CS-GY 6033
Design and Analysis of Algorithms I

Lecture 1 | Part 1

Administrative Stuff

Syllabus

- All course materials, the syllabus, etc., can be found at **course website**[1].
- Lectures will be recorded and posted on Brightspace
- We will use Ed discussion and Gradescope

[1] https://akbarrafiey.github.io/sp26-alg6033/

Tentative Topics

- Intro course to algorithm design and some data structures:
  - Recap of time complexity and asymptotic notations.
  - Sort and search algorithms.
  - Data structures: arrays, stacks, BSTs, heaps, hash maps, etc.
  - Graph algorithms
  - Greedy algorithms
  - Backtracking and dynamic programming
  - Sketching and streaming
  - Complexity theory

Grading

- 25% Written Homework:
  - There will be around 8-9 homeworks
  - 3 Slip days for the semester
  - AI and collaboration policy
- 15% in class quizzes (15-20 minutes):
  - Problems from the homeworks will be selected for quizzes.
- 25% Midterm 1: in class on March 13
- 25% Midterm 2: in class on May 01
- 10% class participation

Redemption Exams

- The final exam is a “no fault” final split into two sections:
  - Optional Midterm 01 “Redemption” section focusing on Lec 01—06
  - Optional Midterm 02 “Redemption” section focusing on Lec 07—12

CS-GY 6033
Design and Analysis of Algorithms I

Lecture 1 | Part 2

What is CS-GY 6033?

Example 1: Minimize Absolute Error

- **Goal**: summarize a collection of numbers, $x_1, \ldots, x_n$:
- **Idea**: find number M minimizing the total absolute error:

$$\sum_{i=1}^{n} |M - x_i|$$

Example 1: Minimize Absolute Error

- **Solution**: The **median** of $x_1, \ldots, x_n$.

The End?

Example 1: Minimize Absolute Error

- How do we actually **compute** the median?

Exercise

Suppose you're on a desert island with no internet connection, but a basic installation of Python. For some reason, you need to compute the median of a million numbers to get off of the island.

How do you do it?

Main Idea

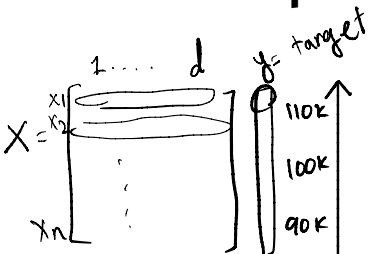
Our work doesn't stop once we solve the math problem.

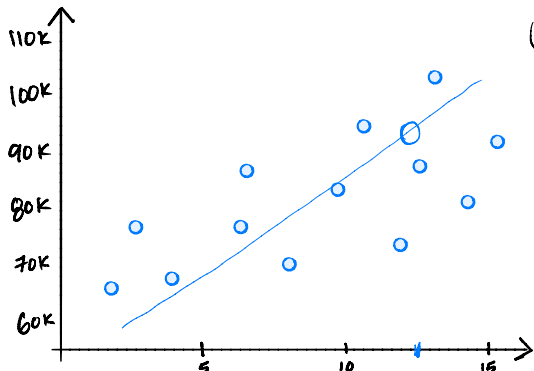
We still need to **compute** the answer.

We need an **algorithm**.

More than that, we need an **implementation** of that algorithm (that is: code).

Example 2: Least Squares Regression





$(\vec{x}_1, b_1)$

$(\vec{x}_2, b_2)$

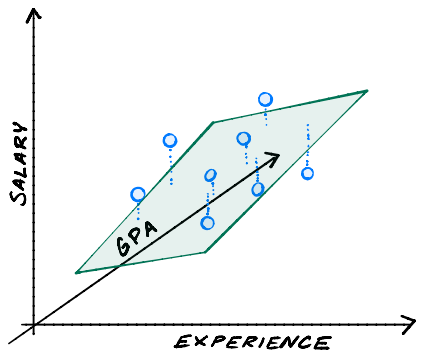
$\vdots$

$(\vec{x}_n, b_n)$

$$X\vec{w} \sim \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

$$\text{obj}: \min_{\vec{w}} \|X\vec{w} - \vec{b}\|_2^2$$

Example 2: Least Squares Regression



Closed form optimal solution

$$(X^T X)\vec{w} = X^T \vec{b}$$

optimal $\vec{w} = (X^T X)^{-1} X^T \vec{b}$

Wait...

- We actually need to **compute** the answer...
- We need an **algorithm**.

An Algorithm?

- Let's say we have `numpy` installed.
- It provides an implementation of an algorithm:

```
>>> import numpy as np
>>> w = np.linalg.solve(X.T @ X, X.T @ b)
```

But...

- Will it work for 1,000,000 data points?
- What about for 1,000,000 features?

Main Idea

Having an algorithm isn't enough – we need to know about its performance. Otherwise, it may be useless for our particular problem.

Example3: Clustering

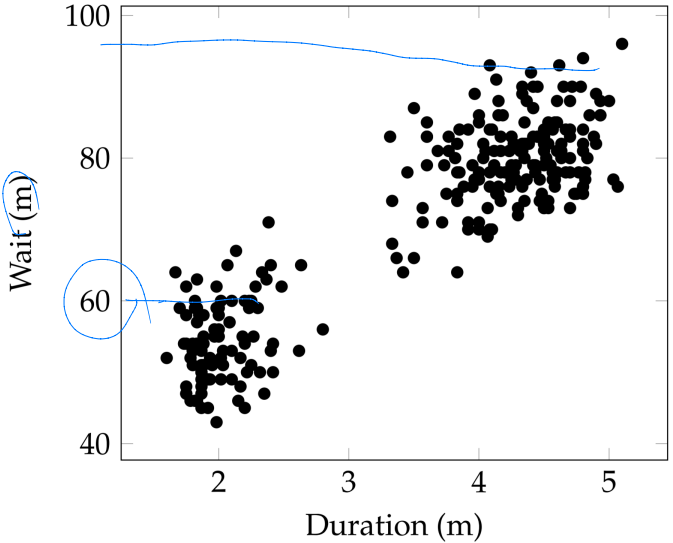
- Given a pile of data, discover similar groups.

- Examples:
  - Find political groups within social network data.
  - Given data on COVID-19 symptoms, discover groups that are affected differently.
  - Find the similar regions of an image (**segmentation**).

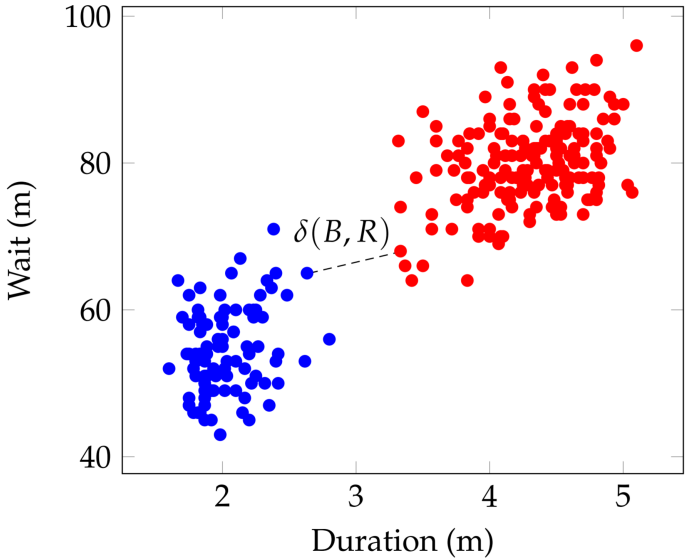
- Most useful when data is high dimensional...

Example: Old Faithful

Example: Old Faithful



Example: Old Faithful

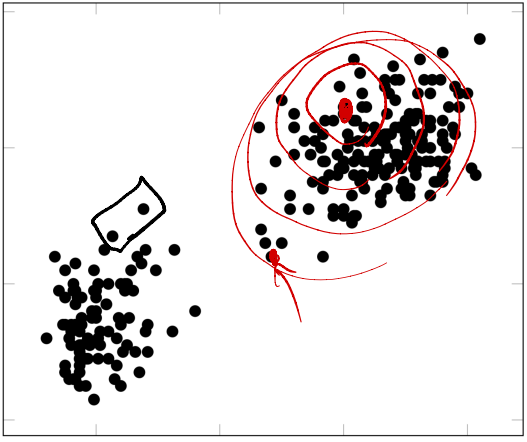


Clustering

- Goal: for computer to identify the two groups in the data.
- A clustering is an assignment of a color to each data point.
- There are many possible clusterings.

Clustering

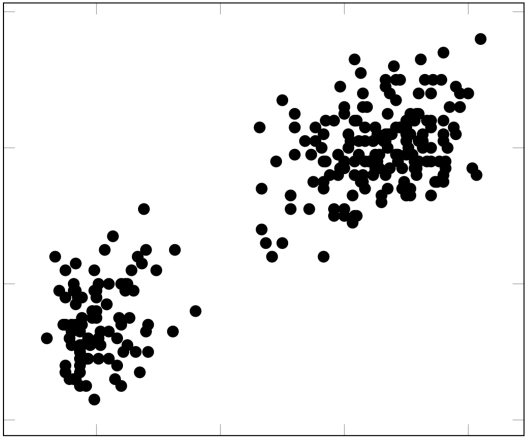
- How do we turn this into something a **computer** can do?
- Machine Learning: “Turn it into an optimization problem”.
- Idea: **design** a way of quantifying the “goodness” of a clustering; find the **best**.
  - Design a **loss function**.
  - There are many possibilities, tradeoffs!



Exercise

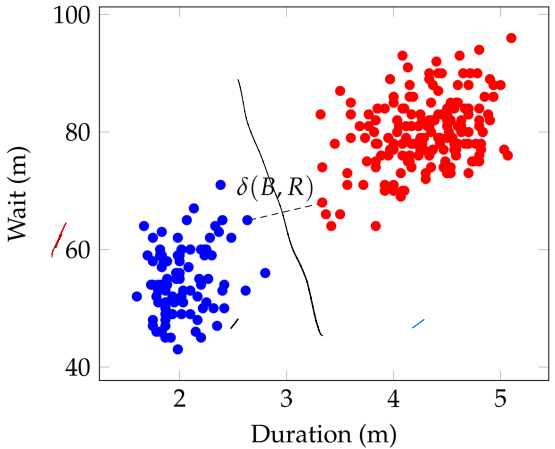
What's a good loss function for this problem? It should assign small loss to a **good** clustering.

Quantifying Separation



Idea: Define the "separation" $\delta(B, R)$ to be the *smallest* distance between a blue point and red point.

Quantifying Separation



Idea: Define the "separation" $\delta(B, R)$ to be the *smallest* distance between a blue point and red point.

The Problem

- **Given**: n points $\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}$.
- **Find**: an assignment of points to clusters R and B so as to maximize $\delta(B, R)$.

“The End?”

CS-GY 6033: “The Beginning”

The “Brute Force” Algorithm

- There are finitely-many possible clusterings.
- **Algorithm**: Try each possible clustering, return that with largest separation, $\delta(B, R)$.
- This is called a **brute force** algorithm.

```
best_separation = -float('inf') # Python for "infinity"
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)
```

The Algorithm

- We have an **algorithm**!
- But how long will this take to run if there are n points?
- How many clusterings of n things are there?

Exercise

How many ways are there of assigning R or B to n points?

Solution

- Two choices [2] for each object: $2 \times 2 \times \ldots \times 2 = 2^n$.

[2] Small nitpick: actual color doesn't matter, 2^{n-1}.

Time

- Suppose it takes at least *1 nanosecond*[3] to check a single clustering.
  - One *billionth* of a second.
  - Time it takes for light to travel 1 foot.

- If there are n points, it will take *at least* 2^n nanoseconds to check all clusterings.

[3] This is an *extremely* optimistic estimate. It's actually much slower, and scales with n.

Time Needed

n	Time
1	1 nanosecond

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days
60	36 years

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days
60	36 years
70	37,000 years

Example: Old Faithful

- The Old Faithful data set has 270 points.
  - Brute force algorithm will finish in 6×10^{64} years.

Example: Old Faithful

- The Old Faithful data set has 270 points.
  - Brute force algorithm will finish in 6×10^{64} years.

Algorithm Design

- Often, most obvious algorithm is **unusably slow**.

Algorithm Design

- Often, most obvious algorithm is **unusably slow**.
- Does this mean our problem is too hard?
  - Direct result of our choice of loss function.

Main Idea

Just having an algorithm isn't enough – it must also be reasonably **efficient**. Otherwise, it might be useless for our particular problem.

CS-GY 6033

- Assess the efficiency of algorithms.
- Understand why and how common algorithms work.
- Develop faster algorithms using design strategies and data structures.

CS-GY 6033
Design and Analysis of Algorithms I

Lecture 1 | Part 3

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?

Approach #1: Timing

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

In [1]:
```
numbers = range(1000)
```

In [3]:
```
%%time
sum(numbers)
```

```
CPU times: user 13 µs, sys: 0 ns, total: 13 µs
Wall time: 13.8 µs
```

Out[3]:
```
499500
```

In [4]:
```
%%timeit
sum(numbers)
```

```
10.8 µs ± 509 ns per loop (mean ± std. dev. of 7 runs, 100000 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**.

CS-GY 6033

Design and Analysis of Algorithms I

Lecture 1 | Part 4

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.

```
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 $T(n)$.
- 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

Which of the below array operations takes constant time?

- accessing an element: `arr[i]` ✓
- asking for the length: `len(arr)` ✓
- finding the max: `max(arr)` ✗

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

	Time/exec.	# of execs.
`def mean(numbers):`		
`    total = 0`	c_1	1
`    n = len(numbers)`	c_2	1
`    for x in numbers:`	c_3	$n+1$
`        total += x`	c_4	n
`    return total / n`	c_5	1

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

Example: `mean`

- Total time:

$$\begin{aligned} T(n) &= c_3(n+1) + c_4 n + (c_1 + c_2 + c_5) \\ &= (c_3 + c_4)n + (c_1 + c_2 + c_3 + c_5) \end{aligned}$$

- "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.

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

Example

	Time/exec.	# of execs.
`def mean_2(numbers):`		
`total = sum(numbers)`	$c_1 n$	1
`n = len(numbers)`	c_2	1
`return total / n`	c_3	1

$$T(n) = c_1 n + c_2 + c_3 \Longrightarrow T(n) = \Theta(n)$$

Example: `mean_2`

- Total time:

$$T(n) = c_1 n + (c_0 + c_2 + c_3)$$

- "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.
`def maximum(numbers):`	c_1	1
`current_max = -float('inf')`	c_2	$n+1$
`for x in numbers:`	c_3	n
`if x > current_max:`	c_4	$\geq 1 \dots n$
`current_max = x`	c_5	1
`return current_max`		

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

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

Main Idea

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

Remaining Questions

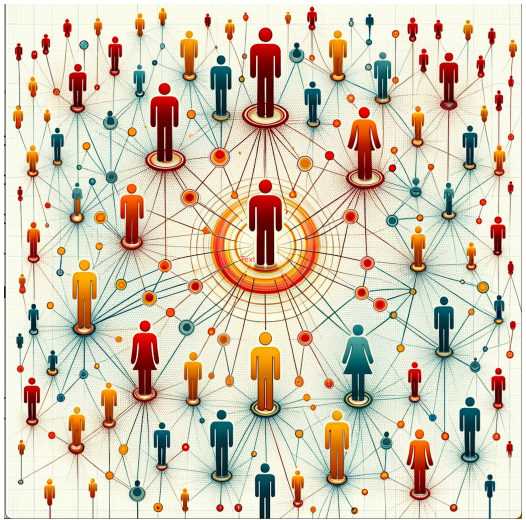
- What if the code is more complex?
  - For example, nested loops.
- What is this notation anyways?

CS-GY 6033
Design and Analysis of Algorithms I

Lecture 1 | Part 5

Nested Loops

Example 1: Influence Maximization



Example 1: Influence Maximization

- Design an algorithm to solve the following:

- Given the influence factor of n people, determine the maximum influence achieved by selecting any two of them?
  - sum of their influence factors is maximized

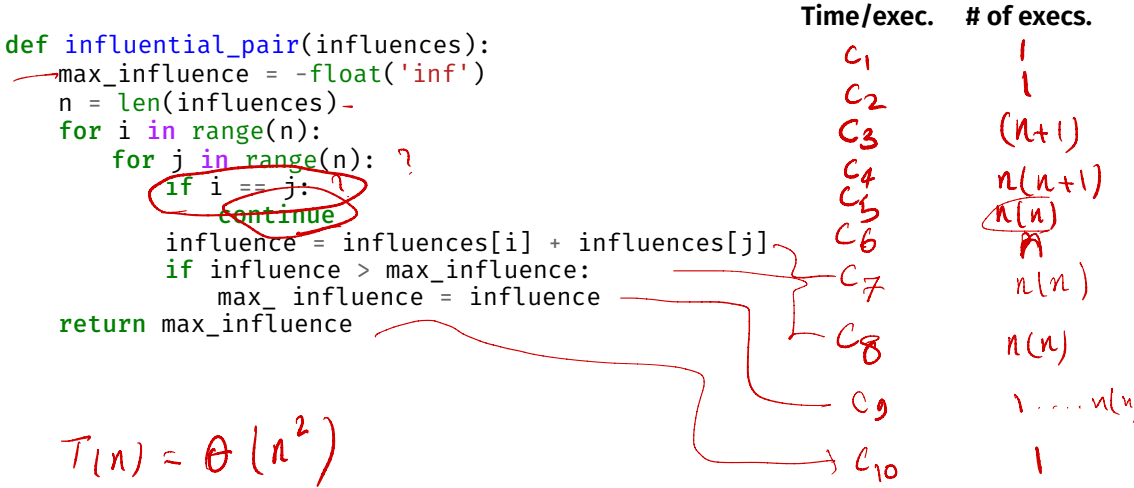
Exercise

- 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 the influence of each pair.
- Keep the best.

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


def influential_pair(influences):
max_influence = -float('inf')
n = len(influences)
for i in range(n):
for j in range(n):
if i == j:
continue
influence = influences[i] + influences[j]
if influence > max_influence:
max_ influence = influence
return max_influence
Time/exec.
# of execs.
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
1
1
(n+1)
n(n+1)
n(n)
n(n)
n(n)
1 . . . n(n)
1
T(n) = Θ(n²)

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):
        influence = influences[i] + influences[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: ___n^2___

$n + n + n + \cdots + n = n \cdot n$

```
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)
```

$3n^3 + 5n^2 - 100 \rightarrow \Theta(n^3)$

$n^6 - n^5 \rightarrow \Theta(n^6)$

$\#ex = n^6 \cdot n^3 = n^9$

$T(n) = \Theta(n^9)$

Example 2: The Median

- **Given:** real numbers $x_1, \ldots, x_n$.
- **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 $x_1, \ldots, x_n$ must be a median.
- **Idea**: compute $R(x_1), R(x_2), \ldots, R(x_n)$, 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
```

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

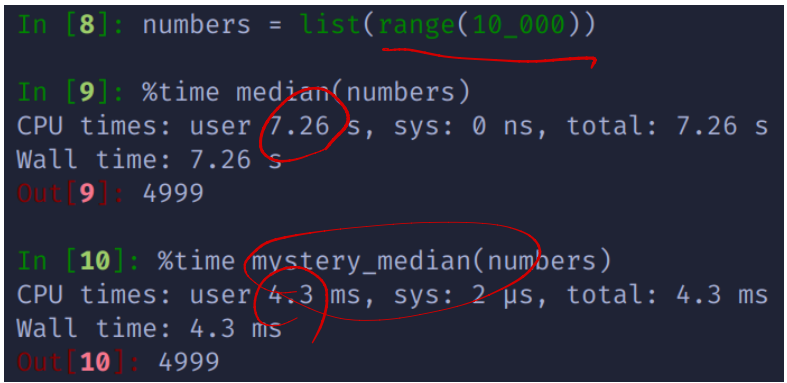
The Median

- 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.[4]

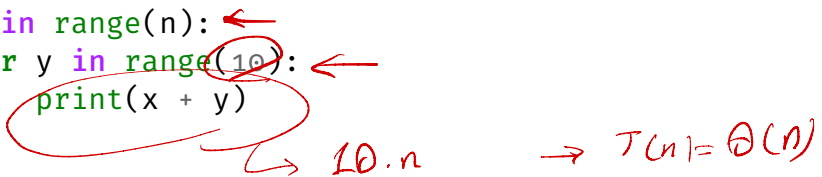
[4] Well, *expected* time.


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)
```



CS-GY 6033
Design and Analysis of Algorithms I

Lecture 1 | Part 6

Dependent Nested Loops

Example 3: Influence Maximization, Again

- Previous algorithm, `influential_pair`, computed influence of 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

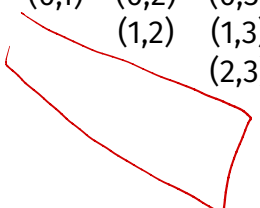
```
for i in range(4):
    for j in range(4):
        print(i, j)
```

(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 influential_pair_2(influences):
2     max_influence = -float('inf')
3     n = len(influences)
4     for i in range(n):
5         for j in range(i + 1, n):
6             influence = influences[i] + influences[j]
7             if influence > max_influence:
8                 max_influence = influence
```

- **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):
        influence = influences[i] + influences[j]
```

- Idea: find formula $f(\alpha)$ for "number of iterations of inner loop during outer iteration α"[5]
- Then total: $\sum_{\alpha=1}^{n} f(\alpha)$

[5] Why α 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):
        influence = influences[i] + influences[j]
```

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

Total: $\sum_{\alpha=1}^{n} n-\alpha =$

Totalling Up

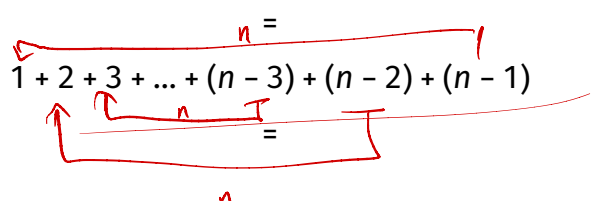
- On outer iteration α, inner body runs $n - \alpha$ times.
  - That is, $f(\alpha) = n - \alpha$
- There are n outer iterations.
- So we need to calculate:

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

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

$$=$$

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



Aside: Arithmetic Sums

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

Time Complexity

- `influential_pair_2` has $\Theta(n^2)$ time complexity
- Same as original `influential_pair`!
- Should we have been able to guess this? Why?

Reason 1: Number of Pairs

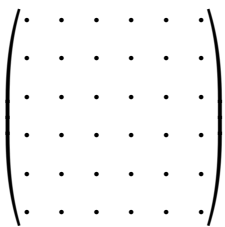
- We're doing constant work for each unordered pair.
- Number of pairs of n objects is

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

- So $\Theta(n^2)$

Reason 2: Half as much work

- Our new solution does roughly half as much work as the old one.
- But Θ doesn't care about constants: $\frac{1}{2}\Theta(n^2)$ is still $\Theta(n^2)$.



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

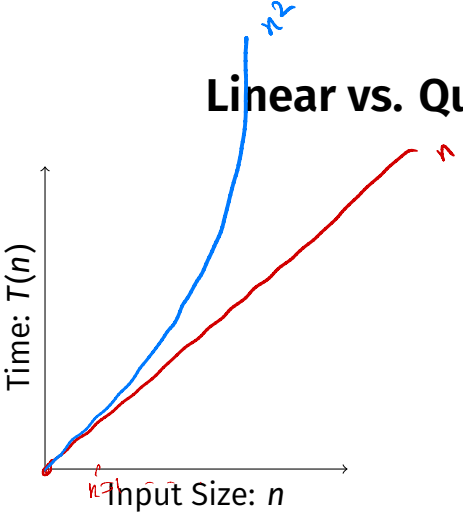
Design a linear time algorithm for this problem.

CS-GY 6033
Design and Analysis of Algorithms I

Lecture 1 | Part 7

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 ≈ 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 ≈ 50,000 seconds.
- i.e., ≈ 14 hours

In data science and ML...

- 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

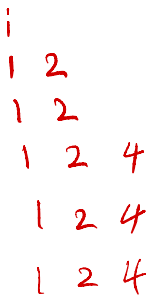
What is the asymptotic time complexity of the code below as a function of n?

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

$O(\log n)$

Solution

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




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

n	# iters.
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4

$T(n) = \log_2 n$

Let k be # iter

$2^k \leq n \Rightarrow k \leq \log n$

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?

$2^n = 2 \times 2 \times \cdots \times 2$

$n! = n \times (n-1) \times (n-2) \cdots \times 2 \times 1$

$n! > 2^n$; $n > 4$