

Example 2: The Median

- **Given**: real numbers x_1 , … , 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 , … , x_n must be a median.
- **Idea**: compute $R(x_1)$, $R(x_2)$, … , $R(x_n)$, return x_i that gives the smallest result.

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

- 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?

```python
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
```

```python
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
```

What is the complexity for each line of code?

```python
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
```

What is the complexity for each line of code?

What line executes the most?

```python
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
```

n^2 (annotating the line `total_abs_loss += abs(x - h)`)

What is the complexity for each line of code?

What line executes the most?

$T(n) = \Theta(n^2)$

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. (*expected*)

The Median

```
numbers = list(range(10000))
```

```
%time median(numbers)
```

```
CPU times: user 4.55 s, sys: 0 ns, total: 4.55 s
Wall time: 4.55 s

4999
```

The Median

```
numbers = list(range(10000))
```

```
%time median(numbers)
```

```
CPU times: user 4.55 s, sys: 0 ns, total: 4.55 s
Wall time: 4.55 s

4999
```

```
%time magic_median(numbers)
```

```
CPU times: user 5.42 ms, sys: 22 µs, total: 5.44 ms
Wall time: 5.04 ms
```

Careful!

- **Not every nested loop has $\Theta(n^2)$ time complexity!**
- In general, if:
  - outer loop iterates a times;
  - inner loop iterates b times for each outer loop iteration
    - We are assuming here that the number of inner loop iterations doesn't depend on which outer loop iteration we're in! That is called a **dependent** nested loop.
  - then the innermost loop body is executed $a \times b$ times.

```
for x in range(n):
    for y in range(n**2):
        print(x + y)
```

Example 3

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

Time complexity?

A: Constant

B: n

C: n log n

D: n^2

Example 4

```
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 4

```
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)
```

Ans:

$\Theta(n^9)$

Thank you!

Do you have any questions?

CampusWire!