

Table of Contents

Lecture 01 - intro

Lecture 02 - time - complexity - pt - I

Lecture 03 - time - complexity - pt - II

Lecture 04 - expected - time

Lecture 05 - binary - search - recurrences

Lecture 06 - sorting

Lecture 07 - quickselect

Lecture 08 - bst

Lecture 09 - hashing

Lecture 10 - graph - theory

Lecture 11 - bfs - pt - I

Lecture 12 - bfs - pt - II

Lecture 13 - dfs

Lecture 14 - bellman - ford

Lecture 15 - dijkstra

Lecture 16 - prim

Lecture 17 - kruskal

Lecture 18 - np - completeness

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 1

Administrative Stuff

Syllabus

- ▶ All course materials, the syllabus, etc., can be found at dsc40b.com.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 2

What is DSC 40B?

Recall DSC 40A...

- ▶ How do we **formalize** learning from data?
- ▶ How do we turn it into something a **computer** can do?

Example 1: Minimize Absolute Error

- ▶ **Goal:** summarize a collection of numbers,
 x_1, \dots, 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, \dots, x_n .

The End

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 bunch of 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 (*a la* DSC 40A).

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

We need an **algorithm**.

Main Idea

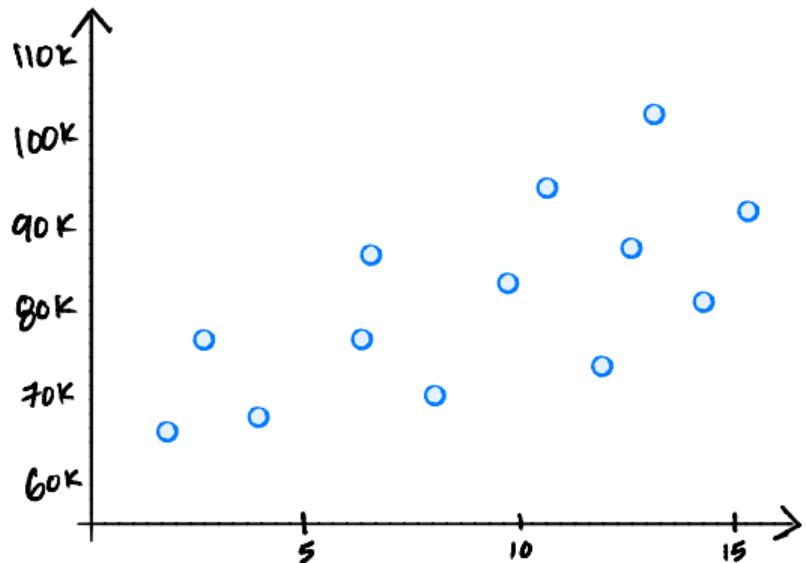
Our work doesn't stop once we solve the math problem (*a la* DSC 40A).

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

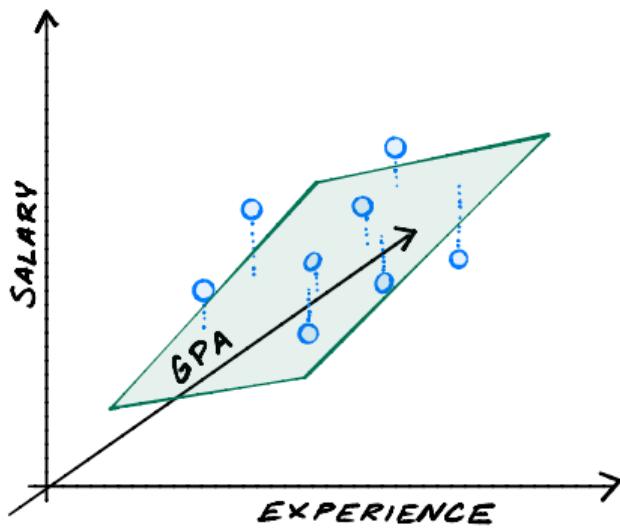
We need an **algorithm**.

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

Example 2: Least Squares Regression



Example 2: Least Squares Regression



The End

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

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.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 3

Example: Clustering

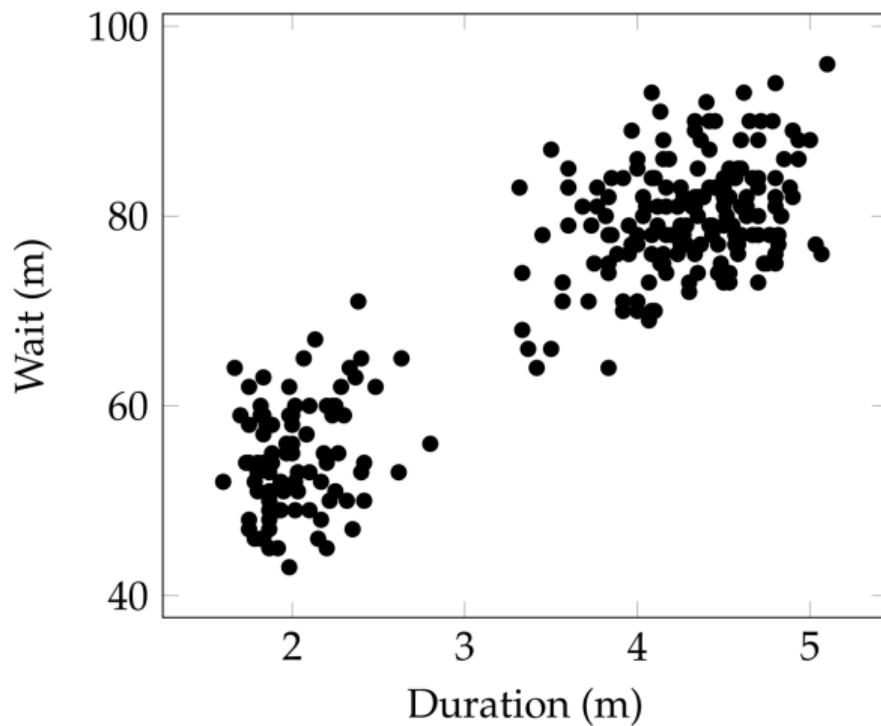
Clustering

- ▶ 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...

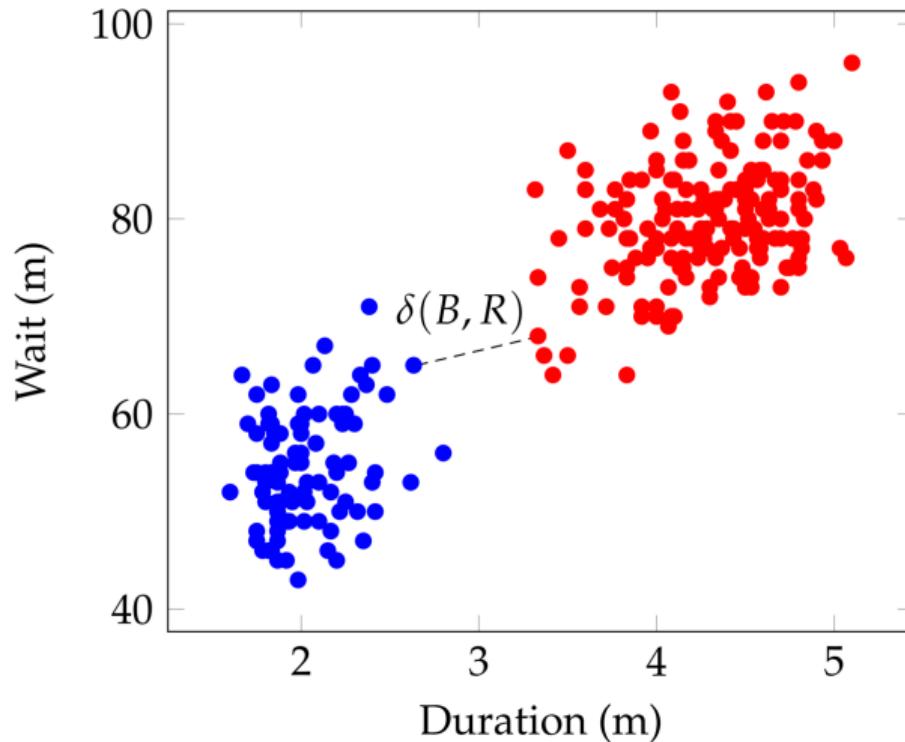
Example: Old Faithful



Example: Old Faithful



Example: Old Faithful

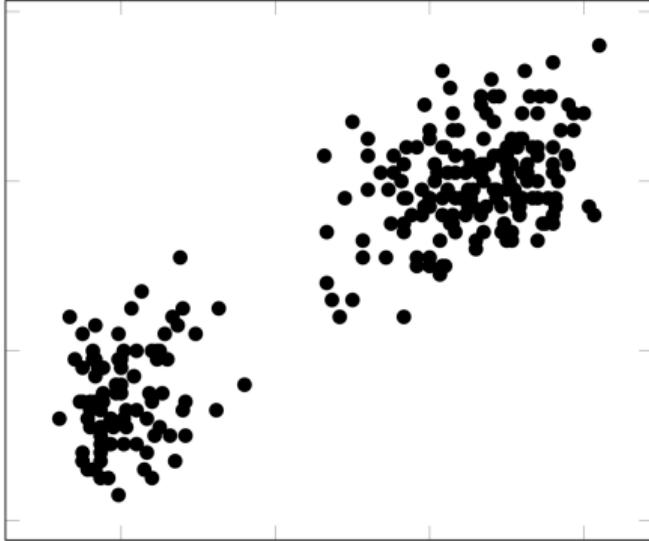


Clustering

- ▶ 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

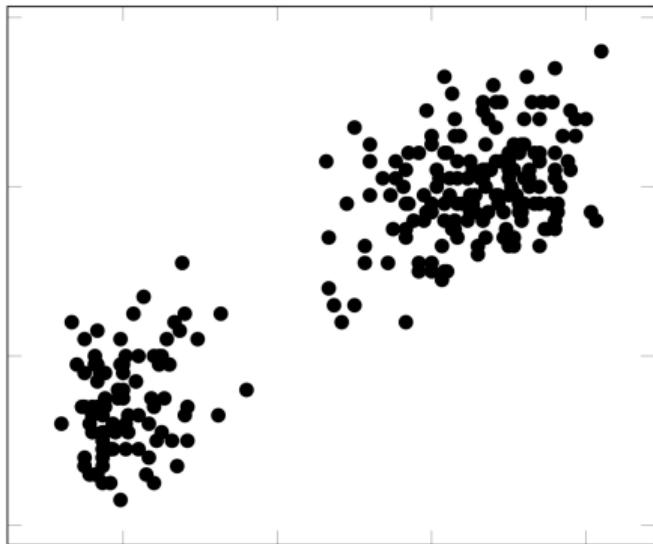
- ▶ How do we turn this into something a **computer** can do?
- ▶ DSC 40A says: “Turn it into an optimization problem”.
- ▶ Idea: **design** a way of quantifying the “goodness” of a clustering; find the **best**.
 - ▶ Design a **loss function**.
 - ▶ There are many possibilities, tradeoffs!



Exercise

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

Quantifying Separation



Define the “separation” $\delta(B, R)$ to be the smallest distance between a blue point and red point.

The Problem

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

DSC 40A: “The End”

DSC 40A: “The End”

DSC 40B: “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 ¹ for each object: $2 \times 2 \times \dots \times 2 = 2^n$.

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

Time

- ▶ Suppose it takes at least 1 nanosecond^2 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.

²This 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

| <i>n</i> | Time |
|----------|---------------|
| 1 | 1 nanosecond |
| 10 | 1 microsecond |
| 20 | 1 millisecond |
| 30 | 1 second |

Time Needed

| <i>n</i> | Time |
|----------|---------------|
| 1 | 1 nanosecond |
| 10 | 1 microsecond |
| 20 | 1 millisecond |
| 30 | 1 second |
| 40 | 18 minutes |

Time Needed

| <i>n</i> | Time |
|----------|---------------|
| 1 | 1 nanosecond |
| 10 | 1 microsecond |
| 20 | 1 millisecond |
| 30 | 1 second |
| 40 | 18 minutes |
| 50 | 13 days |

Time Needed

| <i>n</i> | Time |
|----------|---------------|
| 1 | 1 nanosecond |
| 10 | 1 microsecond |
| 20 | 1 millisecond |
| 30 | 1 second |
| 40 | 18 minutes |
| 50 | 13 days |
| 60 | 36 years |

Time Needed

| <i>n</i> | 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.

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.
- ▶ We'll see an efficient solution by the end of the quarter.

DSC 40B

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

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 4

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?
- ▶ $5 \text{ seconds} \times 100 = 500 \text{ seconds?}$
- ▶ 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

```
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**.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 5

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. |
|--|------------|-------------|
| <pre>def mean(numbers): total = 0 n = len(numbers) for x in numbers: total += x return total / n</pre> | | |

Example: mean

- ▶ Total time:

$$\begin{aligned}T(n) &= c_3(n + 1) + c_4n + (c_1 + c_2 + c_3) \\&= (c_3 + c_4)n + (c_1 + c_2 + c_3)\end{aligned}$$

- ▶ “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.

Careful!

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

Example

| | Time/exec. | # of execs. |
|--|------------|-------------|
| <pre>def mean_2(numbers): total = sum(numbers) n = len(numbers) return total / n</pre> | | |

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?

Time/exec. # of execs.

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

Main Idea

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

By the way...

Approximate timing for various operations on a typical PC:

| | |
|-------------------------------------|--|
| execute typical instruction | 1/1,000,000,000 sec = 1 nanosec |
| fetch from L1 cache memory | 0.5 nanosec |
| branch misprediction | 5 nanosec |
| fetch from L2 cache memory | 7 nanosec |
| Mutex lock/unlock | 25 nanosec |
| fetch from main memory | 100 nanosec |
| send 2K bytes over 1Gbps network | 20,000 nanosec |
| read 1MB sequentially from memory | 250,000 nanosec |
| fetch from new disk location (seek) | 8,000,000 nanosec |
| read 1MB sequentially from disk | 20,000,000 nanosec |
| send packet US to Europe and back | 150 milliseconds = 150,000,000 nanosec |

From Peter Norvig's essay, "Teach Yourself Programming in Ten Years"
<http://norvig.com/21-days.html>

Remaining Questions

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

Remaining Questions

- ▶ What if the code is more complex?
 - ▶ For example, nested loops.
- ▶ What is this notation anyways?
- ▶ Next time in DSC 40B.

DSC 40B

Theoretical Foundations II

Lecture 2 | Part 1

News

News

- ▶ Lab 01 posted
 - ▶ Due Sunday @ 11:59 pm PST on Gradescope.
- ▶ Homework 01 posted
 - ▶ Due Wednesday @ 11:59 pm PST on Gradescope.
 - ▶ LaTeX template available.

Agenda

1. Analyzing nested loops.
2. What is Θ 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. |
|---|------------|-------------|
| <pre>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</pre> | | |

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 we'll find a line that **dominates** time complexity; i.e., yields the leading term of $T(n)$.

Totalling Up

```
for i in range(n):
    for j in range(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. # α , inner body runs _____ times.
- ▶ The outer loop body runs _____ times.

```
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.

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

- ▶ 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.¹

¹Well, *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] + height[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^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 α ”²

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

²Why 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):
        height = heights[i] + heights[j]
```

- ▶ On outer iter. # 1, inner body runs _____ times.
- ▶ On outer iter. # 2, inner body runs _____ times.
- ▶ On outer iter. # α , inner body runs _____ times.
- ▶ The outer loop body runs _____ times.

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:

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

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

=

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

=

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

=

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 $\Theta(n^2)$

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.

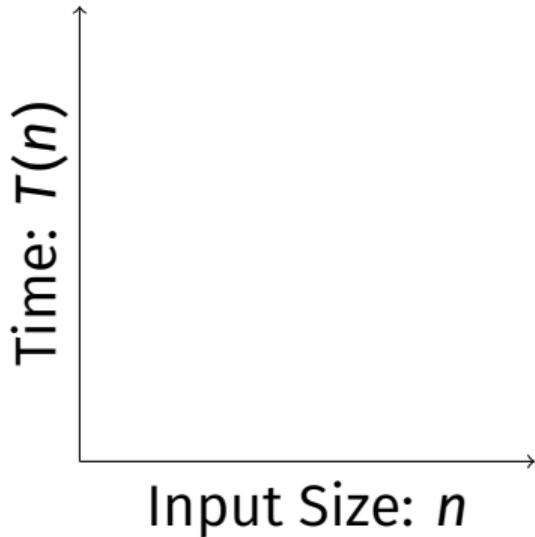
DSC 40B

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^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
```

| n | # 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 Θ -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” $g(n)$.

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

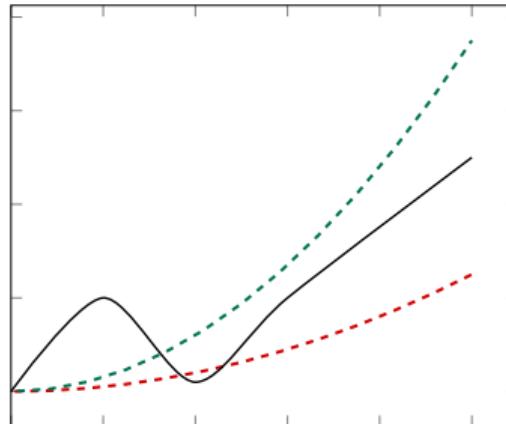
Theta Notation Examples

- ▶ $4n^2 + 3n - 20 = \Theta(n^2)$
- ▶ $3n + \sin(4\pi n) = \Theta(n)$
- ▶ $2^n + 100n = \Theta(2^n)$

Definition

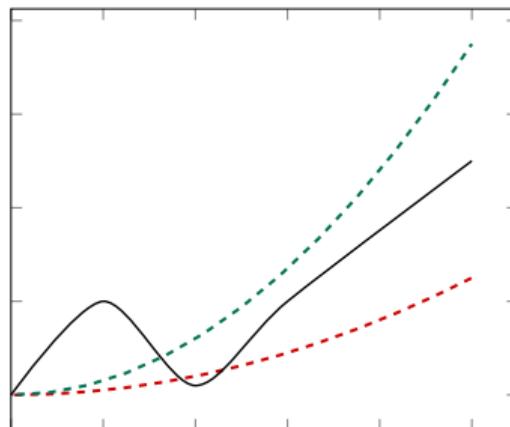
We write $f(n) = \Theta(g(n))$ if there are positive constants N , c_1 and c_2 such that for all $n \geq N$:

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



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.

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

- ▶ 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:

Example

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

- ▶ Lower bound:

Upper-Bounding Tips

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

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

- ▶ “Drop” **negative** terms

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

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.

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

$n^3 - 10n^2 \geq 0$ 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 of the following are true?

- ▶ $f = \Theta(n)$
- ▶ $f = \Theta(n^2)$
- ▶ $f = \Theta(n \sin(\pi n))$
- ▶ $f = \Theta(n^2(n \sin(\pi n) + 3))$

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 1

Big Theta, Formalized

Today in DSC 40B...

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

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

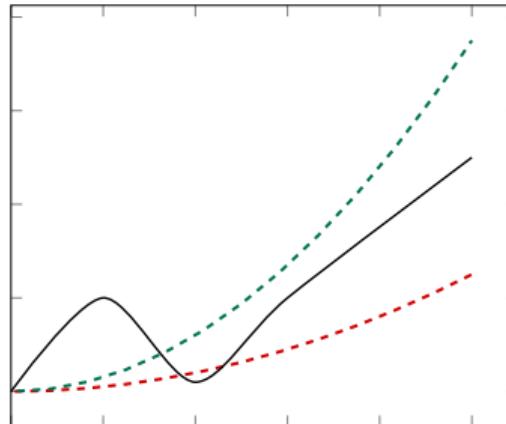
- ▶ $f(n) = \Theta(g(n))$ if $f(n)$ “grows like” $g(n)$.

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

Definition

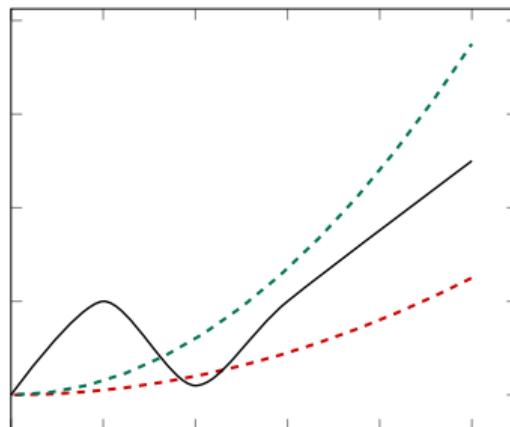
We write $f(n) = \Theta(g(n))$ if there are positive constants N , c_1 and c_2 such that for all $n \geq N$:

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



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.

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

$n^3 - 10n^2 \geq 0$ 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 2

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.

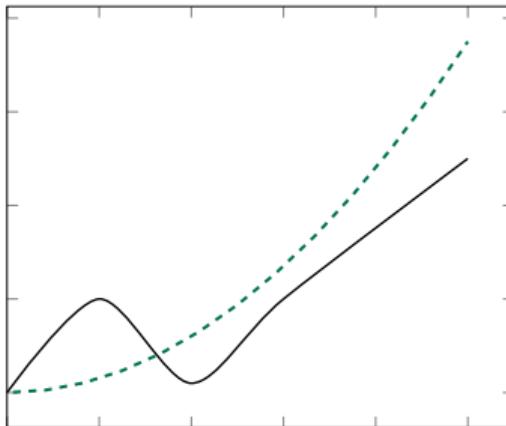
Big-O Notation, Informally

- ▶ 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 $n \geq N$:

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



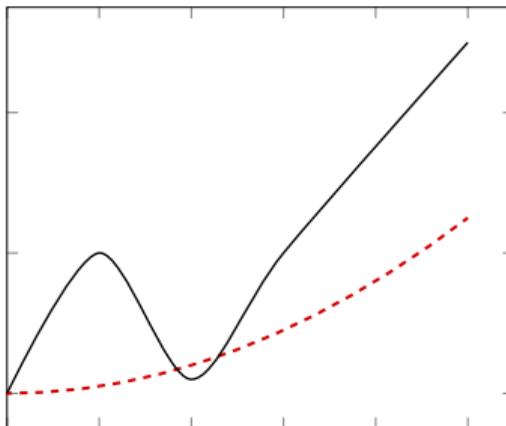
Big-Omega Notation

- ▶ 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:
 - ▶ $\Theta \implies (O \text{ and } \Omega)$
 - ▶ $(O \text{ and } \Omega) \implies \Theta$

Analogies

- ▶ 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 3

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 4

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 $f(n) = c_3 n^3 + c_2 n^2 + c_1 n + c_0$.

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”.
- ▶ $T_1(n) = 1,000,000n = \Theta(n)$
- ▶ $T_2(n) = 0.00001n^2 = \Theta(n^2)$
- ▶ 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 5

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 6

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 view of the performance of the algorithm.

DSC 40B

Theoretical Foundations II

Lecture 3 | Part 7

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 8

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:

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

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 1

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.
- ▶ **Example:** arr = [-3, -7, 2, 9, 1, 4]

```
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 asymptotic 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 asymptotic 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 the following code?

```
def foo(arr):
    n = len(arr)
    for x in arr:
        for y in arr:
            if x + y == 10:
                return sum(arr)
```

Best Case

- ▶ When the first element is 5, so $x + y = 10$.
- ▶ `sum(arr)` takes $\Theta(n)$ time.
- ▶ Exits, taking $\Theta(n)$ time in total.

Worst Case

- ▶ No two elements sum to 10.
- ▶ Has to loop over all $\Theta(n^2)$ pairs.
- ▶ Worst case time complexity: $\Theta(n^2)$.
- ▶ **Note:** it's not $\Theta(n^3)$, since the `sum(arr)` only runs once!

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 view of the performance of the algorithm.

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 2

Average Case

Time Taken, Typically

- ▶ Best case and worst case can be **misleading**.
 - ▶ Depend on a **single good/bad input**.
- ▶ How much time is taken, typically?
- ▶ **Idea:** compute the average time taken over all possible inputs.

Recall: The Expectation

- ▶ The expected value of a random variable X is:

$$\sum_x x \cdot P(X = x)$$

| winnings | probability |
|----------|-------------|
| \$ 0 | 50% |
| \$ 1 | 30% |
| \$ 10 | 18% |
| \$ 50 | 2% |

Expected winnings:

Average Case

- ▶ We'll compute the expected time over all cases:

$$T_{\text{avg}}(n) = \sum_{\text{case} \in \text{all cases}} P(\text{case}) \cdot T(\text{case})$$

- ▶ Called the **average case time complexity**.

Strategy for Finding Average Case

- ▶ **Step 0:** Make assumption about distribution of inputs.
- ▶ **Step 1:** Determine the possible cases.
- ▶ **Step 2:** Determine the probability of each case.
- ▶ **Step 3:** Determine the time taken for each case.
- ▶ **Step 4:** Compute the expected time (average).

Example: Linear Search

- ▶ Recall **linear search**:

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

- ▶ Best case? Worst case?

Example: Linear Search

- ▶ What is the **average case time complexity** of **linear search**?

Step 0: Assume input distribution

- ▶ We must assume something about the input.
- ▶ Example: Target must be in array, equally-likely to be any element, no duplicates.
- ▶ This is usually given to you.

Step 1: Determine the Cases

- ▶ Example: linear search.
 - Case 1: target is first element
 - Case 2: target is second element
 - :
 - Case n : target is n th element
 - Case $n + 1$: target is not in array

Step 2: Case Probabilities

- ▶ What is the probability that we see each case?
 - ▶ Example: what is the probability that the target is the k th element?
- ▶ This is where we use assumptions from Step 0.

Example

- ▶ **Assume:** target is in the array exactly once, equally-likely to be any element.
- ▶ Each case has probability $1/n$.

Step 3: Case Times

- ▶ Determine time taken in each case.
- ▶ Example: linear search.
 - ▶ Let's say it takes time c per iteration.

Case 1: time c

Case 2: time $2c$

⋮

Case i : time $c \cdot i$

⋮

Case n : time $c \cdot n$

Step 4: Compute Expectation

$$T_{\text{avg}}(n) = \sum_{i=1}^n P(\text{case } i) \cdot T(\text{case } i)$$

Average Case Time Complexity

- ▶ The **average case** time complexity¹ of **linear search** is $\Theta(n)$.

¹Under these assumptions on the input!

Note

- ▶ **Worst case** time complexity is still useful.
- ▶ Easier to calculate.
- ▶ Often same as average case (but not always!)
- ▶ Sometimes worst case is very important.
 - ▶ Real time applications, time complexity attacks

Note

- ▶ **Hard** to make realistic assumptions on input distribution.
- ▶ Example: linear search.
 - ▶ Is it realistic to assume t is in array?
 - ▶ If not, what is the probability that t *is* in the array?

Exercise

Suppose we change our assumptions:

- ▶ The target has a 50% chance of being in the array.
- ▶ If it is in the array, it is equally-likely to be any element.

What is the average case complexity now?

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 3

Average Case in Movie Problem

Recall: 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**.

```
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

- ▶ Best case: $\Theta(1)$
 - ▶ When the first pair of movies checked equals target.
- ▶ Worst case: $\Theta(n^2)$
 - ▶ When no pair of movies equals target.

“Average” Case?

- ▶ The best and worst cases are **extremes**.
- ▶ How much time is taken, *typically*?
 - ▶ That is, when the target pair is not the first checked nor the last, but somewhere in the middle.

Exercise

How much time do you expect `find_movies` to take on a typical input?

- ▶ $\Theta(1)$
- ▶ $\Theta(n^2)$
- ▶ Something in between, like $\Theta(n)$

The Movie Problem

```
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

- ▶ Best case: $\Theta(1)$
- ▶ Worst case: $\Theta(n^2)$
- ▶ Average case: $\Theta(?)$

Step 0: Assume input distribution

- ▶ Suppose we are told that:
 - ▶ There is a unique pair of movies that add to t .
 - ▶ All pairs are equally likely.

Step 1: Determine the Cases

- ▶ Case α : the α th pair checked sums to t .
- ▶ Each pair of movies is a case.
- ▶ There are $\binom{n}{2}$ cases.

Step 2: Case Probabilities

- ▶ **Assume:** there is a *unique* pair that adds to t.
- ▶ **Assume:** all pairs are equally likely.
- ▶ Probability of any case: $\frac{1}{\binom{n}{2}} = \frac{2}{n(n-1)}$

Step 3: Case Time

- ▶ How much time is taken for a particular case?
- ▶ Example, suppose the movies a and b sum to the target.
- ▶ How long does it take to find this pair?

```
1 def find_movies(movies, t):
2     n = len(movies)
3     for i in range(n):
4         for j in range(i + 1, n):
5             if movies[i] + movies[j] == t:
6                 return (i, j)
7
return None
```

Exercise

Roughly how much time is taken (how many times does line 5 run) if the i th pair checked sums to the target?

Step 4: Compute Expectation

Average Case

- ▶ The average case time complexity of `find_movies` is $\Theta(n^2)$.
- ▶ Same as the worst case!

Note

- ▶ We've seen two algorithms where the average case = the worst case.
- ▶ Not always the case!
- ▶ Interpretation: the worst case is not too extreme.

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 4

Expected Time Complexity

Example: Contrived Algorithm

```
def wibble(n):
    # generate random number between 0 and n
    x = np.random.randint(0, n)

    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

Exercise

How much time does wibble take on average?

Random Algorithms

- ▶ This algorithm is *randomized*.
- ▶ The time it takes is also *random*.
- ▶ What is the **expected time**?

Average Case vs. Expected Time

- ▶ With average case complexity, a probability distribution on inputs is specified.
- ▶ Now, the randomness is *in the algorithm itself*.
- ▶ Otherwise, the analysis is very similar.

Step 1: Determine the cases

```
def wibble(n):
    x = np.random.randint(0, n)

    if x == 0:                                ▶ Case 1: x == 0
        for i in range(n):
            print('Unlucky!')
    else:                                     ▶ Case 2: x != 0
        print('Lucky!')
```

Step 2: Determine case probabilities

```
def wibble(n):
    x = np.random.randint(0, n)
    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

▶ $P(\text{Case 1}) = 1/n$

▶ $P(\text{Case 2}) = (n - 1)/n$

Step 3: Determine case times

```
def wibble(n):
    x = np.random.randint(0, n)
    if x == 0:
        for i in range(n):
            print('Unlucky!')
    else:
        print('Lucky!')
```

▶ Case 1: $\Theta(n)$

▶ Case 2: $\Theta(1)$

Step 4: Compute expectation

- ▶ Compute expected time:

Expected Time

- ▶ This was a contrived example.
- ▶ Some important algorithms involve randomness!
 - ▶ Quicksort
 - ▶ We'll see alg. for median with $\Theta(n)$ expected time.

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 5

Lower Bound Theory

Imagine...

- ▶ You write a simple algorithm to solve a problem.
- ▶ You analyze time complexity and find it is $\Theta(n^2)$.
- ▶ You ask yourself: *can I do better than $\Theta(n^2)$?*
- ▶ Or: *What is the best time complexity possible?*

Doing Better

- ▶ How can you know what you don't know?
- ▶ You can argue that *any* algorithm for solving the problem *must* take at least a certain amount of time in the worst case.

Example: Minimum

- ▶ Problem: Find minimum in array of length n .
- ▶ Any algorithm has to check all n numbers in the worst case.
 - ▶ Or else the number not checked could have been the smallest!
- ▶ Takes at least linear ($\Omega(n)$) time.
 - ▶ **No algorithm** for the min can have worst case of < linear time.

Definition

A **theoretical lower bound** is a lower bound on the worst-case time complexity of **any algorithm** solving a particular problem.

Main Idea

No algorithm's worst case can be better than theoretical lower bound.

Loose Lower Bounds

- ▶ $\Omega(\log n)$, $\Theta(\sqrt{n})$ and $\Theta(1)$ are also theoretical lower bounds for finding the minimum.
- ▶ But no algorithm can exist which has a worst case of $\Theta(\log n)$, $\Theta(\sqrt{n})$, or $\Theta(1)$.
- ▶ This bound is **loose**. Not super useful.

Tight Lower Bounds

- ▶ A lower bound is **tight** if there exists an algorithm with that worst case time complexity.
- ▶ That algorithm is (in a sense) **optimal**.

How to find a TLB

- ▶ Argument from completeness:
 - ▶ The algorithm might not be correct if it doesn't check k things, so the time is $\Omega(k)$.
- ▶ Argument from I/O:
 - ▶ If the output is an array of size k , time taken is $\Omega(k)$
- ▶ More sophisticated arguments...

Tight Bounds can be difficult to find

- ▶ Often require sophisticated combinatorial arguments outside of the scope of DSC 40B.

Assumptions make problems easier

- ▶ The TLB for finding a minimum changes if we assume that the array is sorted.

Exercise

Consider these two problems:

1. Find the min of a sorted array.
2. Given a target t and a sorted array, determine whether t is in the array.

Find tight theoretical lower bounds for each problem.

Main Idea

When coming up with an algorithm, first try to find a tight TLB. Then try to make an algorithm which has that worst-case complexity. If you can, it's **optimal!**

DSC 40B

Theoretical Foundations II

Lecture 4 | Part 6

Case Study: Matrix Multiplication

It's Important

- ▶ Matrix multiplication is a *very* common operation in machine learning algorithms.
- ▶ **Estimate:** 75% - 95% of time training a neural network is spent in matrix multiplication.

Recall

- ▶ If A is $m \times p$ and B is $p \times n$, then AB is $m \times n$.
- ▶ The ij entry of AB is

$$(AB)_{ij} = \sum_{k=1}^p a_{ik} b_{kj}$$

Recall

$$(AB)_{ij} = \sum_{k=1}^p a_{ik} b_{kj}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 5 & -1 \\ 1 & 7 \\ -2 & -3 \end{pmatrix} = \begin{pmatrix} \quad & \quad \\ \quad & \quad \\ \quad & \quad \end{pmatrix}$$

Naïve Algorithm

- ▶ This algorithm is relatively straightforward to code up.

```
def mmul(A, B):
    """
    A is (m x p) and B is (p x n)
    """
    m, p = A.shape
    n = B.shape[1]

    C = np.zeros((m, n))

    for i in range(m):
        for j in range(n):
            for k in range(p):
                C[i, j] += A[i, k] * B[k, j]

    return C
```

Time Complexity

- ▶ The naïve algorithm takes time $\Theta(mnp)$.
- ▶ If both matrices are $n \times n$, then $\Theta(n^3)$ time.
- ▶ **Cubic!**

Cubic Time Complexity

- ▶ The largest problem size that can be solved, if a basic operation takes 1 nanosecond.

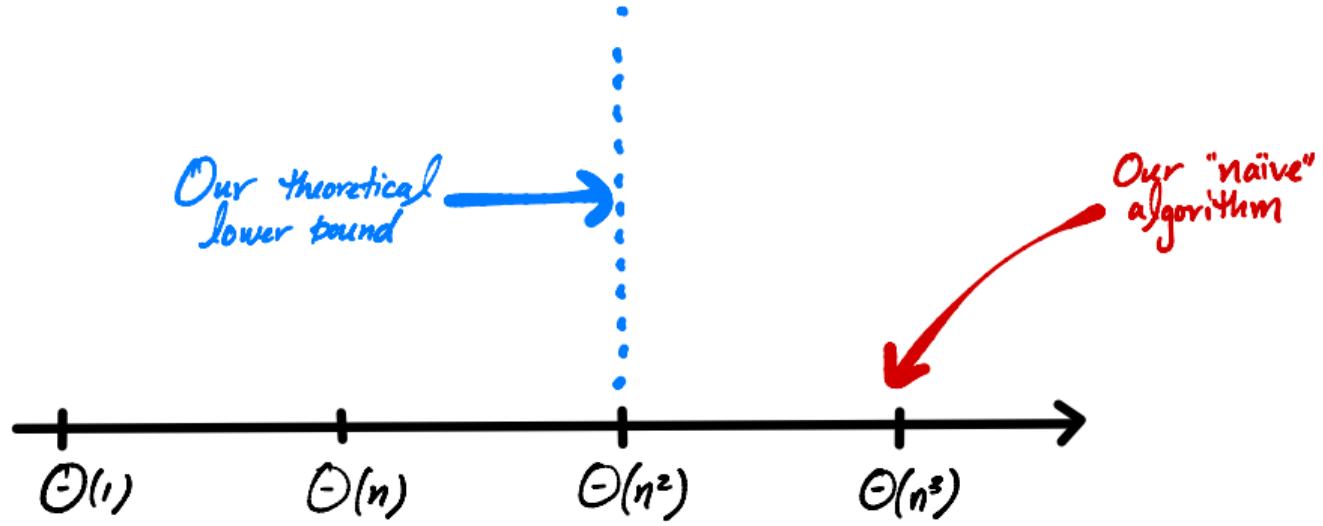
| 1 s | 10 m | 1 hr |
|-------|-------|--------|
| 1,000 | 6,694 | 15,326 |

The Question

- ▶ Can we do better?
- ▶ How fast can we possibly multiply matrices?

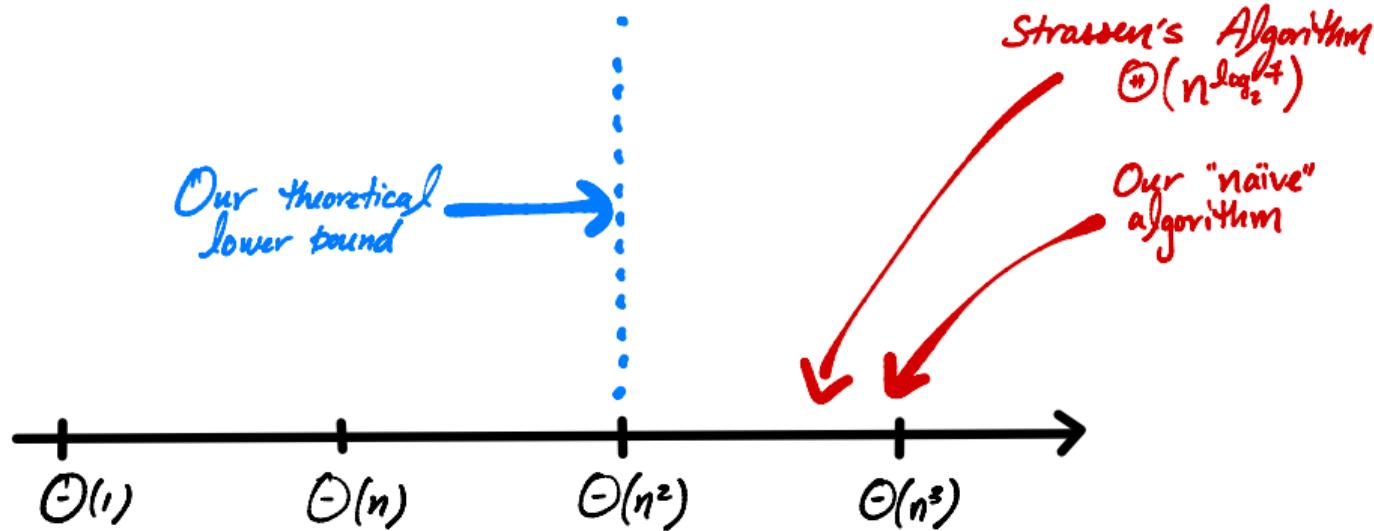
Theoretical Lower Bound

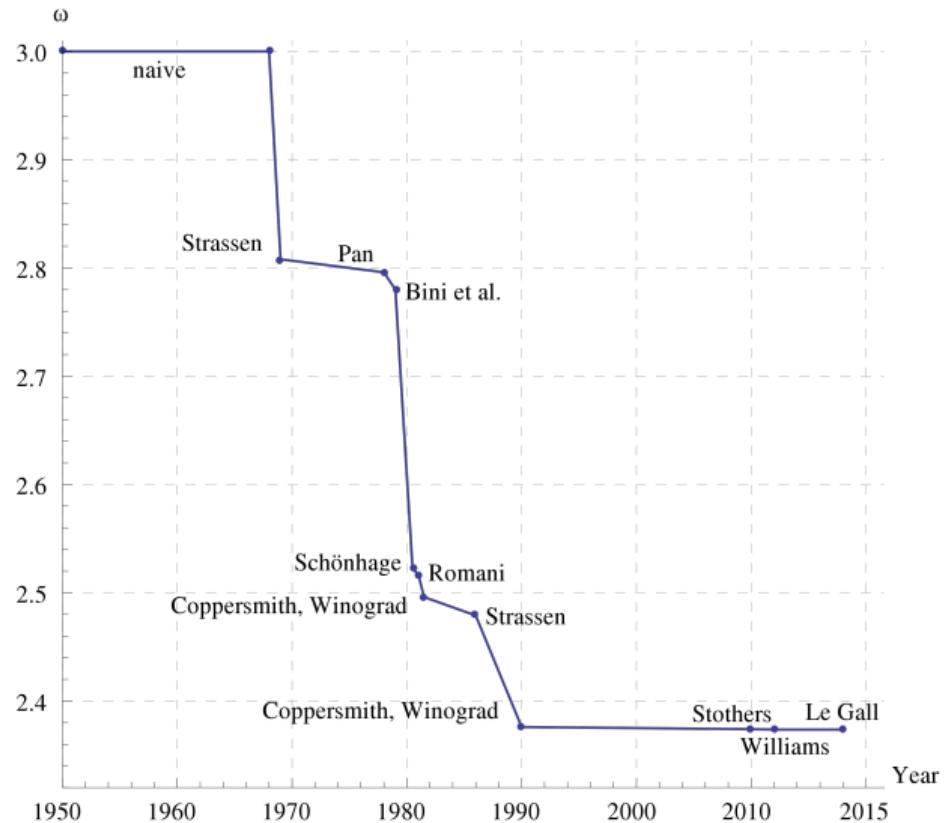
- ▶ If A and B are $n \times n$, C will have n^2 entries.
- ▶ Each entry must be filled: $\Omega(n^2)$ time.
- ▶ That is, matrix multiplication must take at least quadratic time.
- ▶ Is this bound **tight**? Can it be increased?



Strassen's Algorithm

- ▶ Cubic was as good as it got...
- ▶ ...until Strassen, 1969.
- ▶ Time complexity: $\Theta(n^{\log_2 7}) = \Theta(n^{2.8073})$

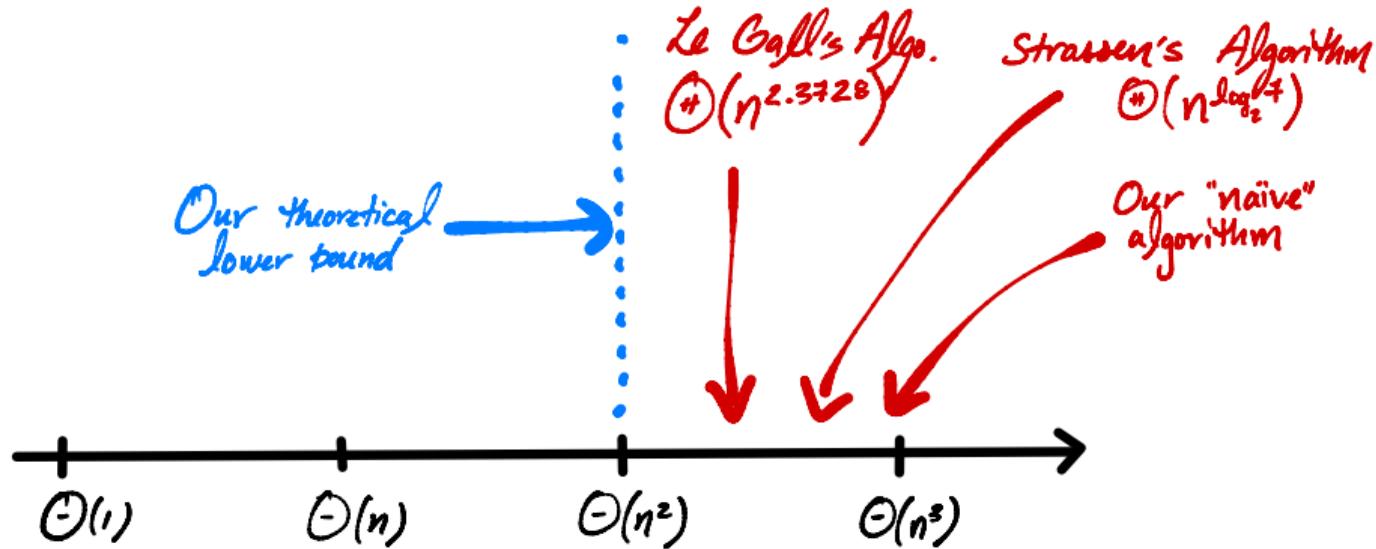




Currently

- ▶ The fastest² known matrix multiplication algorithm is due to Le Gall.
- ▶ $\Theta(n^{2.3728639})$ time.

²In terms of asymptotic time complexity.



Interestingly...

- ▶ No one knows what the lowest possible time complexity is.
- ▶ It could be $\Theta(n^2)$!
- ▶ The “best” matrix multiplication algorithm is probably still undiscovered.

Irony

- ▶ There are many matrix multiplication algorithms.
- ▶ How fast is numpy's matrix multiply?

Irony

- ▶ There are many matrix multiplication algorithms.
- ▶ How fast is numpy's matrix multiply?
- ▶ $\Theta(n^3)$.

Why?

- ▶ Strassen *et al.* have better asymptotic complexity.
- ▶ But much (much!) larger “hidden constants”.
- ▶ Remember, which is better for small n :
 $999,999n^2$ or n^3 ?

Optimization

- ▶ Numpy, most others use **highly optimized** cubic time algorithms³

³The constant c in $T(n) = cn^3 + \dots$ is actually much less than 1, as can be verified by timing.

Main Idea

No one knows what the lowest possible time complexity of matrix multiplication is, and some algorithms are approaching $\Theta(n^2)$.

But most useful implementations take $\Theta(n^3)$ time.

DSC 40B

Theoretical Foundations II

Lecture 5 | Part 1

Searching a Database

Today in DSC 40B...

- ▶ How do we analyze the time complexity of **recursive** algorithms?
- ▶ How do we know that our recursive code is **correct**?

Databases

- ▶ Large data sets are often stored in **databases**.

| PID | FullName | Level |
|-------|---------------|-------|
| A1843 | Wan Xuegang | SR |
| A8293 | Deveron Greer | SR |
| A9821 | Vinod Seth | FR |
| A8172 | Aleix Bilbao | JR |
| A2882 | Kayden Sutton | SO |
| A1829 | Raghu Mahanta | FR |
| A9772 | Cui Zemin | SR |
| : | : | : |

Query

- ▶ What is the name of the student with PID A8172?

Linear Search

- ▶ We *could* answer this with a linear search.
- ▶ Recall worst-case time complexity: $\Theta(n)$.
- ▶ Is there a better way?

Theoretical Lower Bounds

- ▶ **Given:** an array `arr` and a target `t`, determine the index of `t` in the array.
- ▶ Lower bound: $\Omega(n)$
 - ▶ `linear_search` has the best possible worst-case complexity!

Theoretical Lower Bounds

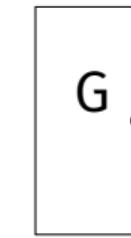
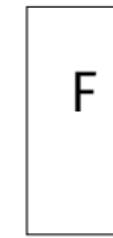
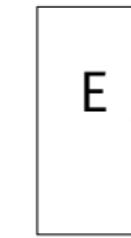
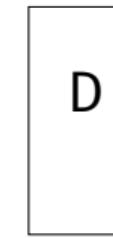
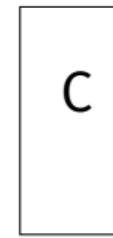
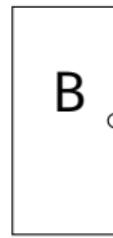
- ▶ **Given:** an **sorted** array arr and a target t, determine the index of t in the array.
- ▶ This is an **easier** problem.
- ▶ Lower bound: $\Omega(?)$

DSC 40B

Theoretical Foundations II

Lecture 5 | Part 2

Binary Search



22

84

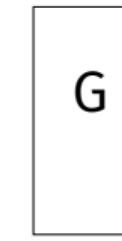
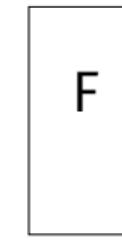
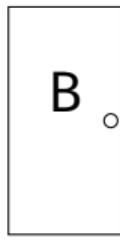
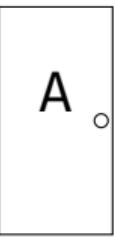
101

14

19

42

20



Game Show

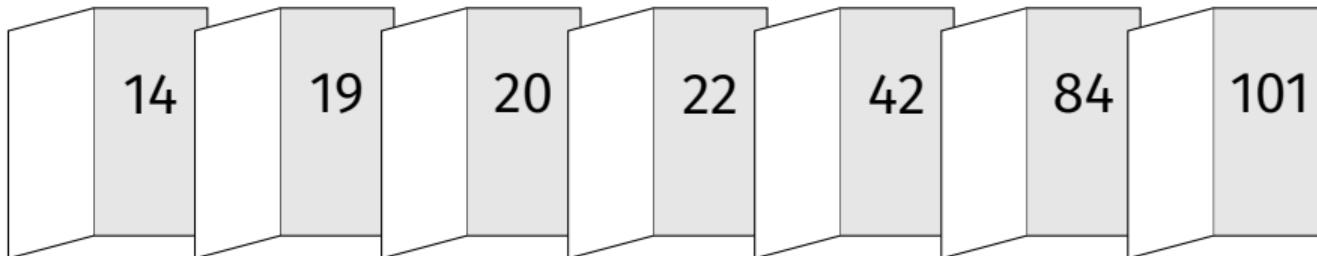
- ▶ **Goal:** guess the door with number 42 behind it.
- ▶ **Caution:** with every wrong guess, your winnings are reduced.

Strategy

- ▶ Can't do much better than linear search.
 - ▶ "Is it door A?"
 - ▶ "OK, is it door B?"
 - ▶ "Door C?"
- ▶ After an incorrect first guess, the right door could be any of the other $n - 1$ doors!

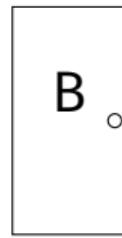
But now...

- ▶ Suppose the host tells you that the numbers are **sorted** in increasing order.



Exercise

Which door do you pick first?



A .

B .

C .

22

E .

F .

G .

A °

B °

C °
22

E °
84

G °

A.

B.

C.

22

42

84

G.

Strategy

- ▶ First pick the middle door.
- ▶ Allows you to rule out half of the other doors.
- ▶ Pick door in the middle of what remains.
- ▶ Repeat, **recursively**.

Binary Search in Code

```
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = _____ # index of the middle element
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, _____, _____)
    else:
        return binary_search(arr, t, _____, _____)
```

Exercise

Fill in the blanks:

```
def binary_search(arr, t, start, stop):
    """
        Searches arr[start:stop] for t.
        Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = _____ # index of the middle element
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, _____, _____)
    else:
        return binary_search(arr, t, _____, _____)
```

The Middle Element

- ▶ What is the index of the middle element of `arr[start:stop]`?

| | | | | | | | | | | |
|-----|----|----|---|---|---|----|----|----|----|----|
| -10 | -6 | -3 | 1 | 2 | 5 | 12 | 21 | 33 | 35 | 42 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Definition

The **floor** of a real number x , denoted $[x]$, is the largest integer that is $\leq x$.

Examples: $[3.14] = 3$ $[-4.5] = -5$ $[10] = 10$

In \LaTeX , $[x]$ is written: “\lfloor x \rfloor”.

Definition

The **ceiling** of a real number x , denoted $[x]$, is the *smallest integer that is $\geq x$* .

Examples: $[3.14] = 4$ $[-4.5] = -4$ $[10] = 10$

In \LaTeX , $[x]$ is written: “\lceil x \rceil”.

Binary Search

```
import math
def binary_search(arr, t, start, stop):
    """
        Searches arr[start:stop] for t.
        Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

t = 21

| | | | | | | | | | | |
|-----|----|----|---|---|---|----|----|----|----|----|
| -10 | -6 | -3 | 1 | 2 | 5 | 12 | 21 | 33 | 35 | 42 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Aside: Default Arguments

```
import math
def binary_search(arr, t, start=0, stop=None):
    if stop is None:
        stop = len(arr)
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

DSC 40B

Theoretical Foundations II

Lecture 5 | Part 3

Thinking Inductively

Recursion

- ▶ Recursive algorithms can almost look like **magic**.
- ▶ How can we be sure that `binary_search` works?

Tips

1. Make sure algorithm works in the **base case**.
2. Check that all recursive calls are on **smaller problems**.
3. **Assuming** that the recursive calls work, does the whole algorithm work?

Base Case

- ▶ Smallest input for which you can easily see that the algorithm works.
- ▶ Recursion works by making problem smaller until base case is reached.
- ▶ Usually $n = 0$ or $n = 1$ (or even both!)

Base Case: $n = 0$

- ▶ Suppose `arr[start:stop]` is empty.
- ▶ In this case, the function returns **None**.
 - ▶ **Correct!**

Base Case: $n = 1$

- ▶ Suppose $\text{arr}[\text{start}:\text{stop}]$ has one element.
- ▶ If that element is the target, the algorithm will find it.
 - ▶ **Correct!**
- ▶ If it isn't, the algorithm will recurse on a problem of size 0 and return **None**.
 - ▶ **Correct!**

Recursive Calls

- ▶ Recursive calls must be on **smaller problems**.
 - ▶ Otherwise, base case never reached. Infinite recursion!

```
import math
def binary_search(arr, t, start, stop):
    """
        Searches arr[start:stop] for t.
        Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

- ▶ Is $\text{arr}[\text{start}:\text{middle}]$ smaller than $\text{arr}[\text{start}:\text{stop}]$?
- ▶ Is $\text{arr}[\text{middle}+1:\text{stop}]$ smaller than $\text{arr}[\text{start}:\text{stop}]$?

Leap of Faith

- ▶ **Assume** the recursive calls work.
- ▶ Does the overall algorithm work, then?

```
import math
def binary_search(arr, t, start, stop):
    """
        Searches arr[start:stop] for t.
        Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Exercise

Does this code work? Why or why not?

```
import math
def summation(numbers):
    n = len(numbers)
    if n == 0:
        return 0
    middle = math.floor(n / 2)
    return (
        summation(numbers[:middle])
        +
        summation(numbers[middle:]))
)
```

Induction

- ▶ These steps can be turned into a formal proof by **induction**.
- ▶ For us, less necessary to prove to other people.
- ▶ Instead, prove to **yourself** that your code works.
- ▶ We won't be doing formal inductive proofs.

Why does this work?

- ▶ Show that it works for size 1 (base case).
- ▶ \Rightarrow will work for size 2 (inductive step).
- ▶ \Rightarrow will work for sizes 3, 4 (inductive step).
- ▶ \Rightarrow will work for sizes 5, 6, 7, 8 (inductive step).

DSC 40B

Theoretical Foundations II

Lecture 5 | Part 4

Recurrence Relations

Time Complexity of Binary Search

- ▶ What is the time complexity of `binary_search`?
- ▶ No loops!

Best Case

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Worst Case

Let $T(n)$ be worst case time on input of size n .

```
import math
def binary_search(arr, t, start, stop):
    """
        Searches arr[start:stop] for t.
        Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Recurrence Relations

- We found

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

- This is a **recurrence relation**.

Solving Recurrences

- ▶ We want simple, non-recursive formula for $T(n)$ so we can see how fast $T(n)$ grows.
 - ▶ Is it $\Theta(n)$? $\Theta(n^2)$? Something else?
- ▶ Obtaining a simple formula is called **solving** the recurrence.

Example: Getting Rich

- ▶ Suppose on day 1 of job, you are paid \$3.
- ▶ Each day thereafter, your pay is doubled.
- ▶ Let $S(n)$ be your pay on day n :

$$S(n) = \begin{cases} 2 \cdot S(n - 1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

Example: Unrolling

$$S(n) = \begin{cases} 2 \cdot S(n - 1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

- Take $n = 4$.

Solving Recurrences

We'll use a four-step process to solve recurrences:

1. “Unroll” several times to find a pattern.
2. Write general formula for k th unroll.
3. Solve for # of unrolls needed to reach base case.
4. Plug this number into general formula.

Step 1: Unroll several times

$$S(n) = \begin{cases} 2 \cdot S(n - 1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

Step 2: Find general formula

$$\begin{aligned}S(n) &= 2 \cdot S(n - 1) \\&= 2 \cdot 2 \cdot S(n - 2) \\&= 2 \cdot 2 \cdot 2 \cdot S(n - 3)\end{aligned}$$

On step k :

Step 3: Find step # of base case

- ▶ On step k , $S(n) = 2^k \cdot S(n - k)$.
- ▶ When do we see $S(1)$?

Step 4: Plug into general formula

- ▶ From step 2: $S(n) = 2^k \cdot S(n - k)$.
- ▶ From step 3: Base case of $S(1)$ reached when $k = n - 1$.
- ▶ So:

Solving the Recurrence

- ▶ We have **solved** the recurrence¹:

$$S(n) = 3 \cdot 2^{n-1}$$

- ▶ This is the **exact** solution. The **asymptotic** solution is $S(n) = \Theta(2^n)$.
- ▶ We'll call this method “solving by unrolling”.

¹On day 20, you'll be paid \approx 1.5 million dollars.

DSC 40B

Theoretical Foundations II

Lecture 5 | Part 5

Binary Search Recurrence

Binary Search

- ▶ What is the time complexity of `binary_search`?
- ▶ Best case: $\Theta(1)$.
- ▶ Worst case:

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

Simplification

- ▶ When solving, we can replace $\Theta(f(n))$ with $f(n)$:

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- ▶ As long as we state final answer using Θ notation!

Another Simplification

- ▶ When solving, we can assume n is a power of 2.

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

Step 2: Find general formula

$$\begin{aligned}T(n) &= T(n/2) + 1 \\&= T(n/4) + 2 \\&= T(n/8) + 3\end{aligned}$$

On step k :

Step 3: Find step # of base case

- ▶ On step k , $T(n) = T(n/2^k) + k$
- ▶ When do we see $T(1)$?

Step 4: Plug into general formula

- ▶ $T(n) = T(n/2^k) + k$
- ▶ Base case of $T(1)$ reached when $k = \log_2 n$.
- ▶ So:

Note

- ▶ Remember: $\log_b x = (\log_a x) / (\log_a b)$
- ▶ So we don't write $\Theta(\log_2 n)$
- ▶ Instead, just: $\Theta(\log n)$

Time Complexity of Binary Search

- ▶ Best case: $\Theta(1)$
- ▶ Worst case: $\Theta(\log n)$

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.
- ▶ Linear search: 317 years.

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.
- ▶ Linear search: 317 years.
- ▶ Binary search: ≈ 60 nanoseconds.

Exercise

Binary search seems so much faster than linear search. What's the caveat?

Caveat

- ▶ The array must be **sorted**.
- ▶ This takes $\Omega(n)$ time.

Why use binary search?

- ▶ If data is **not sorted**, sorting + binary search takes longer than linear search.
- ▶ But if doing **multiple queries**, looking for nearby elements, sort once and use binary search after.

Theoretical Lower Bounds

- ▶ A lower bound for searching a sorted list is $\Omega(\log n)$.
- ▶ This means that binary search has **optimal** worst case time complexity.

Databases

- ▶ Some database servers will **sort** by key, use binary search for queries.
- ▶ Often instead of sorting, **B-Tree indexes** are used.
- ▶ But sorting + binary search still used when space is limited.

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 1

Binary Search Recurrence

Binary Search

```
import math
def binary_search(arr, t, start, stop):
    """
        Searches arr[start:stop] for t.
        Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Binary Search

- ▶ What is the time complexity of `binary_search`?
- ▶ Best case: $\Theta(1)$.
- ▶ Worst case:

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

Simplification

- ▶ When solving, we can replace $\Theta(f(n))$ with $f(n)$:

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- ▶ As long as we state final answer using Θ notation!

Another Simplification

- ▶ When solving, we can assume n is a power of 2.

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

Step 2: Find general formula

$$\begin{aligned}T(n) &= T(n/2) + 1 \\&= T(n/4) + 2 \\&= T(n/8) + 3\end{aligned}$$

On step k :

Step 3: Find step # of base case

- ▶ On step k , $T(n) = T(n/2^k) + k$
- ▶ When do we see $T(1)$?

Step 4: Plug into general formula

- ▶ $T(n) = T(n/2^k) + k$
- ▶ Base case of $T(1)$ reached when $k = \log_2 n$.
- ▶ So:

Note

- ▶ Remember: $\log_b x = (\log_a x) / (\log_a b)$
- ▶ So we don't write $\Theta(\log_2 n)$
- ▶ Instead, just: $\Theta(\log n)$

Time Complexity of Binary Search

- ▶ Best case: $\Theta(1)$
- ▶ Worst case: $\Theta(\log n)$

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.
- ▶ Linear search: 317 years.

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.
- ▶ Linear search: 317 years.
- ▶ Binary search: ≈ 60 nanoseconds.

Exercise

Binary search seems so much faster than linear search. What's the caveat?

Caveat

- ▶ The array must be **sorted**.
- ▶ This takes $\Omega(n)$ time.

Why use binary search?

- ▶ If data is **not sorted**, sorting + binary search takes longer than linear search.
- ▶ But if doing **multiple queries**, looking for nearby elements, sort once and use binary search after.

Theoretical Lower Bounds

- ▶ A lower bound for searching a sorted list is $\Omega(\log n)$.
- ▶ This means that binary search has **optimal** worst case time complexity.

Databases

- ▶ Some database servers will **sort** by key, use binary search for queries.
- ▶ Often instead of sorting, **B-Tree indexes** are used.
- ▶ But sorting + binary search still used when space is limited.

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 2

Selection Sort and Loop Invariants

Sorting

- ▶ Sorting is a very common operation.
- ▶ But why is it important?

Sorting

- ▶ Sorting is a very common operation.
- ▶ But why is it important?
- ▶ Aesthetic reasons?

Sorting

- ▶ Sorting is a very common operation.
- ▶ But why is it important?
- ▶ Aesthetic reasons?
- ▶ Sorting makes some problems easier to solve.

Today

- ▶ How do we sort?
- ▶ How fast can we sort?
- ▶ How do we use sorted structure to write faster algorithms?

Today

- ▶ **Also:** how to understand complex loops with **loop invariants**.

Selection Sort

- ▶ Repeatedly remove smallest element.
- ▶ Put it at beginning of new list.

Example: arr = [5, 6, 3, 2, 1]

In-place Selection Sort

- ▶ We don't need a separate list.
 - ▶ We can swap elements until sorted.
- ▶ Store “new” list at the beginning of input list.
- ▶ Separate the old and new with a **barrier**.

Example: arr = [5, 6, 3, 2, 1]

```
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(n-1):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
            arr[min_ix], arr[barrier_ix]
        )
```

```
def find_minimum(arr, start):
    """Finds index of minimum. Assumes non-empty."""
    n = len(arr)
    min_value = arr[start]
    min_ix = start
    for i in range(start + 1, n):
        if arr[i] < min_value:
            min_value = arr[i]
            min_ix = i
    return min_ix
```

Loop Invariants

- ▶ How we understand an iterative algorithm?
- ▶ A **loop invariant** is a statement that is true after every iteration.
 - ▶ And before the loop begins!

Loop Invariant(s)

After the α th iteration of selection sort, each of the first α elements is \leq each of the remaining elements.

Example: arr = [5, 6, 3, 2, 1]

Loop Invariant(s)

After the α th iteration, the first α elements are sorted.

Example: arr = [5, 6, 3, 2, 1]

Loop Invariants

- ▶ Plug the total number of iterations into the loop invariant to learn about the result.
 - ▶ `selection_sort` makes $n - 1$ iterations:
 - ▶ After the $(n - 1)$ th iteration, the first $(n - 1)$ elements are sorted.
 - ▶ After the $(n - 1)$ th iteration, each of the first $(n - 1)$ elements is \leq each of the remaining elements.

Time Complexity

```
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(n-1):
        # find index of min in arr[barrier_ix:]
        min_value = arr[barrier_ix]
        min_ix = barrier_ix
        for i in range(barrier_ix + 1, n):
            if arr[i] < min_value:
                min_value = arr[i]
                min_ix = i
        #swap
        arr[barrier_ix], arr[min_ix] = (
            arr[min_ix], arr[barrier_ix]
        )
```

Time Complexity

- ▶ Selection sort takes $\Theta(n^2)$ time.

Exercise

Modify `selection_sort` so that it computes a **median** of the input array. What is the time complexity?

```
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(n-1):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
            arr[min_ix], arr[barrier_ix]
        )
```

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 3

Mergesort

Can we sort faster?

- ▶ The tight theoretical lower bound for **comparison** sorting is $\Theta(n \log n)$.
- ▶ Selection sort is quadratic.
- ▶ How do we sort in $\Theta(n \log n)$ time?

Mergesort

- ▶ Mergesort is a fast sorting algorithm.
- ▶ Has **best possible** (worst-case) time complexity:
 $\Theta(n \log n)$.
- ▶ Implements **divide/conquer/recombine** strategy.

The Idea

- ▶ **Divide:** split the array into halves
 - ▶ $[6, 1, 9, 2, 4, 3] \rightarrow [6, 1, 9], [2, 4, 3]$
- ▶ **Conquer:** sort each half, recursively
 - ▶ $[6, 1, 9] \rightarrow [1, 6, 9]$ and $[2, 4, 3] \rightarrow [2, 3, 4]$
- ▶ **Combine:** merge sorted halves together
 - ▶ $[1, 6, 9], [2, 3, 4] \rightarrow [1, 2, 3, 4, 6, 9]$

Aside: splitting arrays

- ▶ Splitting an array in half by **slicing**:

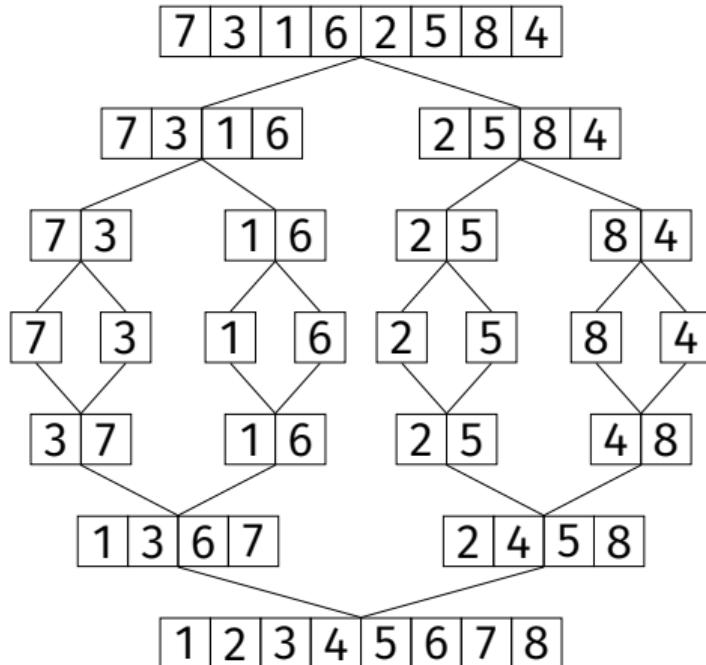
```
»> arr = [9, 1, 4, 2, 5]
»> middle = math.floor(len(arr) / 2)
»> arr[:middle]
[9, 1]
»> arr[middle:]
[4, 2, 5]
```

- ▶ **Warning!** Creates a copy!

Mergesort

```
def mergesort(arr):
    """Sort array in-place."""
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

The Idea



Understanding Mergesort

1. What is the base case?
2. Are the recursive problems smaller?
3. Assuming the recursive calls work, does the whole algorithm work?

1. Base Case: $n = 1$

- ▶ Arrays of size one are trivially sorted.
- ▶ Returns immediately. **Correct!**

2. Smaller Problems?

- ▶ Are `arr[:middle]` and `arr[middle:]` always smaller than `arr`?
- ▶ Try it for `len(arr) == 2`.

3. Does it Work?

- ▶ Assume mergesort works on arrays of size $< n$.
- ▶ Does it work on arrays of size n ?

Mergesort

```
def mergesort(arr):
    """Sort array in-place."""
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 4

Merge

Merging

- ▶ We have sorted each half.
- ▶ Now we need to **merge** together.

Merging

- ▶ We have sorted each half.
- ▶ Now we need to **merge** together.
- ▶ **Note:** this is an example of a problem that is made easier by sorting.

merge

3

1

merge

3

2

1

merge

3

6

1 2

merge

5

6

1 2 3

merge

7

6

1

2

3

5

merge

7

1 2 3 5 6

merge

8

1 2 3 5 6 7

merge

1 2 3 5 6 8

merge

```
def merge(left, right, out):
    """Merge sorted arrays, store in out."""
    left.append(float('inf'))
    right.append(float('inf'))
    left_ix = 0
    right_ix = 0

    for ix in range(len(out)):
        if left[left_ix] < right[right_ix]:
            out[ix] = left[left_ix]
            left_ix += 1
        else:
            out[ix] = right[right_ix]
            right_ix += 1
```

Loop Invariant

- ▶ Assume left and right are sorted.
- ▶ **Loop invariant:** After α th iteration,
`out[: α] == sorted(left + right)[: α]`

Key of mergesort

- ▶ merge is where the **actual sorting** happens.
- ▶ Example: `merge([3], [1], ...)` results in `[1, 3]`

Time Complexity of merge

```
def merge(left, right, out):
    """Merge sorted arrays, store in out."""
    left.append(float('inf'))
    right.append(float('inf'))
    left_ix = 0
    right_ix = 0

    for ix in range(len(out)):
        if left[left_ix] < right[right_ix]:
            out[ix] = left[left_ix]
            left_ix += 1
        else:
            out[ix] = right[right_ix]
            right_ix += 1
```

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 5

Time Complexity of Mergesort

Time Complexity

```
def mergesort(arr):
    """Sort array in-place."""
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

Aside: Copying

- ▶ What is `arr[:middle]` doing “under the hood”?
- ▶ What is the time complexity?

The Recurrence

```
def mergesort(arr):
    """Sort array in-place."""
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

Solving the Recurrence

$$T(n) = 2T(n/2) + \Theta(n)$$

Optimality

- ▶ **Theorem:** Any (comparison) sorting algorithm's worst-case time complexity must be $\Omega(n \log n)$.
- ▶ Mergesort is **optimal!**

Be Careful!

- ▶ It is possible for a sorting algorithm to have a **best case** time complexity smaller than $n \log n$.
 - ▶ Insertion sort, for example.
- ▶ Mergesort has best case time complexity of $\Theta(n \log n)$.
- ▶ Mergesort is **sub-optimal** in this sense!

Be Careful!

- ▶ The $\Theta(n \log n)$ lower-bound is for **comparison sorting**.
- ▶ It is possible to sort in worst-case $\Theta(n)$ time without comparing.¹

¹Bucket sort, radix sort, etc.

What if?

- ▶ **Divide:** split the array into halves
- ▶ **Conquer:** sort each half **using selection sort**
- ▶ **Combine:** merge sorted halves together

mergeselectionsort

```
def mergeselectionsort(arr):
    """Sort array in-place."""
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        selection_sort(left)
        selection_sort(right)
        merge(left, right, arr)
```

Exercise

What is the time complexity of this algorithm?

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 6

Using Sorted Structure

Sorted structure is useful

- ▶ Some problems become **much easier** if input is sorted.
 - ▶ For example, median, minimum, maximum.
- ▶ Sorting is useful as a **preprocessing** step.

Recall: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ You want the total time of the two movies to be **as close as possible** to D .

The Movie Problem

- ▶ Brute force algorithm: $\Theta(n^2)$
- ▶ We can do better, if movie times are **sorted**.

Example

- ▶ Flight duration $D = 155$
- ▶ Movie times: 60, 80, 90, 120, 130

| | 60 | 80 | 90 | 120 | 130 |
|-----|----|----|----|-----|-----|
| 60 | | | | | |
| 80 | | | | | |
| 90 | | | | | |
| 120 | | | | | |
| 130 | | | | | |

Best pair:

The Algorithm

- ▶ Keep index of shortest and longest remaining.
- ▶ On every iteration, pair the shortest and longest.
- ▶ If this pair is too long, remove longest movie; otherwise remove shortest.
 - ▶ If times are **sorted**, finding new longest/shortest movie takes $\Theta(1)$ time!

60, 80, 90, 120, 130

The Algorithm

```
def optimize_entertainment(times, target):
    """assume times is sorted."""
    shortest = 0
    longest = len(times) - 1

    best_pair = (shortest, longest)
    best_objective = None

    for i in range(len(times) - 1):
        total_time = times[shortest] + times[longest]

        if abs(total_time - target) < best_objective:
            best_objective = abs(total_time - target)
            best_pair = (shortest, longest)

        if total_time == target:
            return (shortest, longest)
        elif total_time < target:
            shortest += 1
        else: # total_time > target
            longest -= 1

    return best_pair
```

Main Idea

Sorted structure allows you to rule out possibilities without explicitly checking them. But, it requires you to spend the time sorting first.

Tip: when designing an algorithm, think about sorting the input first.

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 1

The Median and Order Statistics

The Median

- ▶ How fast can we find a **median** of n numbers?

Algorithms

- ▶ We have seen several ways of computing a median:
 - ▶ Alg. 1: Minimize absolute error, brute force.
 - ▶ Alg. 2: Use definition (half \leq , half \geq).
 - ▶ ...

Exercise

Using what we know so far, what approach for finding the median has the best **worst-case time complexity?**

Best so far...

- ▶ Sort the list with mergesort, return middle element.
- ▶ Time complexity: $\Theta(n \log n)$.

Is sorting necessary?

- ▶ Need to sort the whole list just to find middle?
- ▶ Seems like more work than necessary.

Today

- ▶ We'll design an algorithm which runs in $\Theta(n)$ expected time.
- ▶ Much more useful than just finding median...

Order Statistics

- ▶ The median is an example of an **order statistic**.

Definition

Given n numbers, the **k th order statistic** is the k th smallest number in the collection.

Example

[99, 42, -77, -12, 101]

- ▶ 1st order statistic:
- ▶ 2nd order statistic:
- ▶ 4th order statistic:

Exercise

Some special cases of order statistics go by different names. Can you think of some?

Special Cases

- ▶ **Minimum:** 1st order statistic.
- ▶ **Maximum:** n th order statistic.
- ▶ **Median:** $[n/2]$ th order statistic¹.
- ▶ **p th Percentile:** $\lceil \frac{p}{100} \cdot n \rceil$ th order statistic.

¹What if n is even?

Goal

- ▶ **Fast** algorithm for computing any order statistic.
- ▶ Interestingly, some seem easier than others.
- ▶ Our algorithm will find **any** order statistic in $\Theta(n)$ *expected* time.

Approach #1

- ▶ We can modify `selection_sort` to find the k th order statistic.
- ▶ Loop invariant: after k th iteration, first k elements are in final sorted order.

```
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(n-1):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
            arr[min_ix], arr[barrier_ix]
        )
```

```
def select_k(arr, k):
    """Find kth order statistic."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(k):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        #swap
        arr[barrier_ix], arr[min_ix] = (
            arr[min_ix], arr[barrier_ix]
        )
    return arr[k-1]
```

Exercise

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

Approach #1

- ▶ 1st order statistic: $\Theta(n)$.
- ▶ n th order statistic: $\Theta(n^2)$.
- ▶ Median: $\Theta(n^2)$.
- ▶ k th order statistic: $\Theta(kn)$.

Exercise

Describe how to find any order statistic in $\Theta(n \log n)$ time.

Approach #2

- ▶ Sort with mergesort, return $\text{arr}[k-1]$
- ▶ $\Theta(n \log n)$ time. Could be better...

DSC 40B

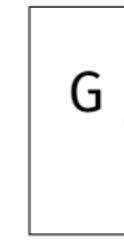
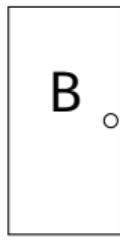
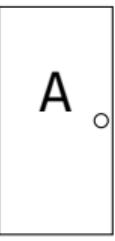
Theoretical Foundations II

Lecture 7 | Part 2

Quickselect

The Goal

- ▶ Given a collection of n numbers and an order, k .
- ▶ Find the k th smallest number in the collection.



22

101

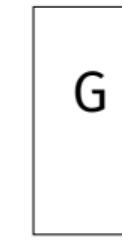
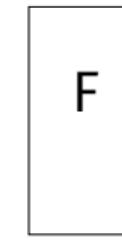
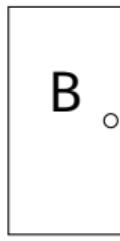
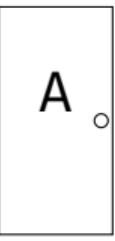
42

19

14

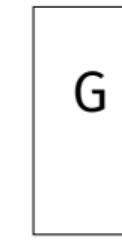
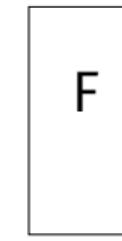
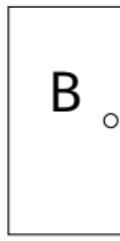
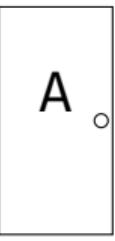
84

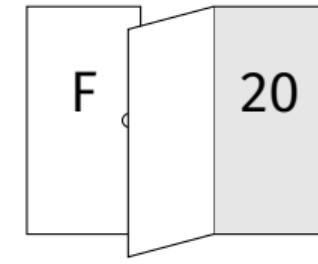
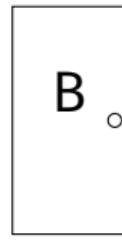
20

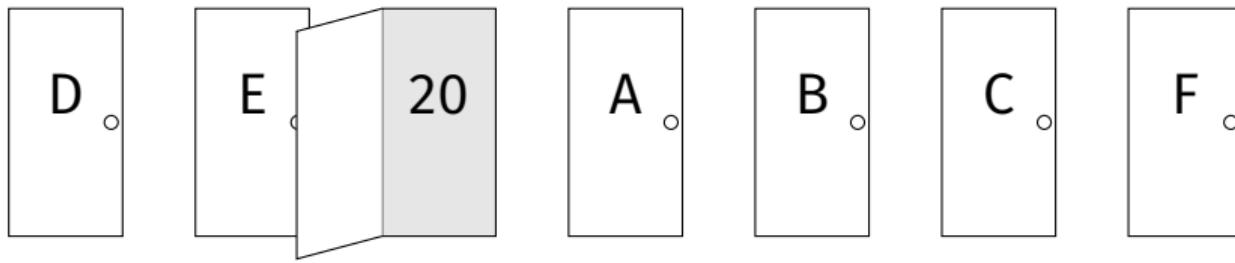


Game Show

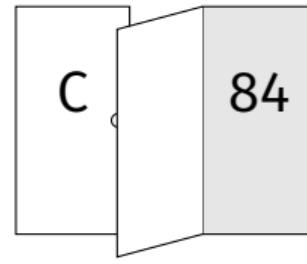
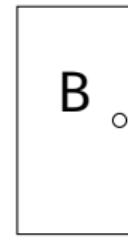
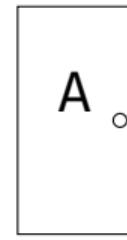
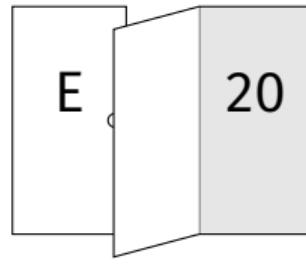
- ▶ **Goal:** tell the host the **largest** number.
- ▶ **Caution:** with every door opened, your money is reduced.
- ▶ **Twist:** After opening a door, the host tells you:
 - ▶ which doors are smaller.
 - ▶ which doors are larger.
 - ▶ they **partition** the doors into higher and lower by moving them.

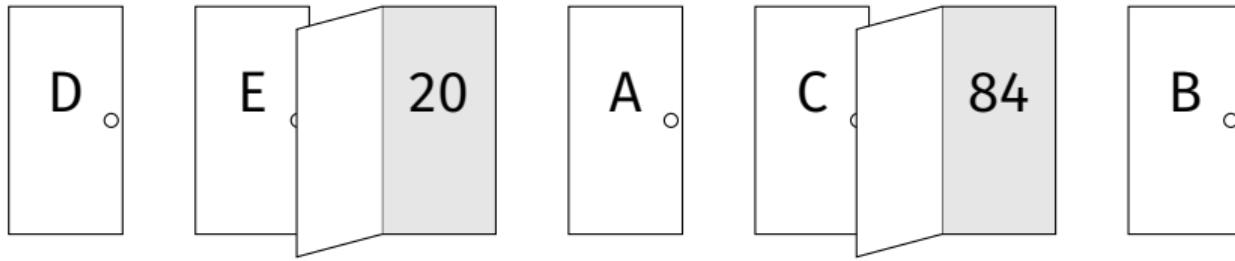




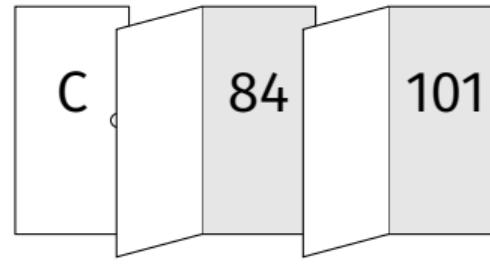
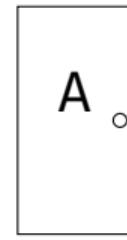
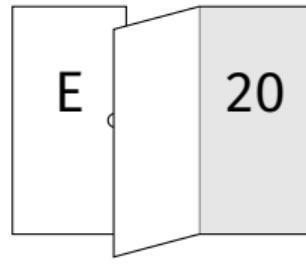


after partitioning





after partitioning



Main Idea

After partitioning, the just-opened door is in the **correct place** in the sorted order (but the other doors may not be).

But, every door to the left is smaller (\leq), every door to the right is larger (\geq).

In general...

- ▶ Let's generalize strategy for k th order statistic.
- ▶ Example: $k = 2$.

A .

B .

C .

D .

E .

F .

G .

A

B

C

D

E

F

20

D .

E . 20

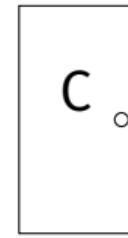
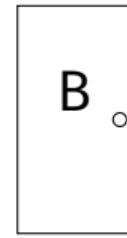
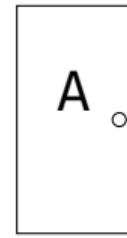
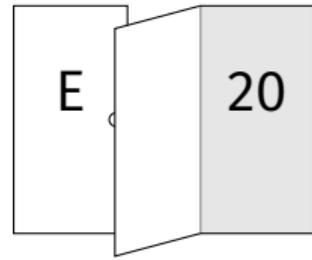
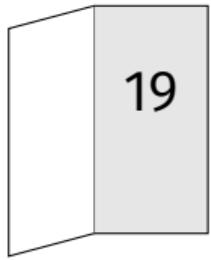
A .

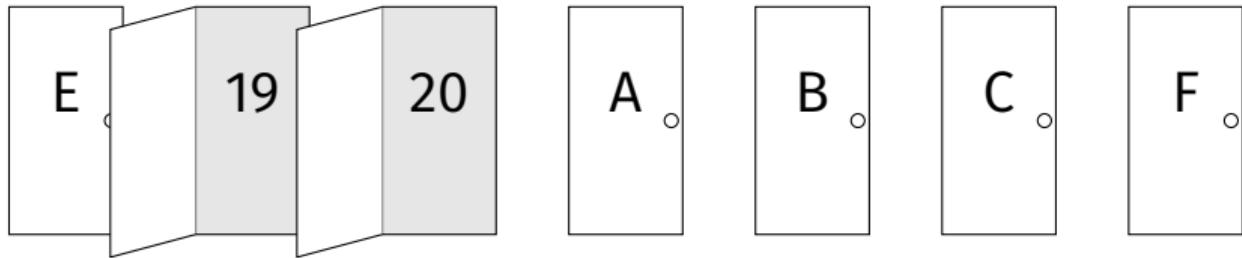
B .

C .

F .

after partitioning





after partitioning

Strategy

- ▶ Open arbitrary door (that hasn't been ruled out).
- ▶ **Partition** the doors around this number:
 - ▶ Move doors smaller than this to the left,
 - ▶ Larger than this to the right.
- ▶ Let p be our door's new position, k be the order we want.
 - ▶ If $p = k$, return this door.
 - ▶ If $p < k$, rule out doors to left.
 - ▶ If $p > k$, rule out doors to right.
- ▶ Repeat.

In Code

```
import random

def quickselect(arr, k, start, stop):
    """Finds kth order statistic in numbers[start:stop]"""
    pivot_ix = random.randrange(start, stop)
    pivot_ix = partition(arr, start, stop, pivot_ix)
    pivot_order = pivot_ix + 1
    if pivot_order == k:
        return arr[pivot_ix]
    elif pivot_order < k:
        return quickselect(arr, k, pivot_ix + 1, stop)
    else:
        return quickselect(arr, k, start, pivot_ix)
```

Example

```
arr = [77, 42, 11, 99, 0, 101]      k = 3
```

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 3

Partition

Partitioning

- ▶ Given an array of n numbers and the index of a **pivot** p .
- ▶ Rearrange elements so that:
 - ▶ Everything $< p$ is first.
 - ▶ Everything $= p$ is next.
 - ▶ Everything $> p$ is last.
- ▶ Return index of first element $\geq p$.

```
def partition(arr, start, stop, pivot_ix):
    """Partition arr[start:stop] around pivot."""
    left = []
    pivot_count = 0
    right = []
    pivot = arr[pivot_ix]
    for ix in range(start, stop):
        if arr[ix] < pivot:
            left.append(arr[ix])
        elif arr[ix] == pivot:
            pivot_count += 1
        else:
            right.append(arr[ix])
    ix = start
    for x in left:
        arr[ix] = x
        ix += 1
    for i in range(pivot_count):
        arr[ix] = pivot
        ix += 1
    for x in right:
        arr[ix] = x
        ix += 1
    return start + len(left)
```

Partition

- ▶ partition takes $\Theta(n)$ time.
 - ▶ This is **optimal**.
- ▶ But we can use memory more efficiently.

Motivation

- ▶ Similar to selection sort, we'll use **two** barriers:
- ▶ “Middle” barrier:
 - ▶ Separates things $<$ pivot from things \geq
 - ▶ Points to first thing in “right”
- ▶ “End” barrier:
 - ▶ Separates processed from unprocessed.
 - ▶ Points to first unprocessed thing.

Example

Simplification: start by moving pivot to end.

```
arr = [77, 42, 11, 99, 0, 101]      pivot = 1
```

```
def in_place_partition(arr, start, stop, pivot_ix):
    def swap(ix_1, ix_2):
        arr[ix_1], arr[ix_2] = arr[ix_2], arr[ix_1]

    pivot = arr[pivot_ix]
    swap(pivot_ix, stop-1)
    middle_barrier = start
    for end_barrier in range(start, stop - 1):
        if arr[end_barrier] < pivot:
            swap(middle_barrier, end_barrier)
            middle_barrier += 1
    # else:
    #     # do nothing
    swap(middle_barrier, stop-1)
    return middle_barrier
```

Efficiency

- ▶ Also takes $\Theta(n)$ time.
- ▶ No auxiliary memory required.

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 4

Time Complexity Analysis

Time Complexity

- ▶ What is time complexity of quickselect?

```
import random
def quickselect(arr, k, start, stop):
    """Finds kth order statistic in numbers[start:stop]"""
    pivot_ix = random.randrange(start, stop)
    pivot_ix = partition(arr, start, stop, pivot_ix)
    pivot_order = pivot_ix + 1
    if pivot_order == k:
        return arr[pivot_ix]
    elif pivot_order < k:
        return quickselect(arr, k, pivot_ix + 1, stop)
    else:
```

Problem

- ▶ We don't know the size of the subproblem.
 - ▶ Is random, can be anywhere from 1 to $n - 1$.
- ▶ Difficult to write recurrence relation.

Good and Bad Pivots

- ▶ Some pivots are better than others.
 - ▶ **Good**: splits array into roughly balanced halves.
 - ▶ **Bad**: splits array into wildly unbalanced pieces.

Exercise

Suppose we're searching for the minimum. What would be the worst possible pivot?

Worst Case

- ▶ Suppose we're searching for $k = 1$ (minimum).
- ▶ Worst pivot: the maximum.
- ▶ Worst case: use max as pivot every time.
- ▶ Subproblem size: $n - 1$.

Worst Case

- ▶ Every recursive call is on problem of size $n - 1$.
- ▶ $T(n) = T(n - 1) + \Theta(n)$.
 - ▶ Solution: $\Theta(n^2)$.
- ▶ Intuitively, randomly choosing largest number as pivot every time is **very** unlikely!

$$\frac{1}{n} \times \frac{1}{n-1} \times \frac{1}{n-2} \times \cdots \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{n!}$$

Equally Unlikely

- ▶ Pivot falls exactly in the middle, every time.
- ▶ Subproblems are of size $n/2$.
- ▶ $T(n) = T(n/2) + \Theta(n)$.
 - ▶ Solution: $\Theta(n)$.

Typically

- ▶ Pivot falls somewhere in the middle.
- ▶ Sometimes **good**, sometimes **bad**.
- ▶ But **good** pivots reduce problem size by **so much** that they make up for **bad** pivots.

Analogy

- ▶ You're 100 miles away from home.
- ▶ You have a button that, if you press it, teleports you **1 mile** closer to home.
- ▶ How many times must you press it before you're 1 mile away from home?

Analogy

- ▶ You're 100 miles away from home.
- ▶ You have a button that, if you press it, teleports you **half the distance** to home.
- ▶ How many times must you press it before you're 1 mile away from home?

Analogy

- ▶ You're 100 miles away from home.
- ▶ You have a button that, if you press it, teleports you **half the distance** to home with probability $1/2$, does nothing with probability $1/2$.
- ▶ How many times must you press it before you're 1 mile away from home?

Analysis

- ▶ We'll call a pivot **good** if it falls in $[\frac{n}{4}, \frac{3n}{4}]$.
 - ▶ Probability: 1/2
 - ▶ Max problem size: $3n/4$.
- ▶ We'll call a pivot **bad** if it falls outside $[\frac{n}{4}, \frac{3n}{4}]$.
 - ▶ Probability: 1/2
 - ▶ Max problem size: $n - 1$.

Observation

$T(n)$ = time to get from n to base case

Observation

$T(n) = \text{time to get from } n \text{ to } \frac{3}{4}n$

+ time to get from $\frac{3}{4}n$ to $\left(\frac{3}{4}\right)^2 n$

+ time to get from $\left(\frac{3}{4}\right)^2 n$ to $\left(\frac{3}{4}\right)^3 n$

+ ...

Observation

Expected $T(n)$ = expected time to get from n to $\frac{3}{4}n$

- + expected time to get from $\frac{3}{4}n$ to $\left(\frac{3}{4}\right)^2 n$
- + expected time to get from $\left(\frac{3}{4}\right)^2 n$ to $\left(\frac{3}{4}\right)^3 n$
- + ...

Related

- ▶ What is the expected number of coin flips necessary in order to see “heads”?

Related

- ▶ What is the expected number of coin flips necessary in order to see “heads”?
- ▶ Answer: 2

Implication

- ▶ Expected number of calls necessary to go from n to $3n/4$ is two.
- ▶ First call does cn work, second does $c \times (3/4)n$, third does $c \times (3/4)^2 n$, ...

Observation

Expected $T(n)$ = expected time to get from n to $\frac{3}{4}n$

- + expected time to get from $\frac{3}{4}n$ to $\left(\frac{3}{4}\right)^2 n$
- + expected time to get from $\left(\frac{3}{4}\right)^2 n$ to $\left(\frac{3}{4}\right)^3 n$
- + ...

Total Expected Time

$$2cn + 2\left(\frac{3}{4}\right)cn + 2\left(\frac{3}{4}\right)^2 cn + \dots = 2cn \cdot \sum_{p=0}^{\infty} \left(\frac{3}{4}\right)^p$$

Quickselect

- ▶ Expected time complexity: $\Theta(n)$.
- ▶ Worst case: $\Theta(n^2)$, but **very unlikely**.

Median

- We can find the median in expected linear time with **quickselect**.

DSC 40B

Theoretical Foundations II

Lecture 7 | Part 5

Quicksort

Last Time

- ▶ We saw mergesort.
- ▶ **Divide:** split list directly down the middle
- ▶ **Conquer:** sort each half
- ▶ **Combine:** merge sorted halves together

merge does all the work

- ▶ In mergesort, we are lazy when we divide.
- ▶ So we have to work to combine.

$[4, 1, 3, 2] \rightarrow [4, 1], [3, 2] \rightarrow [4, 3], [2, 3] \rightarrow [1, 2, 3, 4]$

What if?

- ▶ Suppose we divide so that everything in left is smaller than everything in right:
- ▶ After sorting, no need for merge.
- ▶ $[5, 1, 3, 8, 6, 2] \rightarrow [1, 3, 2], [5, 8, 6]$

What if?

- ▶ Suppose we divide so that everything in left is smaller than everything in right:
- ▶ After sorting, no need for merge.
- ▶ $[5, 1, 3, 8, 6, 2] \rightarrow [1, 3, 2], [5, 8, 6]$
- ▶ This is what partition does!

Quicksort

```
def quicksort(arr, start, stop):
    """Sort arr[start:stop] in-place."""
    if stop - start > 1:
        pivot_ix = random.randrange(start, stop)
        pivot_ix = partition(arr, start, stop, pivot_ix)
        quicksort(arr, start, pivot_ix)
        quicksort(arr, pivot_ix+1, stop)
```

Time Complexity

- ▶ Average case: $\Theta(n \log n)$
- ▶ Worst case: $\Theta(n^2)$.
- ▶ Like with quickselect, worst case is **very rare**.

Mergesort vs Quicksort

- ▶ Mergesort has better worst case complexity.
- ▶ But in practice, Quicksort is often faster.
- ▶ Takes less memory, too.

Memory Requirements

- ▶ merges requires output array, $\Theta(n)$ additional space.
- ▶ partition works in-place, requires no additional space²
- ▶ Example: sorting 3 GB of data with 4 GB of RAM.

²Call stack for quicksort requires $\Theta(\log n)$ additional space.

DSC 40B

Theoretical Foundations II

Lecture 8 | Part 1

Dynamic Sets

Bookkeeping

- ▶ How do you store your books?

Bookkeeping

- ▶ How do you store your books?



Bookkeeping

- ▶ How do you store your books?



<https://bookriot.com/how-to-organize-bookshelves/>

Bookkeeping: Tradeoffs

- ▶ Messy:
 - ▶ No upfront cost.
 - ▶ Cost to search is high.
- ▶ Organized
 - ▶ Big upfront cost.
 - ▶ Cost to search is low.
- ▶ “Right” choice depends on how often we search.

Data Structures and Algorithms

- ▶ **Data structures** are ways of organizing data to make certain operations faster.
- ▶ Come with an upfront cost (preprocessing).
- ▶ “Right” choice of data structure depends on what operations we’ll be doing in the future.

Queries: Easy to Hard

- ▶ We've been thinking about **queries**.
 - ▶ Given a collection of data, is x in the collection?
- ▶ Querying is a fundamental operation.
 - ▶ Useful in a data science sense.
 - ▶ But also frequently performed in algorithms.
- ▶ There are several situations to think about.

Situation #1: Static Set, One Query

- ▶ **Given:** an unsorted collection of n numbers (or strings, etc.).
- ▶ In future, you will be asked single query.
- ▶ Which is better: linear search or sort + binary search?

Situation #1: Static Set, One Query

- ▶ **Given:** an unsorted collection of n numbers (or strings, etc.).
- ▶ In future, you will be asked single query.
- ▶ Which is better: linear search or sort + binary search?
 - ▶ Linear search: $\Theta(n)$ worst case.
 - ▶ Binary search would require sorting first in $\Theta(n \log n)$ worst case

Situation #2: Static Set, Many Queries

- ▶ **Given:** an unsorted collection of n numbers (or strings, etc.).
- ▶ In future, you will be asked **many** queries.
- ▶ Which is better: linear search or sort + binary search?
 - ▶ Depends on number of queries!

Exercise

Suppose you have a static set of n items. How long will it take^a to perform k queries in total with:

1. linear search?
2. sort + binary search?

If $k = n/10$, which should you use?

What if $k = \log n$?

^aOn average. Assume the best case is rare.

Situation #3: Dynamic Set, Many Queries

- ▶ **Given:** a collection of n numbers (or strings, etc.).
- ▶ In future, you will be asked **many** queries *and* to **insert** new elements.
- ▶ Best approach: ?

Binary Search?

- ▶ Can we still use binary search?
- ▶ **Problem:** To use binary search, we must maintain array in sorted order as we insert new elements.
- ▶ Inserting into array takes $\Theta(n)$ time in worst case.
 - ▶ Must “make room” for new element.
 - ▶ Can we use linked list with binary search?

Exercise

Suppose we have a collection of n elements. We make $n/4$ insertions and $n/4$ queries. How long will this take in total with

1. append to linked list append + linear search?
2. maintain sorted array + binary search?

Today

- ▶ Introduce (or review) **binary search trees**.
- ▶ BSTs support fast queries *and* insertions.
- ▶ Preserve sorted order of data after insertion.
- ▶ Can be modified to solve many problems efficiently.
 - ▶ Example: finding order statistics.

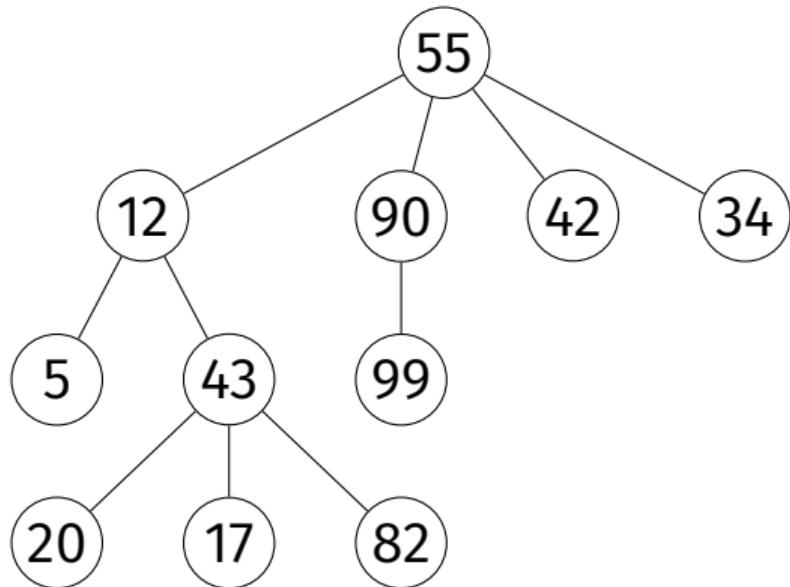
DSC 40B

Theoretical Foundations II

Lecture 8 | Part 2

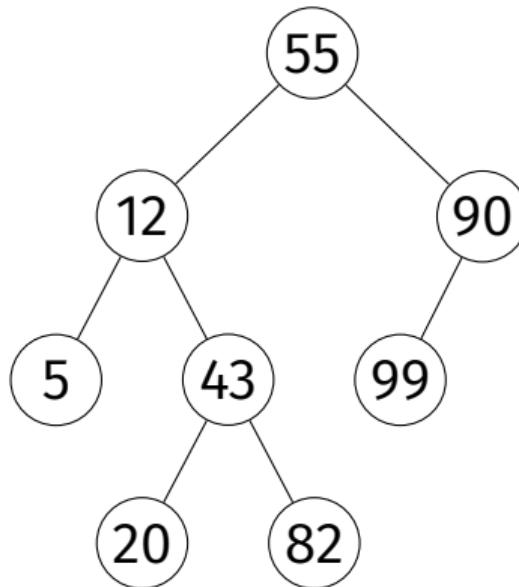
Binary Search Trees

Trees



Binary Trees

- ▶ Each node has *at most* two children (left and right).



Binary Search Tree

- ▶ A **binary search tree** (BST) is a binary tree that satisfies the following for any node x :
 - ▶ if y is in x 's **left** subtree:
$$y.\text{key} \leq x.\text{key}$$
 - ▶ if y is in x 's **right** subtree:
$$y.\text{key} \geq x.\text{key}$$

Assumption (for simplicity)

- ▶ We'll assume keys are unique (no duplicates).
- ▶ if y is in x 's **left** subtree:

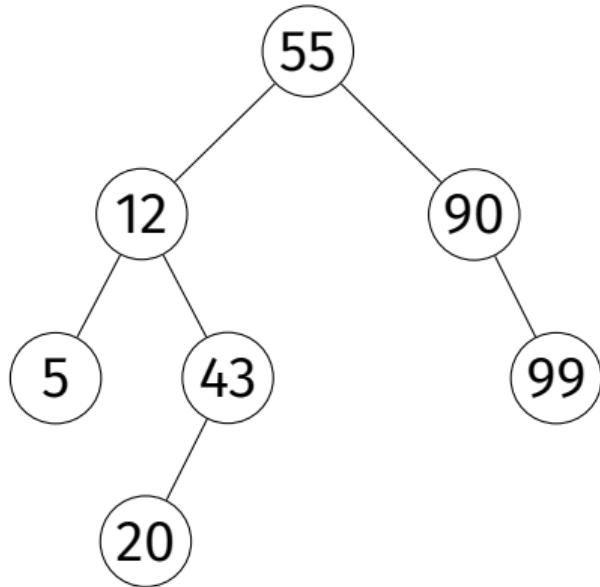
$$y.\text{key} < x.\text{key}$$

- ▶ if y is in x 's **right** subtree:

$$y.\text{key} > x.\text{key}$$

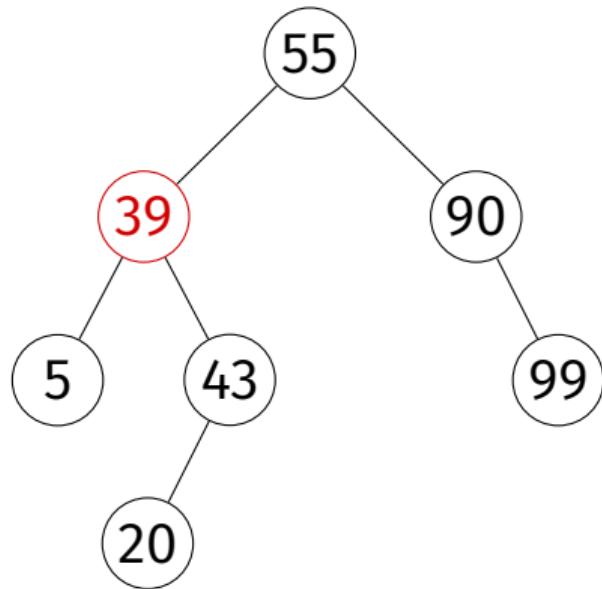
Example

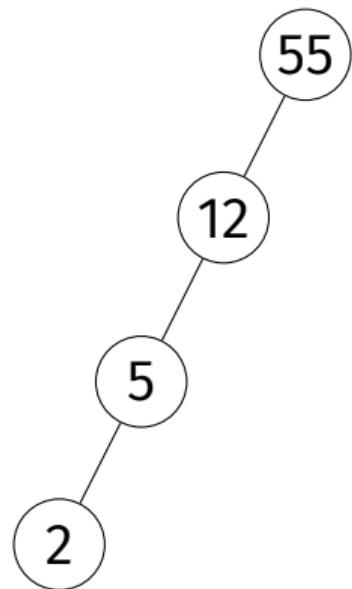
- ▶ This **is** a BST.



Example

- ▶ This is **not** a BST.





Exercise

Is this is a BST?

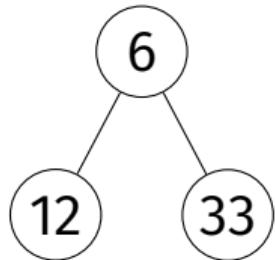
Height

- ▶ The **height** of a tree is the number of edges from the root to any leaf.
- ▶ Suppose a binary tree has n nodes.
- ▶ The **tallest** it can be is $\approx n$
- ▶ The **shortest** it can be is $\approx \log_2 n$

In Python

```
class Node:  
    def __init__(self, key, parent=None):  
        self.key = key  
        self.parent = parent  
        self.left = None  
        self.right = None  
  
class BinarySearchTree:  
    def __init__(self, root: Node):  
        self.root = root
```

In Python



```
root = Node(6)
n1 = Node(12, parent=root)
root.left = n1
n2 = Node(33, parent=root)
root.right = n2
tree = BinarySearchTree(root)
```

DSC 40B

Theoretical Foundations II

Lecture 8 | Part 3

Queries and Insertions in BSTs

Why?

- ▶ BSTs impose structure on data.
- ▶ “Not quite sorted”.
- ▶ Preprocessing for making insertions *and* queries faster.

Operations on BSTs

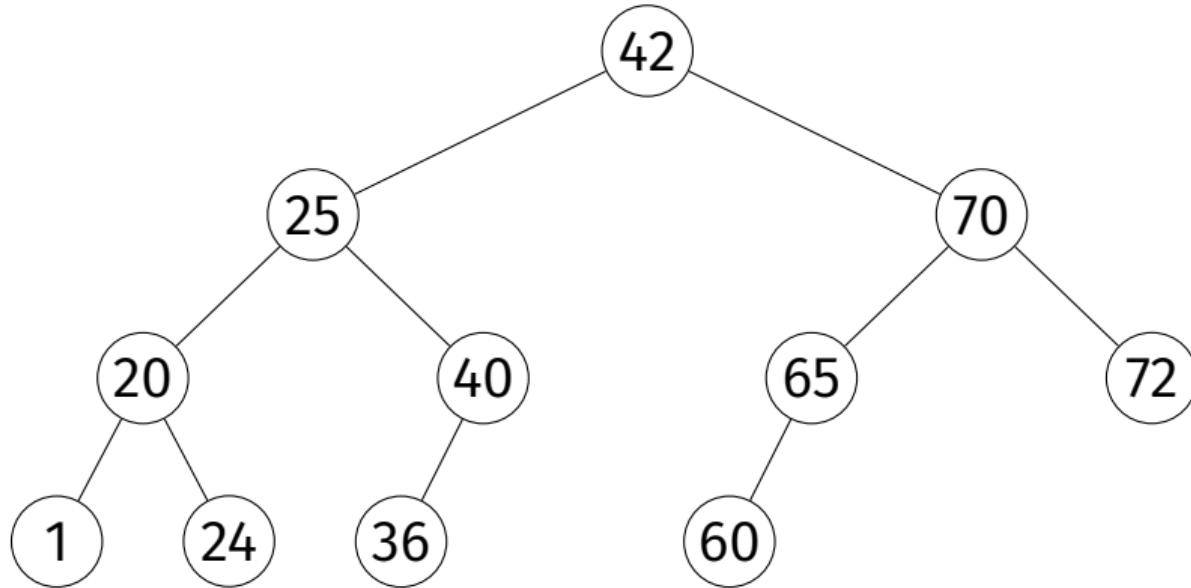
- ▶ We will want to:
 - ▶ **query** a key (is it in the tree?)
 - ▶ **insert** a new key

Queries

- ▶ **Given:** a BST and a target, t .
- ▶ **Return:** **True** or **False**, is the target in the collection?

Queries

- ▶ Is 36 in the tree? 65? 23?



Queries

- ▶ Start walking from root.
- ▶ If current node is:
 - ▶ equal to target, return **True**;
 - ▶ too large ($>$ target), follow left edge;
 - ▶ too small ($<$ target), follow right edge;
 - ▶ **None**, return **False**

Queries, in Python

```
def query(self, target):
    """As method of BinarySearchTree."""
    current_node = self.root
    while current_node is not None:
        if current_node.key == target:
            return current_node
        elif current_node.key < target:
            current_node = current_node.right
        else:
            current_node = current_node.left
    return None
```

Exercise

Complete the recursive version of query.

```
def query_recursive(node, target):
    """As a 'free function'."""
    if node is None:
        return False

    if node.key == target:
        ...
    elif ...:
        ...
    else:
        ...
```

Queries (Recursive)

```
def query_recursive(node, target):
    """As a 'free function'."""
    if node is None:
        return False

    if node.key == target:
        return node
    elif node.key < target:
        return query_recursive(node.right, target)
    else:
        return query_recursive(node.left, target)
```

Queries, Analyzed

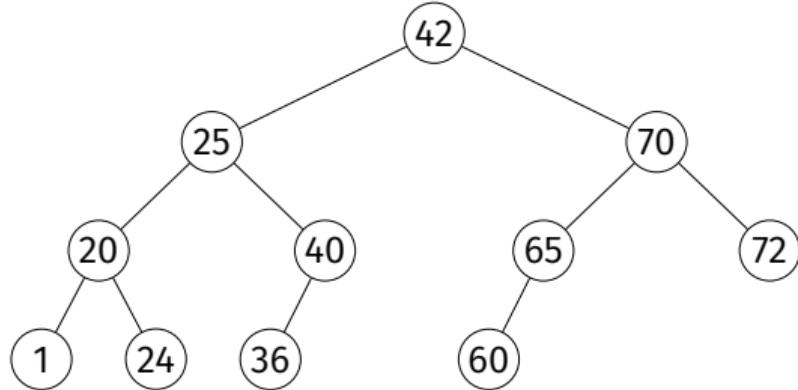
- ▶ Best case: $\Theta(1)$.
- ▶ Worst case: $\Theta(h)$, where h is **height** of tree.

Insertion

- ▶ **Given:** a BST and a new key, k .
- ▶ **Modify:** the BST, inserting k .
- ▶ Must **Maintain** the BST properties.

Insertion

- ▶ Insert 23 into the BST.



Insertion (The Idea)

- ▶ Traverse the tree as in query to find empty spot where new key should go, keeping track of last node seen.
- ▶ Create new node; make last node seen the parent, update parent's children.
- ▶ Be careful about inserting into empty tree!

```
def insert(self, new_key):
    # assume new_key is unique
    current_node = self.root
    parent = None

    # find place to insert the new node
    while current_node is not None:
        parent = current_node
        if current_node.key < new_key:
            current_node = current_node.right
        else: # current_node.key > new_key
            current_node = current_node.left

    # create the new node
    new_node = Node(key=new_key, parent=parent)

    # if parent is None, this is the root. Otherwise, update the
    # parent's left or right child as appropriate
    if parent is None:
        self.root = new_node
    elif parent.key < new_key:
        parent.right = new_node
    else:
        parent.left = new_node
```

Insertion, Analyzed

- ▶ Worst case: $\Theta(h)$, where h is **height** of tree.

Main Idea

Querying and insertion take $\Theta(h)$ time in the worst case, where h is the height of the tree.

DSC 40B

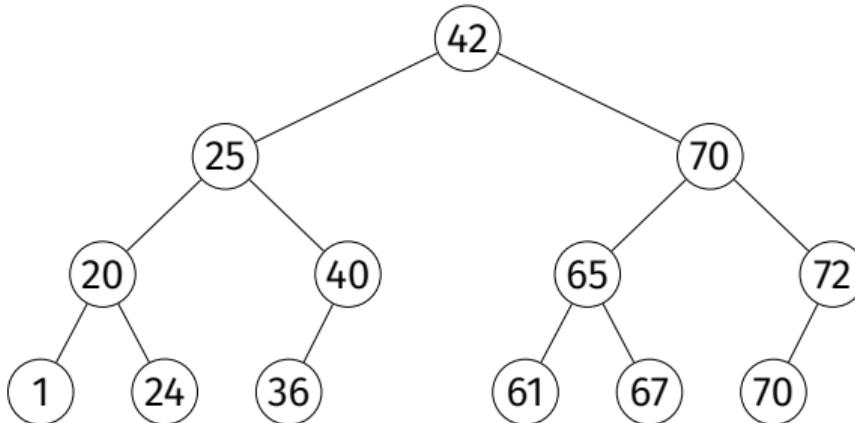
Theoretical Foundations II

Lecture 8 | Part 4

Balanced and Unbalanced BSTs

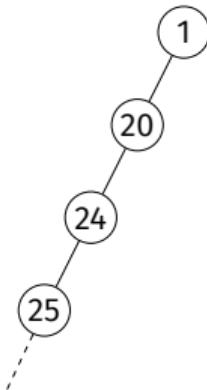
Binary Tree Height

- ▶ In case of very balanced tree, $h = \Theta(\log n)$.
 - ▶ Query, insertion take worst case $\Theta(\log n)$ time in a **balanced** tree.



Binary Tree Height

- ▶ In the case of very unbalanced tree, $h = \Theta(n)$.
 - ▶ Query, insertion take worst case $\Theta(n)$ time in **unbalanced** trees.



Unbalanced Trees

- ▶ Occurs if we insert items in (close to) sorted or reverse sorted order.
- ▶ This is a **common** situation.

Example

- ▶ Insert 1, 2, 3, 4, 5, 6, 7, 8 (in that order).

Time Complexities

| | |
|-----------|-------------|
| query | $\Theta(h)$ |
| insertion | $\Theta(h)$ |

Where h is height, and $h = \Omega(\log n)$ and $h = O(n)$.

Time Complexities (Balanced)

| | |
|-----------|-------------|
| query | $O(\log n)$ |
| insertion | $O(\log n)$ |

Where h is height, and $h = \Omega(\log n)$ and $h = O(n)$.

Worst Case Time Complexities (Unbalanced)

| | |
|-----------|-------------|
| query | $\Theta(n)$ |
| insertion | $\Theta(n)$ |

- ▶ The worst case is **bad**.
 - ▶ Worse than using a sorted array!
- ▶ The worst case is **not rare**.

Main Idea

The operations take linear time in the worst case **unless** we can somehow ensure that the tree is **balanced**.

Self-Balancing Trees

- ▶ There are variants of BSTs that are **self-balancing**.
 - ▶ Red-Black Trees, AVL Trees, etc.
- ▶ Quite complicated to implement correctly.
- ▶ But their height is **guaranteed** to be $\sim \log n$.
- ▶ So insertion, query take $\Theta(\log n)$ in worst case.

Warning!

If asked for the time complexity of a BST operation, be careful! A common mistake is to say that insertion/query are $\Theta(\log n)$ without being told that the tree is balanced.

Main Idea

In general, insertion/query take $\Theta(h)$ time in worst case. If tree is balanced, $h = \Theta(\log n)$, so they take $\Theta(\log n)$ time. If tree is badly unbalanced, $h = O(n)$, and they can take $O(n)$ time.

DSC 40B

Theoretical Foundations II

Lecture 8 | Part 5

Augmenting BSTs

Modifying BSTs

- ▶ Perhaps more than most other data structures, BSTs must be modified (**augmented**) to solve unique problems.

Order Statistics

- ▶ Given n numbers, the **k th order statistic** is the k th smallest number in the collection.

Example

[99, 42, -77, -12, 101]

- ▶ 1st order statistic:
- ▶ 2nd order statistic:
- ▶ 4th order statistic:

Dynamic Set, Many Order Statistics

- ▶ Quickselect finds any order statistic in linear expected time.
- ▶ This is efficient for a static set.
- ▶ Inefficient if set is dynamic.

Goal

- ▶ Create a **dynamic** set data structure that supports fast computation of **any** order statistic.

BST Solution

- ▶ For each node, keep attribute `.size`, containing # of nodes in subtree rooted at current node
- ▶ Property:¹
 $x.size = x.left.size + x.right.size + 1$

¹If a left or right child doesn't exist, consider its size zero.

Computing Sizes

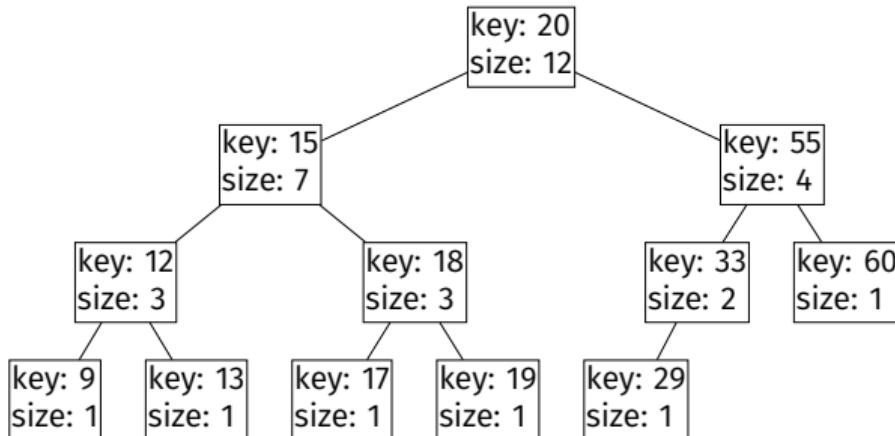
```
def add_sizes_to_tree(node):
    if node is None:
        return 0
    left_size = add_sizes_to_tree(node.left)
    right_size = add_sizes_to_tree(node.right)
    node.size = left_size + right_size + 1
    return node.size
```

Note

- ▶ Also need to maintain size upon inserting a node.

Computing Order Statistics

- ▶ 8th? 2nd? 12th



Augmenting Data Structures

- ▶ This is just one example, but many more.
- ▶ Understanding how BSTs work is key to augmenting them.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 1

Warmup

Exercise

- ▶ How fast can we query/insert with these data structures?

| | Query | Insert |
|----------------------|-------|--------|
| Unsorted linked list | | |
| Unsorted array | | |
| Sorted array | | |
| BST | | |

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 2

Direct Address Tables

Counting Frequencies

- ▶ How many times does each age appear?

| PID | Name | Age |
|-------|---------|-----|
| A1843 | Wan | 24 |
| A8293 | Deveron | 22 |
| A9821 | Vinod | 41 |
| A8172 | Aleix | 17 |
| A2882 | Kayden | 4 |
| A1829 | Raghu | 51 |
| A9772 | Cui | 48 |
| : | : | : |

Exercise

What data structure would you use to store the age counts?

Direct Address Tables

- ▶ Idea: keep an **array** `arr` of length, say, 125.
- ▶ Initialize to zero.
- ▶ If we see age x , increment `arr[x]` by one.

Building the Table

```
# loading the table
table = np.zeros(125)

for age in ages:
    table[age] += 1
```

- ▶ Time complexity if there are n people?

Query

```
# query: how many people are 55?  
print(table[55])
```

- ▶ Time complexity if there are n people?

Counting Names

- ▶ How many times does each name appear?

| PID | Name | Age |
|-------|---------|-----|
| A1843 | Wan | 24 |
| A8293 | Deveron | 22 |
| A9821 | Vinod | 41 |
| A8172 | Aleix | 17 |
| A2882 | Kayden | 4 |
| A1829 | Raghu | 51 |
| A9772 | Cui | 48 |
| : | : | : |

Downsides

- ▶ DATs are **fast**.
- ▶ What are the downsides of DATs?
- ▶ Could we use a DAT to store:
 - ▶ zip codes?
 - ▶ phone numbers?
 - ▶ credit card numbers?
 - ▶ names?

Downsides

- ▶ Things being stored must be integers, or convertible to integers
 - ▶ why? valid array indices
- ▶ Must come from a small range of possibilities
 - ▶ why? memory usage. example: phone numbers

Hash Tables

- ▶ Insight: anything can be “converted” to an integer through **hashing**.
- ▶ But not uniquely!
- ▶ Hash tables have many of the same advantages as DATs, but work more generally.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 3

Hashing

Hashing

- ▶ One of the most important ideas in CS.
- ▶ Tons of uses:
 - ▶ Verifying message integrity.
 - ▶ Fast queries on a large data set.
 - ▶ Identify if file has changed in version control.

Hash Function

- ▶ A **hash function** takes a (large) object and returns a (smaller) “fingerprint” of that object.
- ▶ Usually the fingerprint is a number, guaranteed to be in some range.

How?

- ▶ Looking at certain bits, combining them in ways that *look* random (but aren't!)

Hash Function Properties

- ▶ Hashing same thing twice returns the same hash.
- ▶ Unlikely that different things have same fingerprint.
 - ▶ But not impossible!

Collisions

- ▶ Hash functions map objects to numbers in a defined range.
 - ▶ Example: given image, return number in $[0, 1, 2, \dots, 1024]$
- ▶ There will be two images with the same hash.
 - ▶ **Pigeonhole principle:** if there are n pigeons, $< n$ holes, there will a hole with ≥ 2 pigeons.
- ▶ **Collision:** two objects have the same hash

“Good” Hash Functions

- ▶ A good hash function tries to minimize collisions.

Hashing in Python

- The `hash` function computes a hash.

```
»> hash("This is a test")
-670458579957477203
»> hash("This is a test")
-670458579957477203
»> hash("This is a test!")
1860306055874153109
```

MD5

- ▶ MD5 is a **cryptographic** hash function.
 - ▶ Hard to “reverse engineer” input from hash.
- ▶ Returns a *really large* number in hex.

a741d8524a853cf83ca21eabf8cea190

- ▶ Used to “fingerprint” whole files.

Example

```
> echo "My name is Justin" | md5  
a741d8524a853cf83ca21eabf8cea190  
> echo "My name is Justin" | md5  
a741d8524a853cf83ca21eabf8cea190  
> echo "My name is Justin!" | md5  
f11eed2391bbdoa5a2355397c089fafd
```

Example

```
> md5 slides.pdf  
e3fd4370fd30ceb978390004e07b9df
```

Why?

- ▶ I release a piece of software.
- ▶ I host it on Google Drive.
- ▶ Someone (Google, US Gov., etc.) decides to insert extra code into software to spy on users.
- ▶ You have no way of knowing.

Why?

- ▶ I release a piece of software & **publish the hash**.
- ▶ I host it on Google Drive.
- ▶ Someone inserts extra code.
- ▶ You download the software and hash it. If hash is different, you know the file has been changed!

Another Use: De-duplication

- ▶ Building a massive training set of images.
- ▶ Given a new image, is it already in my collection?
- ▶ Don't need to compare images pixel-by-pixel!
- ▶ Instead, compare **hashes**.

Hashing for Data Scientists

- ▶ Don't need to know much about *how* the hash function works.
- ▶ But should know how they are used.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 4

Hash Tables

Membership Queries

- ▶ **Given:** a collection of n numbers and a target t .
- ▶ **Find:** determine if t is in the collection.

Goal

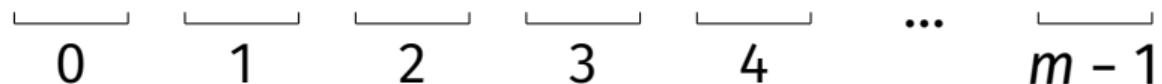
- ▶ DATs are fast, but won't work for things that aren't numbers in a small range.
- ▶ Idea: hash objects to numbers in a small range, use a DAT.
- ▶ But must deal with collisions.

Hash Tables

- ▶ Pick a table size m .
 - ▶ Usually $m \approx$ number of things you'll be storing.
- ▶ Create hash function to turn input into a number in $\{0, 1, \dots, m - 1\}$.
- ▶ Create DAT with m bins.

Example

```
hash('hello') == 3  
hash('data') == 0  
hash('science') == 4
```

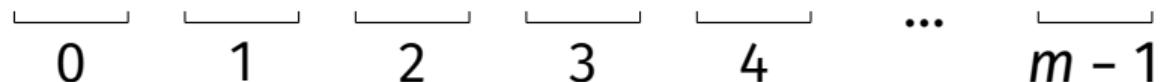


Collisions

- ▶ The **universe** is the set of all possible inputs.
- ▶ This is usually much larger than m (even infinite).
- ▶ Not possible to assign each input to a unique bin.
- ▶ If `hash(a) == hash(b)`, there is a **collision**.

Example

```
hash('hello') == 3  
hash('data') == 0  
hash('san diego') == 3
```



Chaining

- ▶ Collisions stored in same bin, in linked list.
- ▶ **Query:** Hash to find bin, then linear search.



The Idea

- ▶ A good hash function will utilize all bins evenly.
 - ▶ Looks like uniform random distribution.
- ▶ If $m \approx n$, then only a few elements in each bin.
- ▶ As we add more elements, we need to add bins.

Average Case

- ▶ n elements in bin.
- ▶ m bins.
- ▶ Assume elements placed randomly in bins¹.
- ▶ Expected bin size:

¹Of course, they are placed deterministically.

Average Case

- ▶ n elements in bin.
- ▶ m bins.
- ▶ Assume elements placed randomly in bins¹.
- ▶ Expected bin size: n/m

¹Of course, they are placed deterministically.

Analysis

- ▶ Query:
 - ▶ Time to find correct bin:
 - ▶ Expected number of elements in the bin:
 - ▶ Time to perform linear search:
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin:
 - ▶ Time to perform linear search:
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search:
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total: $\Theta(1 + n/m)$

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total: $\Theta(1 + n/m)$
 - ▶ We usually guarantee $m = O(n)$

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total: $\Theta(1 + n/m)$
 - ▶ We usually guarantee $m = O(n)$
 - ▶ Expected time: $\Theta(1)$.

Worst Case

- ▶ Everything hashed to same bin.
 - ▶ Really unlikely!
 - ▶ Adversarial attack?
- ▶ Query:
 - ▶ $\Theta(1)$ to find bin
 - ▶ $\Theta(n)$ for linear search.
 - ▶ Total: $\Theta(n)$.

Exercise

What is the worst case time complexity of inserting an element into a hash table that uses chaining with linked lists?

Growing the Hash Table

- ▶ Insertions take $\Theta(1)$ **unless** the hash table needs to grow.
- ▶ We need to ensure that $m \leq c \cdot n$.
 - ▶ Otherwise, too many collisions.
- ▶ If we add a bunch of elements, we'll need to increase m .
- ▶ Increasing m means allocating a new array, $\Theta(m) = \Theta(n)$ time.

Main Idea

Hash tables support constant (expected) time insertion and membership queries.

Dictionaries

- ▶ Hash tables can also be used to store (key, value) pairs.
- ▶ Often called **dictionaries** or **associative arrays**.

Hashing in Python

- ▶ `dict` and `set` implement hash tables.
- ▶ Querying is done using `in`:

```
»> # make a set
»> L = {3, 6, -2, 1, 7, 12}
»> 1 in L # Theta(1)
False
»> 7 in L # Theta(1)
True
```

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 5

Fast Algorithms with Hash Tables

Faster Algorithms

- ▶ Hashing is a super common trick.
- ▶ The “best” solution to interview problems often involves hashing.

Example 1: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ Find two whose durations sum to **exactly D** .

Recall: Previous Solutions

- ▶ Brute force: $\Theta(n^2)$.
- ▶ Sort, use sorted structure: $\Theta(n \log n) + \Theta(n)$.
- ▶ Theoretical lower bound: $\Omega(n)$?
- ▶ Can we speed this up with hash tables?

Idea

- ▶ To use hash tables, we want to frame problem as a **membership query**.

Example

- ▶ Suppose flight is 360 minutes long.
- ▶ Suppose first movie is fixed: 120 minutes.
- ▶ Is there a movie lasting $(360 - 120) = 140$ minutes?

```
def optimize_entertainment_hash(times, D):
    hash_table = dict()
    for i, time in enumerate(times):
        hash_table[time] = i

    for i, time in enumerate(times):
        target = D - time
        if target in hash_table:
            return i, hash_table[target]
```

Example 2: Anagrams

Definition

Two strings w_1 and w_2 are **anagrams** if the letters of w_1 one can be permuted to make w_2 .

Examples

- ▶ abcd / dbca
- ▶ listen / silent
- ▶ sandiego / doginsea

Problem

- ▶ Given a collection of n strings, determine if any two of them are anagrams.

Exercise

Design an efficient algorithm for solving this problem. What is its time complexity?

Solution

- ▶ We need to turn this into a **membership query**.
- ▶ **Trick:** two strings are anagrams iff

```
sorted(w_1) == sorted(w_2)
```

```
def any_anagrams(words):
    seen = set()
    for word in words:
        w = sorted(word)
        if w in seen
            return True
    else:
        seen.add(w)
```

Hashing **Downsides**

- ▶ Problem must involve **membership query**.

Example: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ Find two whose added durations is **closest** to D .

Hashing **Downsides**

- ▶ No locality: similar items map to different bins.
- ▶ There is no way to quickly query entry closest to given input.

Example: Number of Elements

- ▶ Given a collection of n numbers and two endpoints, a and b , determine how many of the numbers are contained in $[a, b]$.
- ▶ Not a membership query.
- ▶ Idea: **sort** and use modified binary search.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 6

Hash Table Drawbacks

Hashing **Downsides**

- ▶ No locality: similar items map to different bins.
- ▶ But we often want similar items at the same time.
- ▶ Results in many **cache misses, slow.**

Hashing **Downsides**

- ▶ Memory overhead.

Hash Tables vs. BSTs

- ▶ Hash Table: $\Theta(1)$ insertion, query (expected time).
- ▶ BST: $\Theta(\log n)$ insertion, query (if balanced).
- ▶ Why ever use a BST?

Hash Tables vs. BSTs

- ▶ Hash tables keep items in arbitrary order.
- ▶ Example: how many elements are in the interval [3, 23]?
- ▶ Example: what is the min/max/median?
- ▶ BSTs win when order is important.

DSC 40B

Theoretical Foundations II

Lecture 10 | Part 1

News

Midterm 01

- ▶ Midterm 01 grades released.
- ▶ Remember: redemption exam at end of the quarter.
- ▶ Recap video with solutions/explanations will be posted soon.

DSC 40B

Theoretical Foundations II

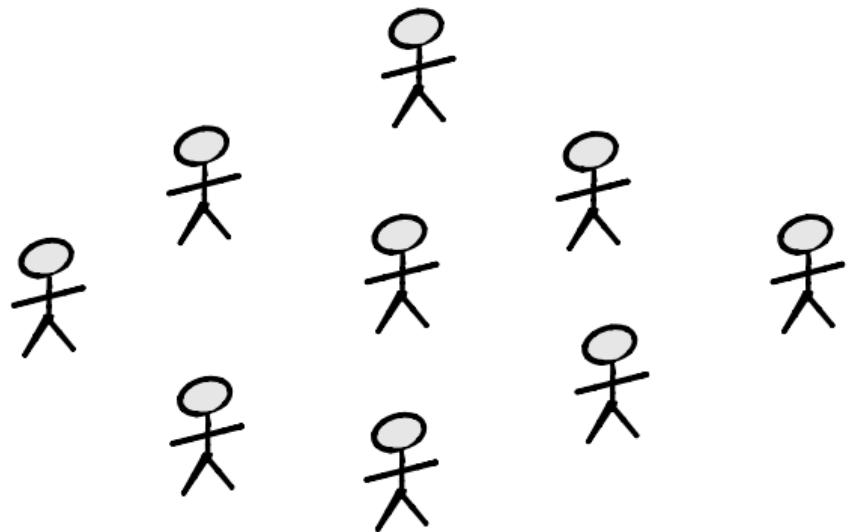
Lecture 10 | Part 2

Graphs

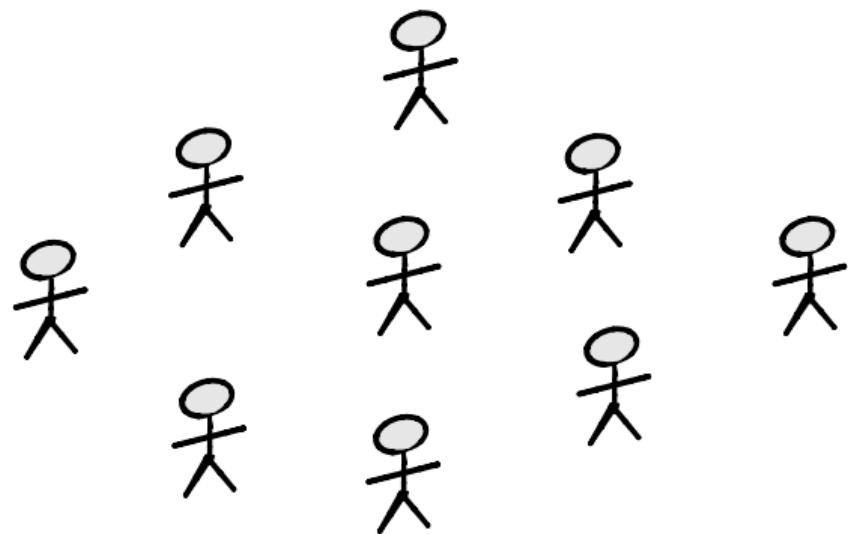
Data Types

- ▶ **Feature vectors**
 - ▶ We care about attributes of individuals.
- ▶ **Graphs**
 - ▶ We care about relationships between individuals.

Example: Facebook



Example: Twitter

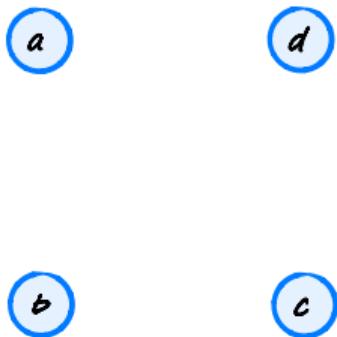


Definition

A **directed graph** (or **digraph**) G is a pair (V, E) where V is a finite set of **nodes** (or **vertices**) and E is a set of ordered pairs (the **edges**).

Example:

$$V = \{a, b, c, d\}$$
$$E = \{(a, c), (a, b), (d, b), (b, d), (b, b)\}$$



Directed Graphs (More Formally)

E is a subset of the **Cartesian product**, $V \times V$.

Example:

$$\{a, b, c\} \times \{1, 2\} =$$

Consequences

Because the edge set of a directed graph is allowed to be *any* subset of $V \times V$:

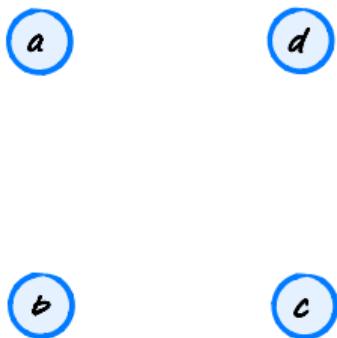
- ▶ the edges have directions.
 - ▶ e.g., (a, b) is “from a to b ”
- ▶ can have “opposite” edges.
 - ▶ e.g., (a, b) and (b, a) .
- ▶ can have “self-loops”
 - ▶ e.g., (a, a)

Definition

An **undirected graph** G is a pair (V, E) where V is a finite set of **nodes** (or **vertices**) and E is a set of unordered, distinct pairs (the **edges**).

Example:

$$V = \{a, b, c, d\}$$
$$E = \{\{a, c\}, \{a, b\}, \{d, b\}\}$$



Undirected Graphs (More Formally)

An edge in an undirected graph is a set $\{u, v\}$ where $u \neq v$. This has consequences:

- ▶ the edges have **no direction**.
 - ▶ e.g., $\{a, b\}$ is **not** “from” a “to” b .
- ▶ **cannot** have “opposite” edges.
 - ▶ e.g., $\{a, b\}$ and $\{b, a\}$ are the same.
- ▶ **cannot** have “self-loops”
 - ▶ e.g., $\{a, a\}$ is not a valid edge

Notational Note

Although edges in undirected graphs are sets, we typically write them as pairs: (u, v) instead of $\{u, v\}$.

Summary

- ▶ Edges have direction:
 - ▶ Directed: **yes**
 - ▶ Undirected: **no**
- ▶ Self-loops, (u, u) ?
 - ▶ Directed: **yes**
 - ▶ Undirected: **no**
- ▶ Opposite edges, (u, v) and (v, u) ?
 - ▶ Directed: **yes**
 - ▶ Undirected: **no** (they are the same edge)

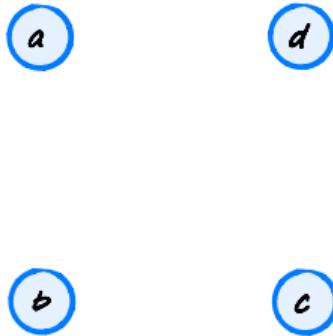
Note

Neither directed nor undirected graphs can have
duplicate edges¹

¹There are other definitions which allow duplicate edges.

Note

Graphs don't need to be "connected"²



²There are other definitions which allow duplicate edges.

Exercise

What is the greatest number edges possible in a **directed** graph?

Counting Edges

What is the greatest number edges possible in a **directed** graph?

a

d

b

c

Exercise

What is the greatest number edges possible in an **undirected** graph?

Counting Edges

What is the greatest number edges possible in an **undirected** graph?

a

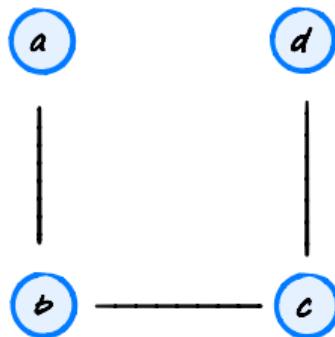
d

b

c

Degree

The **degree** of a node in an undirected graph is the number of edges containing that node.



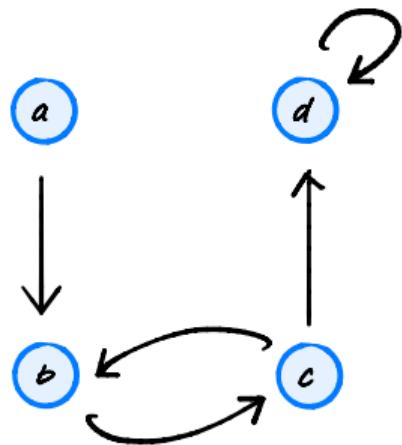
In-Degree/Out-Degree

The **in-degree** of a node in an directed graph is the number of edges **entering** that node.

The **out-degree** of a node in an directed graph is the number of edges **leaving** that node.

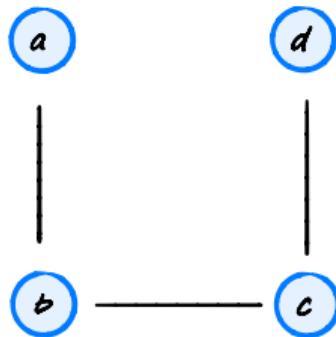
The **degree** of a node in a directed graph is the in-degree + out-degree.

Examples



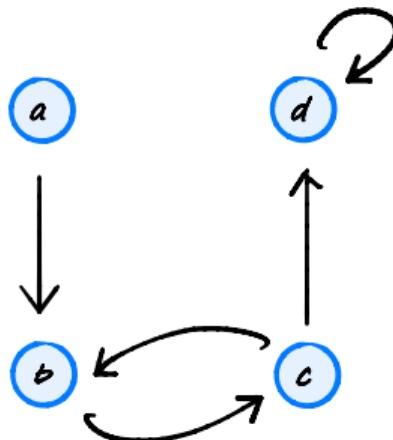
Neighbors

Definition: in an undirected graph, the set of **neighbors** of a node u is the set of all nodes which share an edge with u .



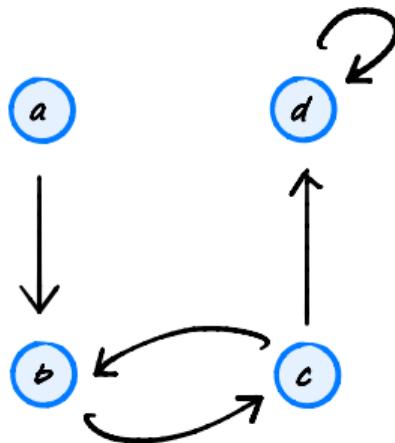
Predecessors

Definition: in an directed graph, the set of **predecessors** of a node u is the set of all nodes which are at the **start** of an edge **entering** u .



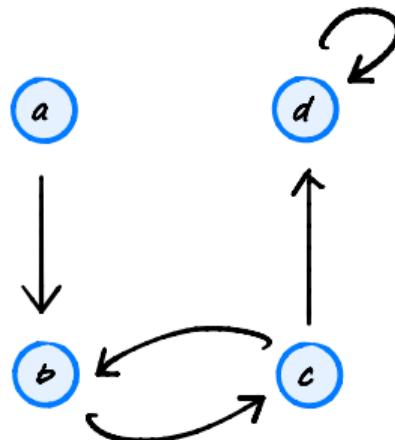
Successors

Definition: in an directed graph, the set of **successors** of a node u is the set of all nodes which are at the **end** of an edge **leaving** u .



A Convention

In a directed graph, the **neighbors** of u are the **successors** of u .



DSC 40B

Theoretical Foundations II

Lecture 10 | Part 3

Paths

Example

- ▶ Consider a graph of direct flights.
- ▶ Each node is an airport.
- ▶ Each edge is a direct flight.
- ▶ Should the graph be directed or undirected?

Example



Example

- ▶ Can we get from San Diego to Columbus?
- ▶ Not with a single edge.
- ▶ But with a **path**.

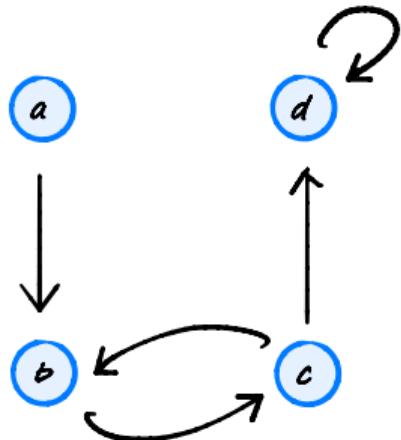
Definition

A **path** from u to u' in a (directed or undirected) graph $G = (V, E)$ is a sequence of one or more nodes $u = v_0, v_1, \dots, v_k = u'$ such that there is an edge between each consecutive pair of nodes in the sequence.

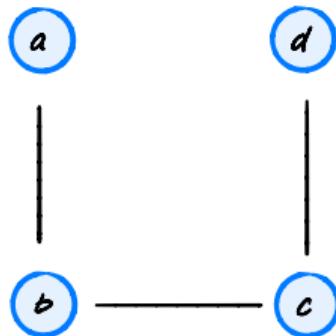
Path Length

Definition: The **length** of a path is the number of nodes in the sequence, minus one. Paths of length zero are possible!

Examples

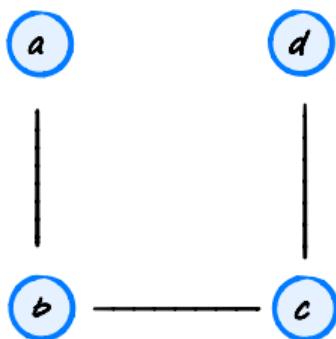


Examples



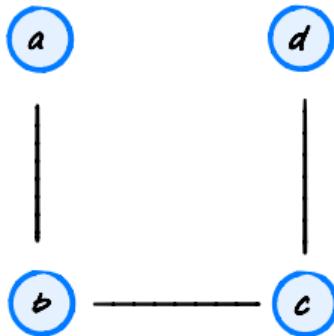
Note

Paths **can** go through the same node more than once!



Simple Paths

Definition: A **simple path** is a path in which every node is unique.



Reachability

Definition: node v is **reachable** from node u if there is a path from u to v .

Reachability and Directedness

- ▶ If G is undirected, reachability is symmetric.
 - ▶ If u reachable from v , then v reachable from u .
- ▶ If G is directed, reachability is **not** symmetric.
 - ▶ If u reachable from v , then v may/may not be reachable from u .

Important Trivia

- ▶ In any graph, any node is **reachable** from **itself**.

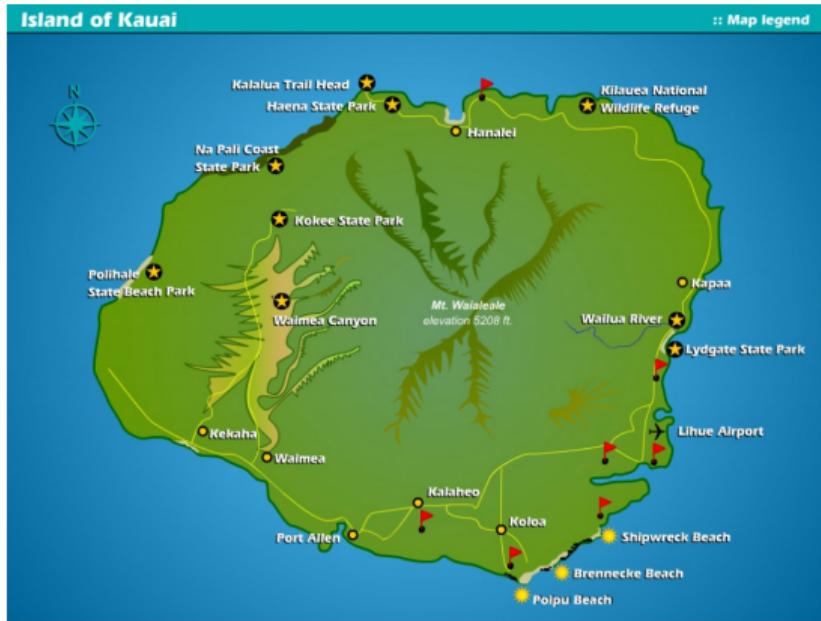
DSC 40B

Theoretical Foundations II

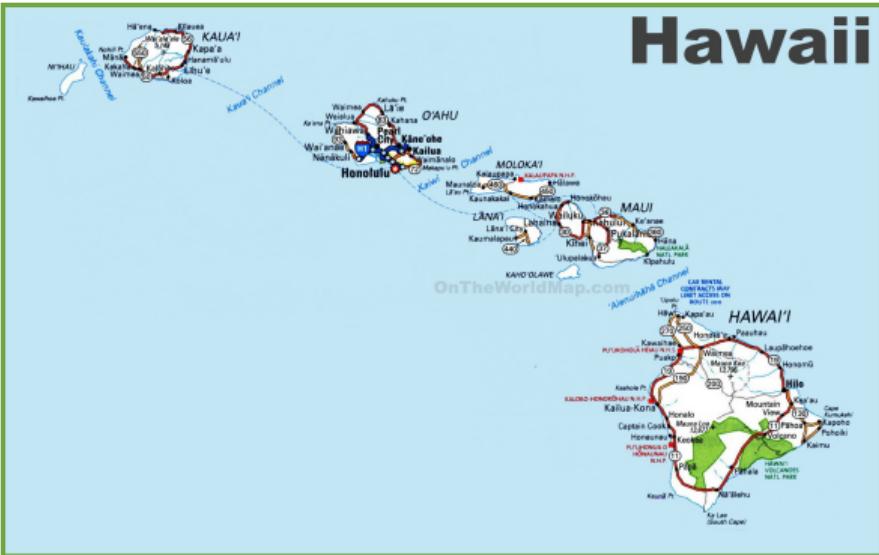
Lecture 10 | Part 4

Connected Components

Example



Example



Connectedness

A graph is **connected** if every node u is reachable from every other node v . Otherwise, it is **disconnected**.

Equivalent: there is a path between every pair of nodes.

Connected Components

A **connected component** is a maximally-connected set of nodes.

I.e., if $G = (V, E)$ is an undirected graph, a connected component is a set $C \subset V$ such that

- ▶ any pair $u, u' \in C$ are reachable from one another; and
- ▶ if $u \in C$ and $z \notin C$ then u and z are not reachable from one another.

Exercise

What are the connected components?

$$V = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E = \{(0, 2), (1, 5), (3, 1), (2, 4), (0, 4), (5, 3)\}$$

Example

What are the connected components?

$$V = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E = \{(0, 2), (1, 5), (3, 1), (2, 4), (0, 4), (5, 3)\}$$

DSC 40B

Theoretical Foundations II

Lecture 10 | Part 5

Adjacency Matrices

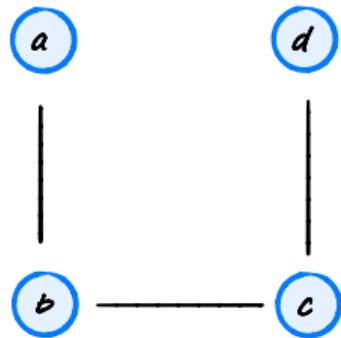
Representations

- ▶ How do we **store** a graph in a computer's memory?
- ▶ Three approaches:
 1. Adjacency matrices.
 2. Adjacency lists.
 3. “Dictionary of sets”

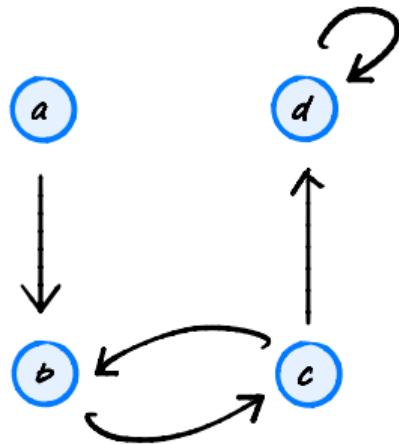
Adjacency Matrices

- ▶ Assume nodes are numbered $0, 1, \dots, |V| - 1$
- ▶ Allocate a $|V| \times |V|$ (Numpy) array
- ▶ Fill array as follows:
 - ▶ $\text{arr}[i, j] = 1$ if $(i, j) \in E$
 - ▶ $\text{arr}[i, j] = 0$ if $(i, j) \notin E$

Example



Example



Observations

- ▶ If G is undirected, matrix is symmetric.
- ▶ If G is directed, matrix may not be symmetric.

Time Complexity

| operation | code | time |
|---------------|--------------------------------|---------------|
| edge query | <code>adj[i, j] == 1</code> | $\Theta(1)$ |
| degree(i) | <code>np.sum(adj[i, :])</code> | $\Theta(V)$ |

Space Requirements

- ▶ Uses $|V|^2$ bits, even if there are very few edges.
- ▶ But most real-world graphs are **sparse**.
 - ▶ They contain many fewer edges than possible.

Example: Facebook

- ▶ Facebook has 2 billion users.

$$(2 \times 10^9)^2 = 4 \times 10^{18} \text{ bits}$$

= 500 petabits

≈ 6500 years of video at 1080p

≈ 60 copies of the internet as it was in 2000

Adjacency Matrices and Math

- ▶ Adjacency matrices are useful mathematically.
- ▶ Example: (i, j) entry of A^2 gives number of hops of length 2 between i and j .

DSC 40B

Theoretical Foundations II

Lecture 10 | Part 6

Adjacency Lists

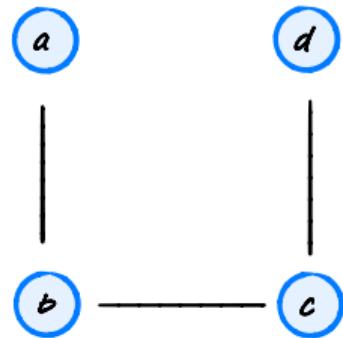
What's Wrong with Adjacency Matrices?

- ▶ Requires $\Theta(|V|^2)$ storage.
- ▶ Even if the graph has no edges.
- ▶ **Idea:** only store the edges that exist.

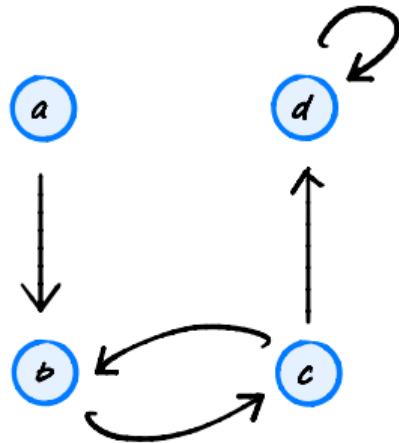
Adjacency Lists

- ▶ Create a list adj containing $|V|$ lists.
- ▶ $\text{adj}[i]$ is list containing the neighbors of node i .

Example



Example



Observations

- ▶ If G is undirected, each edge appears twice.
- ▶ If G is directed, each edge appears once.

Time Complexity

| operation | code | time |
|--------------------|--------------------------|----------------------------|
| edge query | <code>j in adj[i]</code> | $\Theta(\text{degree}(i))$ |
| $\text{degree}(i)$ | <code>len(adj[i])</code> | $\Theta(1)$ |

Space Requirements

- ▶ Need $\Theta(|V|)$ space for outer list.
- ▶ Plus $\Theta(|E|)$ space for inner lists.
- ▶ In total: $\Theta(|V| + |E|)$ space.

Example: Facebook

- ▶ Facebook has 2 billion users, 400 billion friendships.
- ▶ If each edge requires 32 bits:
$$\begin{aligned} & (2 \text{ bits} \times 200 \times (2 \text{ billion})) \\ &= 64 \times 400 \times 10^9 \text{ bits} \\ &= 3.2 \text{ terabytes} \\ &= 0.04 \text{ years of HD video} \end{aligned}$$

DSC 40B

Theoretical Foundations II

Lecture 10 | Part 7

Dictionary of Sets

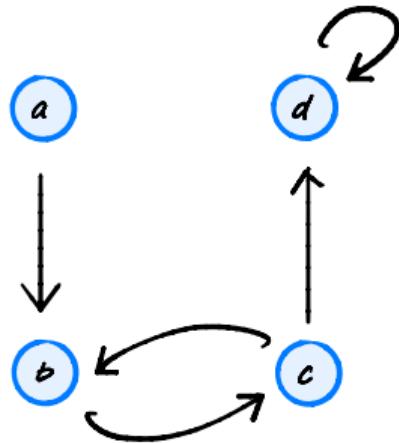
Tradeoffs

- ▶ Adjacency matrix: fast edge query, lots of space.
- ▶ Adjacency list: slower edge query, space efficient.
- ▶ Can we have the best of both?

Idea

- ▶ Use **hash tables**.
- ▶ Replace inner edge lists by **sets**.
- ▶ Replace outer list with **dict**.
 - ▶ Doesn't speed things up, but allows nodes to have arbitrary labels.

Example



Time Complexity

| operation | code | time |
|---------------|--------------------------|---------------------|
| edge query | <code>j in adj[i]</code> | $\Theta(1)$ average |
| degree(i) | <code>len(adj[i])</code> | $\Theta(1)$ average |

Space Requirements

- ▶ Requires only $\Theta(E)$.
- ▶ But there is overhead to using hash tables.

Dict-of-sets implementation

- ▶ Install with `pip install dsc40graph`
- ▶ Import with `import dsc40graph`
- ▶ Docs: <https://eldridgejm.github.io/dsc40graph/>
- ▶ Source code:
<https://github.com/eldridgejm/dsc40graph>
- ▶ Will be used in HW coding problems.

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 1

Adjacency Matrices (Recap)

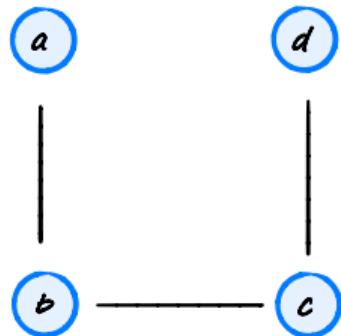
Representations

- ▶ How do we **store** a graph in a computer's memory?
- ▶ Three approaches:
 1. Adjacency matrices.
 2. Adjacency lists.
 3. “Dictionary of sets”

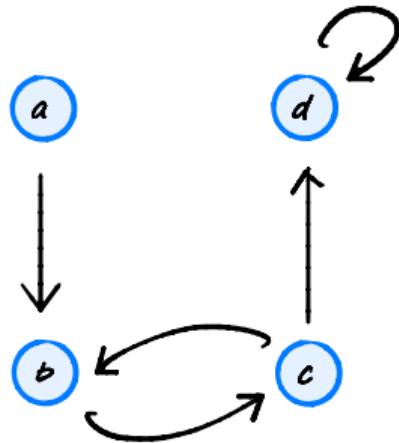
Adjacency Matrices

- ▶ Assume nodes are numbered $0, 1, \dots, |V| - 1$
- ▶ Allocate a $|V| \times |V|$ (Numpy) array
- ▶ Fill array as follows:
 - ▶ $\text{arr}[i, j] = 1$ if $(i, j) \in E$
 - ▶ $\text{arr}[i, j] = 0$ if $(i, j) \notin E$

Example



Example



Observations

- ▶ If G is undirected, matrix is symmetric.
- ▶ If G is directed, matrix may not be symmetric.

Time Complexity

| operation | code | time |
|---------------|--------------------------------|---------------|
| edge query | <code>adj[i, j] == 1</code> | $\Theta(1)$ |
| degree(i) | <code>np.sum(adj[i, :])</code> | $\Theta(V)$ |

Space Requirements

- ▶ Uses $|V|^2$ bits, even if there are very few edges.
- ▶ But most real-world graphs are **sparse**.
 - ▶ They contain many fewer edges than possible.

Example: Facebook

- ▶ Facebook has 2 billion users.

$$(2 \times 10^9)^2 = 4 \times 10^{18} \text{ bits}$$

= 500 petabits

≈ 6500 years of video at 1080p

≈ 60 copies of the internet as it was in 2000

Adjacency Matrices and Math

- ▶ Adjacency matrices are useful mathematically.
- ▶ Example: (i, j) entry of A^2 gives number of hops of length 2 between i and j .

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 2

Adjacency Lists

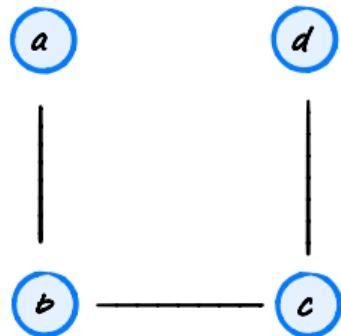
What's Wrong with Adjacency Matrices?

- ▶ Requires $\Theta(|V|^2)$ storage.
- ▶ Even if the graph has no edges.
- ▶ **Idea:** only store the edges that exist.

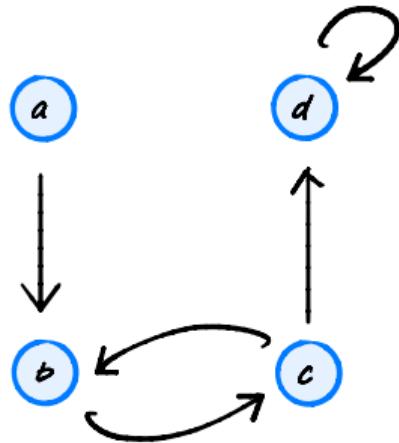
Adjacency Lists

- ▶ Create a list adj containing $|V|$ lists.
- ▶ $\text{adj}[i]$ is list containing the neighbors of node i .

Example



Example



Observations

- ▶ If G is undirected, each edge appears twice.
- ▶ If G is directed, each edge appears once.

Time Complexity

| operation | code | time |
|--------------------|--------------------------|----------------------------|
| edge query | <code>j in adj[i]</code> | $\Theta(\text{degree}(i))$ |
| $\text{degree}(i)$ | <code>len(adj[i])</code> | $\Theta(1)$ |

Space Requirements

- ▶ Need $\Theta(|V|)$ space for outer list.
- ▶ Plus $\Theta(|E|)$ space for inner lists.
- ▶ In total: $\Theta(|V| + |E|)$ space.

Example: Facebook

- ▶ Facebook has 2 billion users, 400 billion friendships.
- ▶ If each edge requires 32 bits:
$$\begin{aligned} & (2 \text{ bits} \times 200 \times (2 \text{ billion})) \\ &= 64 \times 400 \times 10^9 \text{ bits} \\ &= 3.2 \text{ terabytes} \\ &= 0.04 \text{ years of HD video} \end{aligned}$$

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 3

Dictionary of Sets

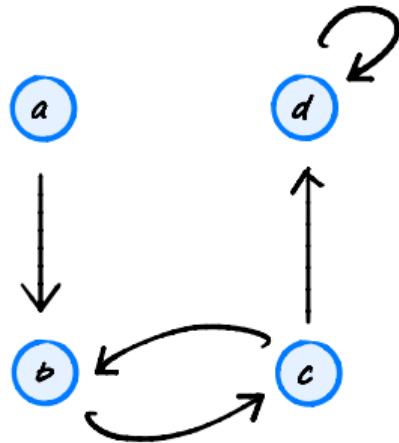
Tradeoffs

- ▶ Adjacency matrix: fast edge query, lots of space.
- ▶ Adjacency list: slower edge query, space efficient.
- ▶ Can we have the best of both?

Idea

- ▶ Use **hash tables**.
- ▶ Replace inner edge lists by **sets**.
- ▶ Replace outer list with **dict**.
 - ▶ Doesn't speed things up, but allows nodes to have arbitrary labels.

Example



Time Complexity

| operation | code | time |
|---------------|--------------------------|---------------------|
| edge query | <code>j in adj[i]</code> | $\Theta(1)$ average |
| degree(i) | <code>len(adj[i])</code> | $\Theta(1)$ average |

Space Requirements

- ▶ Requires only $\Theta(E)$.
- ▶ But there is overhead to using hash tables.

Dict-of-sets implementation

- ▶ Install with `pip install dsc40graph`
- ▶ Import with `import dsc40graph`
- ▶ Docs: <https://eldridgejm.github.io/dsc40graph/>
- ▶ Source code:
<https://github.com/eldridgejm/dsc40graph>
- ▶ Will be used in HW coding problems.

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 4

Graph Search Strategies

How do we:

- ▶ determine if there is a path between two nodes?
- ▶ check if graph is connected?
- ▶ count connected components?

Search Strategies

- ▶ A **search strategy** is a procedure for exploring a graph.
- ▶ Different strategies are useful in different situations.

Node Statuses

At any point during a search, a node is in exactly one of three states:

- ▶ **visited**
- ▶ **pending** (discovered, but not yet visited)
- ▶ **undiscovered**

Rules

- ▶ At every step, next visited node chosen from among **pending** nodes.
- ▶ When a node is marked as **visited**, all of its neighbors have been marked as **pending**.

Choosing the next Node

How to choose among pending nodes?

- ▶ Idea 1: Visit **newest** pending (**depth-first search**).
- ▶ Idea 2: Visit **oldest** pending (**breadth-first search**).

Main Idea

DFS and BFS each discover different properties of the graph.

For example, we'll see that BFS is useful for finding shortest paths (DFS in general is not).

DSC 40B

Theoretical Foundations II

Lecture 11 | Part 5

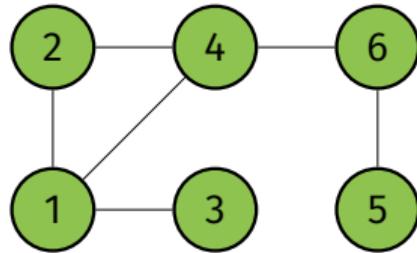
Breadth-First Search

Breadth-First Search

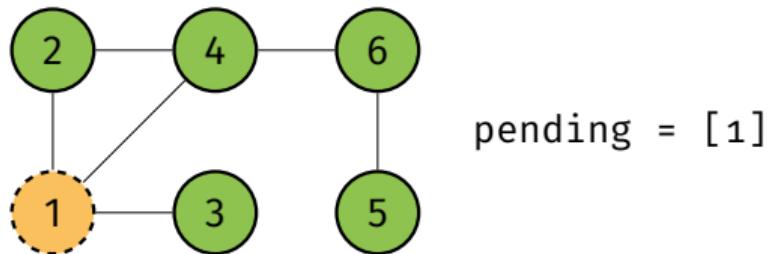
- ▶ At every step:
 1. Visit oldest pending node.
 2. Mark its undiscovered neighbors as pending.
- ▶ Convention: in this class, neighbors produced in sorted order.¹

¹In general, the order in which a node's neighbors produced is arbitrary.

Example

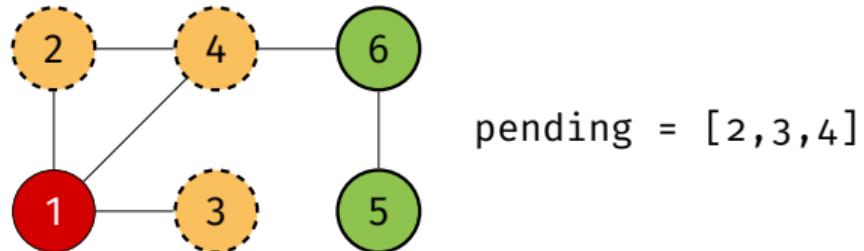


Example



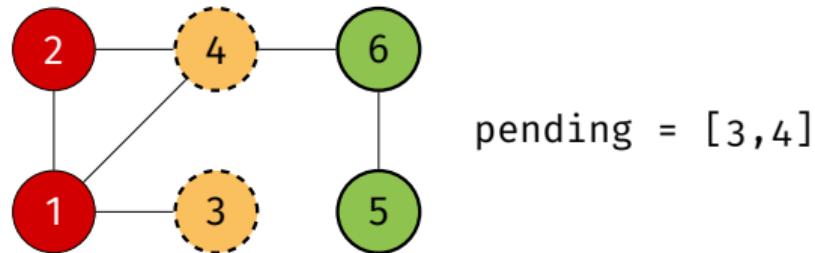
Before iterating.

Example



After 1st iteration.

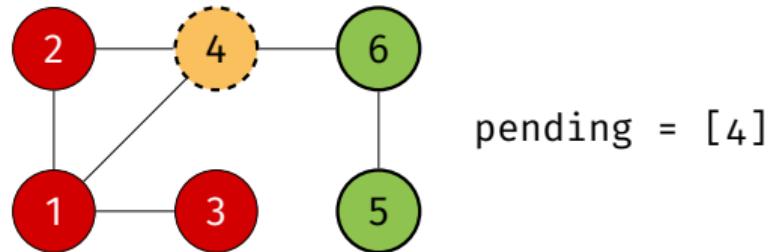
Example



After 2nd iteration.

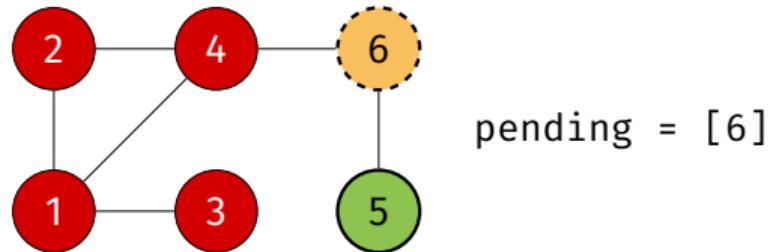
Exercise: what will the picture look like after the next two iterations?

Example



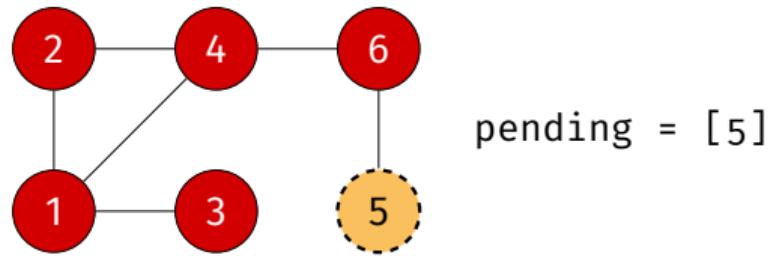
After 3rd iteration.

Example



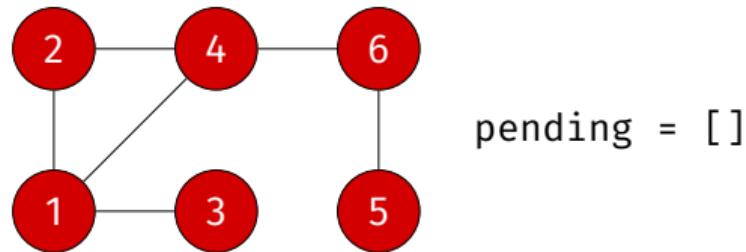
After 4th iteration.

Example



After 5th iteration.

Example



After 6th iteration.

Implementation

- ▶ To store pending nodes, use a FIFO **queue**.
- ▶ While queue is not empty:
 - ▶ Pop a node, u .
 - ▶ Add undiscovered neighbors to queue.

Queues in Python

- ▶ Want $\Theta(1)$ time pops/appends on either side.
- ▶ `from collections import deque ("deck")`.
 - ▶ `.popleft()` and `.pop()`
 - ▶ `list` doesn't have right time complexity!
 - ▶ `import queue` isn't what you want!
- ▶ Keep track of node status attribute using dictionary.

Exercise

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        # EXERCISE: fill this in...
```

BFS

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Note

- ▶ What does this code actually *return*?

Note

- ▶ What does this code actually *return*?
- ▶ Nothing, yet. It is a *foundation*.

Note

- ▶ BFS works just as well for directed graphs.

DSC 40B

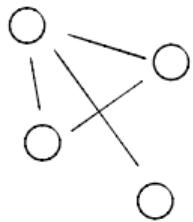
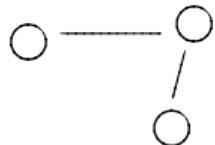
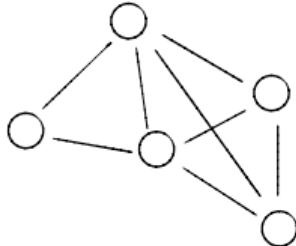
Theoretical Foundations II

Lecture 11 | Part 6

Analysis of BFS

Exercise

What will bfs do when run on a disconnected graph?



Claim

- ▶ bfs with source u will visit all nodes reachable from u (and only those nodes).
- ▶ Useful!
 - ▶ Is there a path between u and v ?
 - ▶ Is graph connected?

Exploring with BFS

- ▶ BFS will visit all nodes reachable from source.
- ▶ If **disconnected**, BFS will not visit all nodes.
- ▶ We can do so with a **full BFS**.
 - ▶ Idea: “re-start” BFS on undiscovered node.
 - ▶ Must pass statuses between calls.

Making Full BFS

Modify bfs to accept statuses:

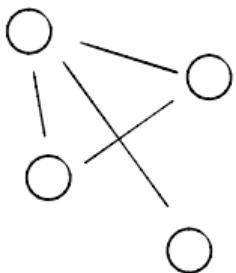
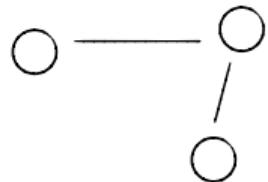
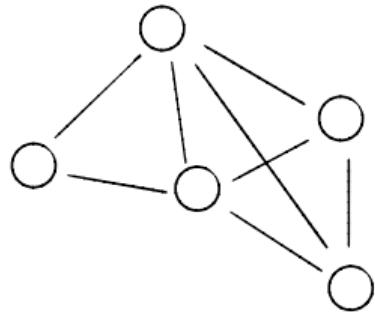
```
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
    # ...
```

Making Full BFS

Call bfs multiple times:

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered':
            bfs(graph, node, status)
```

Example



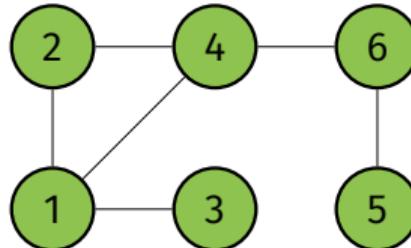
Observation

- ▶ If there are k connected components, bfs in line 5 is called exactly k times.

```
1 def full_bfs(graph):
2     status = {node: 'undiscovered' for node in graph.nodes}
3     for node in graph.nodes:
4         if status[node] == 'undiscovered':
5             bfs(graph, node, status)
```

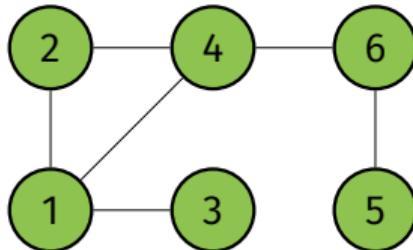
Exercise

How many times is each node added to the queue in a BFS of the graph below?



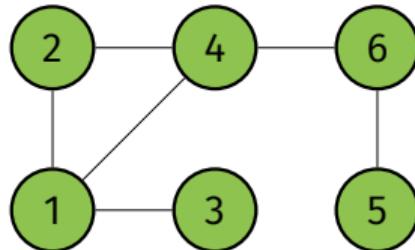
Exercise

How many times is each edge “explored” in a BFS of the graph below?



Exercise

How many times is each edge “explored” in a BFS of the *directed* graph below?



Key Properties of full_bfs

- ▶ Each node added to queue **exactly once**.
- ▶ Each edge is explored **exactly**:
 - ▶ **once** if graph is **directed**.
 - ▶ **twice** if graph is **undirected**.

Time Complexity of full_bfs

- ▶ Analyzing full_bfs is easier than analyzing bfs.
 - ▶ full_bfs visits all nodes, no matter the graph.
- ▶ Result will be **upper bound** on time complexity of bfs.
- ▶ We'll use an **aggregate analysis**.

BFS

```
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u, v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
    status[u] = 'visited'
```



Time Complexity

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered':
            bfs(graph, node, status)

def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Time Complexity of Full BFS

- ▶ $\Theta(V + E)$
- ▶ If $|V| > |E|$: $\Theta(V)$
- ▶ If $|V| < |E|$: $\Theta(E)$
- ▶ Namely, if graph is **complete**: $\Theta(V^2)$.
- ▶ Namely, if graph is **very sparse**: $\Theta(V)$.

Notational Note

- ▶ We'll often write $\Theta(V + E)$ instead of $\Theta(|V| + |E|)$.
- ▶ You can use whichever.

Next Time

- ▶ Finding **shortest paths** using BFS.

DSC 40B

Theoretical Foundations II

Lecture 12 | Part 1

Warmup: Aggregate Analysis

Time Complexity

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered':
            bfs(graph, node, status)

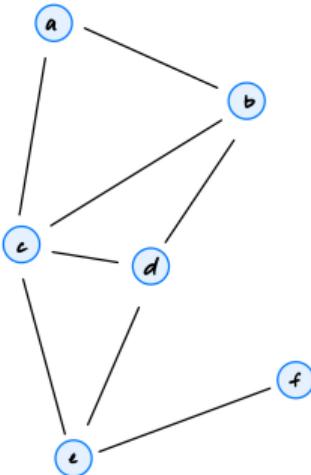
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Exercise

What is printed if we run a BFS starting at a?



```
...
while pending:
    u = pending.popleft()
    print(f'Popped {u}')
    for v in graph.neighbors(u):
        print(f'Exploring edge ({u}, {v})')
        # explore edge (u,v)
    ...

```

Answer

Popping a

Exploring edge (a, b)

Exploring edge (a, c)

Popping b

Exploring edge (b, a)

Exploring edge (b, c)

Exploring edge (b, d)

Popping c

Exploring edge (c, a)

Exploring edge (c, b)

Exploring edge (c, d)

Exploring edge (c, e)

Popping d

Exploring edge (d, b)

Exploring edge (d, c)

Exploring edge (d, e)

Popping e

Exploring edge (e, c)

Exploring edge (e, d)

Exploring edge (e, f)

Popping f

Exploring edge (f, e)

Aggregate Analysis

- ▶ During any one call to bfs:
 - ▶ Number of printed nodes: ?
 - ▶ Number of printed edges: ?
- ▶ In **aggregate** (over all calls):
 - ▶ Number of printed nodes: *exactly* $|V|$
 - ▶ Number of printed edges: *exactly* $2|E|$

Time Complexity

- ▶ Full BFS takes $\Theta(V + E)$

Time Complexity

- ▶ Full BFS takes $\Theta(V + E)$
- ▶ Why not just $\Theta(E)$?
- ▶ $\Theta(V + E)$ works *for all graphs*.
 - ▶ If we know more about the number of edges, we might be able to simplify.
 - ▶ E.g., if the graph is **complete**, $E = \Theta(V^2)$, so time complexity is $\Theta(V + V^2) = \Theta(V^2)$.

DSC 40B

Theoretical Foundations II

Lecture 12 | Part 2

Shortest Paths



Recall

- ▶ The **length** of a path is
 $(\# \text{ of nodes}) - 1$

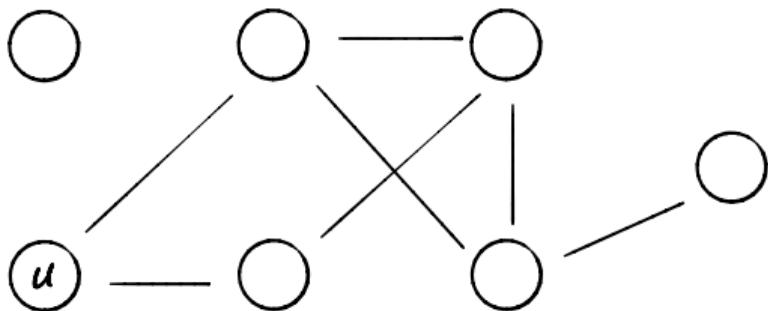
Definitions

- ▶ A **shortest path** between u and v is a path between u and v with smallest possible length.
 - ▶ There may be several, or none at all.
- ▶ The **shortest path distance** is the length of a shortest path.
 - ▶ Convention: ∞ if no path exists.
 - ▶ “the distance between u and v ” means spd.

Today: Shortest Paths

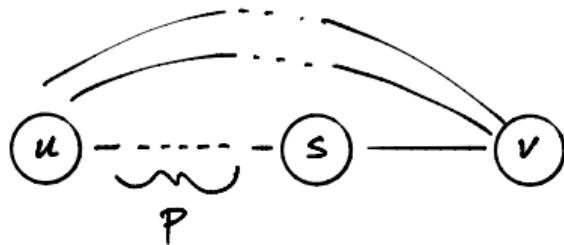
- ▶ **Given:** directed/undirected graph G , source u
- ▶ **Goal:** find shortest path from u to every other node

Example



Key Property

- ▶ A shortest path of length k is composed of:
 - ▶ A **shortest path** of length $k - 1$.
 - ▶ Plus one edge.



Algorithm Idea

- ▶ Find all nodes distance 1 from source.
- ▶ Use these to find all nodes distance 2 from source.
- ▶ Use these to find all nodes distance 3 from source.
- ▶ ...

It turns out...

...this is exactly what BFS does.

DSC 40B

Theoretical Foundations II

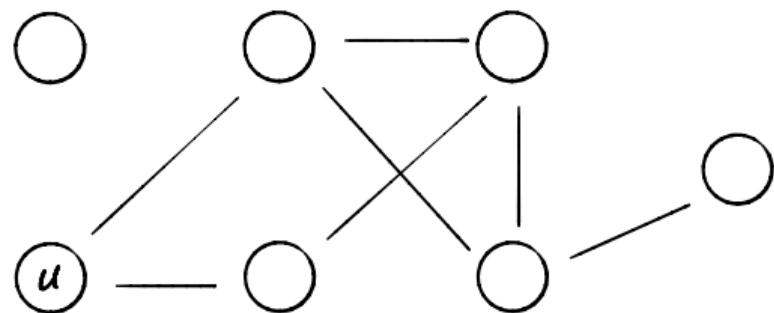
Lecture 12 | Part 3

BFS for Shortest Paths

Key Property of BFS

- ▶ For any $k \geq 1$ you choose:
- ▶ All nodes distance $k - 1$ from source are added to the queue before any node of distance k .
- ▶ Therefore, nodes are “processed” (popped from queue) in order of distance from source.

Example



Discovering Shortest Paths

- ▶ We “discover” shortest paths when we pop a node from queue and look at its neighbors.
- ▶ But the neighbor’s status matters!

Consider This

- ▶ We pop a node s .
- ▶ It has a neighbor v whose status is **undiscovered**.
- ▶ We've discovered a **shortest path** to v through s !

Consider This

- ▶ We pop a node s .
- ▶ It has a neighbor v whose status is **pending** or **visited**.
- ▶ We already have a shortest path to v .

Modifying BFS

- ▶ Use BFS “framework”.
- ▶ Return dictionary of **search predecessors**.
 - ▶ If v is discovered while visiting u , we say that u is the **BFS predecessor** of v .
 - ▶ This encodes the shortest paths.
- ▶ Also return dictionary of shortest path distances.

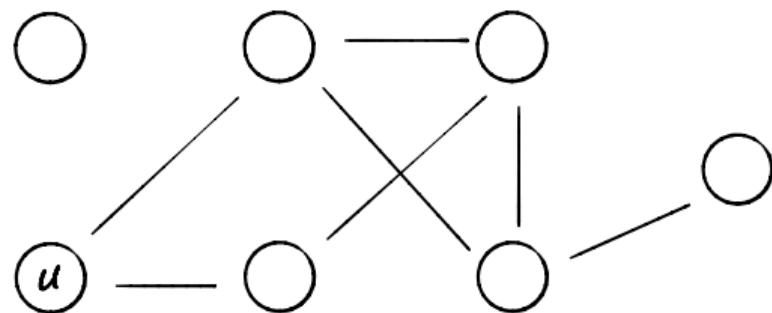
```
def bfs_shortest_paths(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                # append to right
                pending.append(v)
        status[u] = 'visited'

    return predecessor, distance
```

Example



DSC 40B

Theoretical Foundations II

Lecture 12 | Part 4

BFS Trees

Result of BFS

- ▶ Each node reachable from source has a single BFS predecessor.
 - ▶ Except for the source itself.
- ▶ The result is a **tree** (or forest).

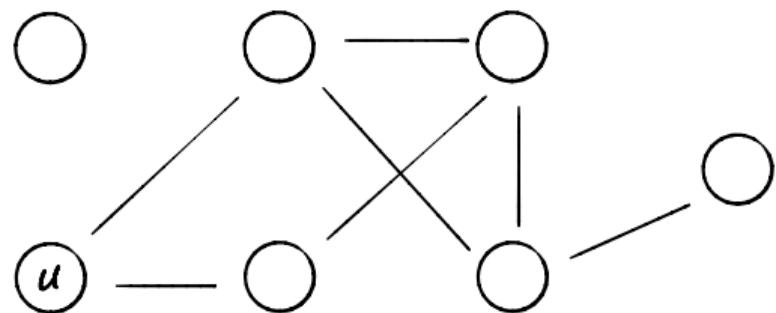
Trees

- ▶ A (free) **tree** is an undirected graph $T = (V, E)$ such that T is connected and $|E| = |V| - 1$.
- ▶ A **forest** is graph in which each connected component is a tree.

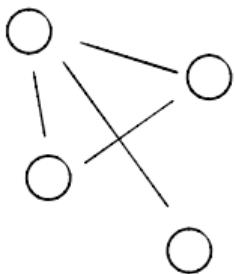
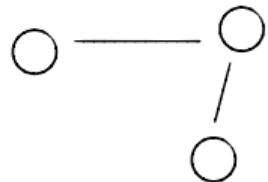
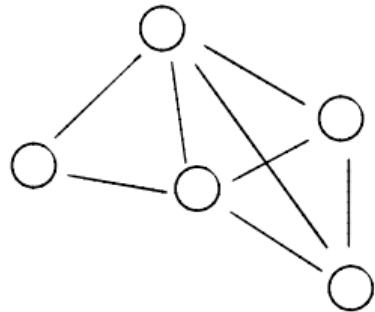
BFS Trees (Forests)

- ▶ If the input is connected, BFS produces a **tree**.
- ▶ If the input is not connected, BFS produces a **forest**.

Example



Example



BFS Trees

- ▶ BFS trees and forests encode shortest path distances.

DSC 40B

Theoretical Foundations II

Lecture 13 | Part 1

Depth First Search

Visiting the Next Node

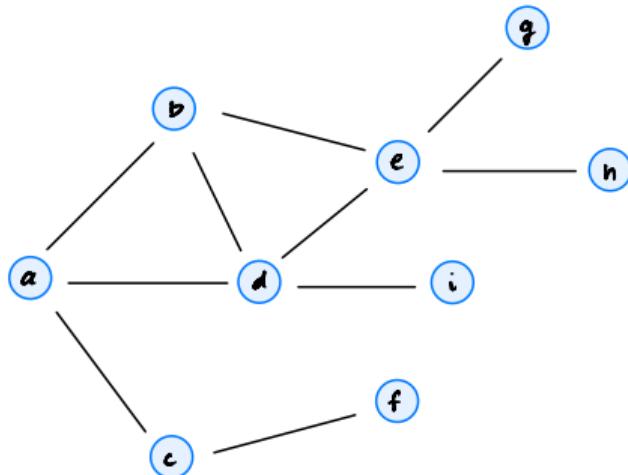
- ▶ Which node do we process next in a search?
- ▶ BFS: the **oldest** pending node.
- ▶ DFS (today): the **newest** pending node.
 - ▶ Naturally recursive.

```
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Example

```
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    ...
    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

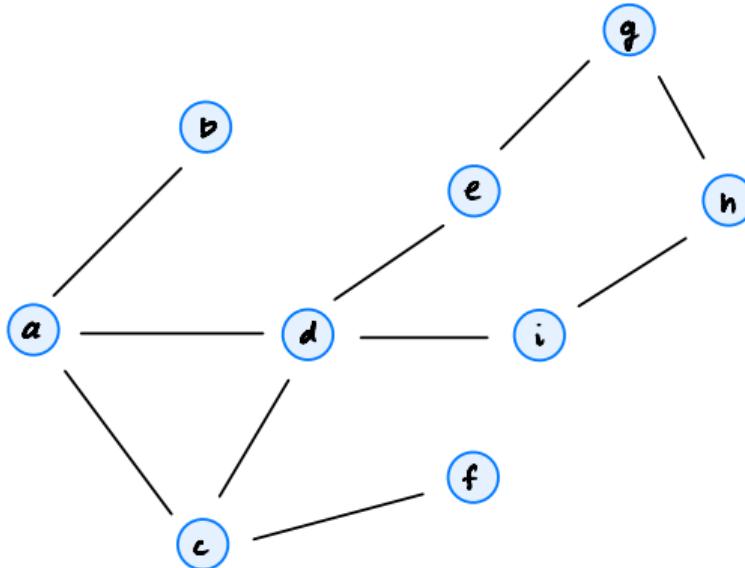


Main Idea

We'll see that the structure of the nested function calls gives us useful information about the graph's structure.

Exercise

Write the nested function calls for a DFS on the graph below.



Full DFS

- ▶ DFS will visit all nodes reachable from source.
- ▶ To visit all nodes in graph, need **full DFS**.

```
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)
```

```
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)

def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Time Complexity

- ▶ In a full DFS:
 - ▶ dfs called on each node exactly once.
 - ▶ Like BFS, each edge is explored exactly:
 - ▶ once if directed
 - ▶ twice if undirected
- ▶ Time: $\Theta(V + E)$, just like BFS.

DSC 40B

Theoretical Foundations II

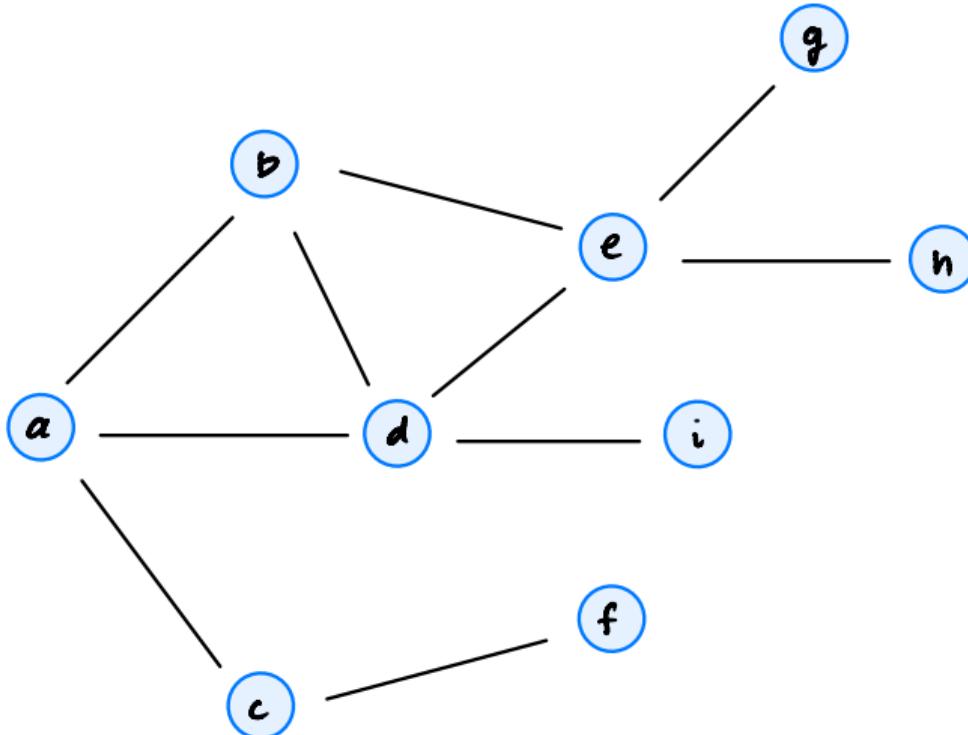
Lecture 13 | Part 2

Nesting Properties of DFS

Key Property of DFS (Informal)

- ▶ Suppose v is reachable from u , and v is **undiscovered** at the time of $\text{dfs}(u)$.
- ▶ If there is a path of undiscovered nodes from u to v at the time of $\text{dfs}(u)$:
 - ▶ $\text{dfs}(v)$ will be run.
 - ▶ v will be marked as **visited**.

Example



Start and Finish Times

- ▶ Keep a running clock (an integer).
- ▶ For each node, record
 - ▶ **Start time**: time when marked pending
 - ▶ **Finish time**: time when marked visited
- ▶ Increment clock whenever node is marked pending/visited

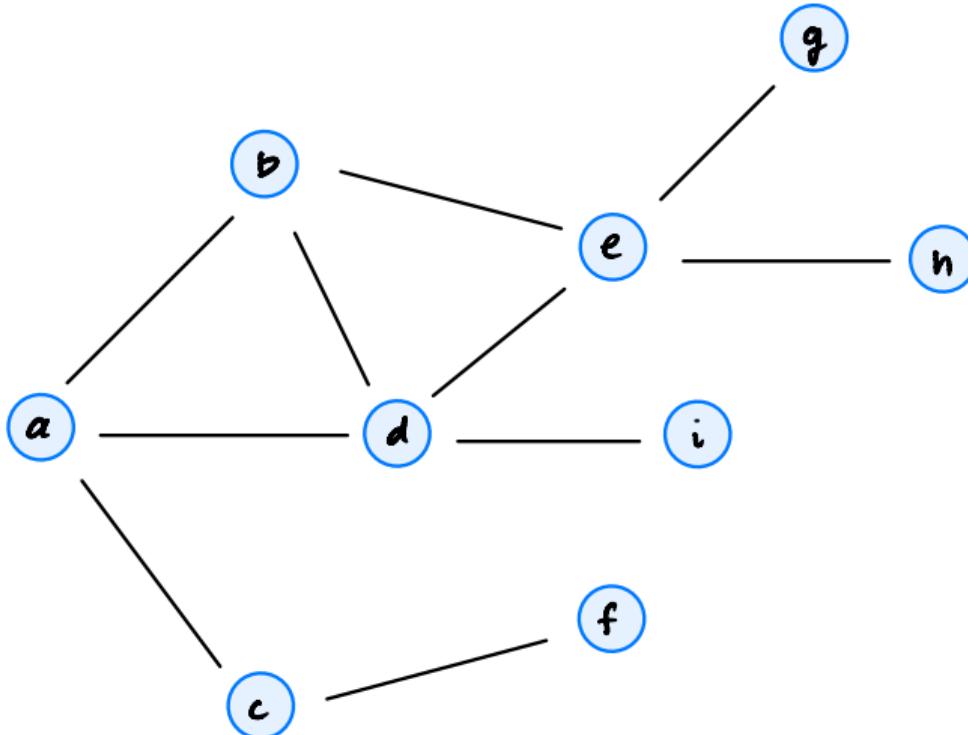
```
from dataclasses import dataclass

@dataclass
class Times:
    clock: int
    start: dict
    finish: dict

def full_dfs_times(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}
    times = Times(clock=0, start={}, finish={})
    for u in graph.nodes:
        if status[u] == 'undiscovered':
            dfs_times(graph, u, status, times)
    return times, predecessor

def dfs_times(graph, u, status, predecessor, times):
    times.clock += 1
    times.start[u] = times.clock
    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            predecessor[v] = u
            dfs_times(graph, v, status, times)
    status[u] = 'visited'
    times.clock += 1
    times.finish[u] = times.clock
```

Example



Key Property

- ▶ Take any two nodes u and v ($u \neq v$).
- ▶ If v is *started* between $\text{start}[u]$ and $\text{finish}[u]$,
then v is *finished* between $\text{start}[u]$ and
 $\text{finish}[u]$.

Key Property

- ▶ Take any two nodes u and v ($u \neq v$).
- ▶ Assume for simplicity that $\text{start}[u] < \text{start}[v]$.
- ▶ Exactly one of these is true:
 - ▶ $\text{start}[u] < \text{start}[v] < \text{finish}[v] < \text{finish}[u]$
 - ▶ $\text{start}[u] < \text{finish}[u] < \text{start}[v] < \text{finish}[v]$

DSC 40B

Theoretical Foundations II

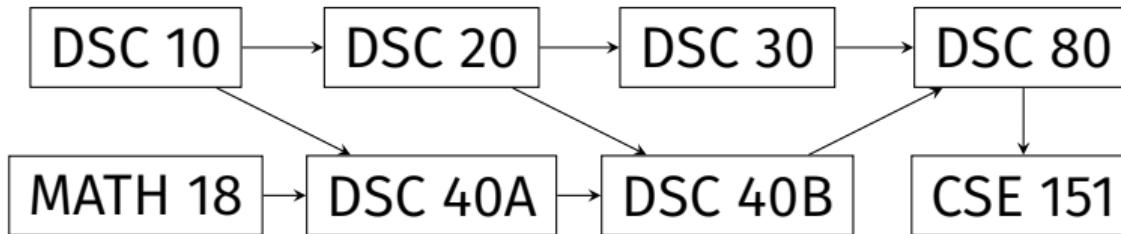
Lecture 13 | Part 3

Topological Sort

Applications of DFS

- ▶ Is node v reachable from node u ?
- ▶ Is the graph connected?
- ▶ How many connected components?
- ▶ What is the shortest path between u and v ? **No.**

Prerequisite Graphs



Goal: find order in which to take classes satisfying prerequisites.

Directed Acyclic Graphs

- ▶ A **directed cycle** is a path from a node to itself with at least one edge.
- ▶ A **directed acyclic graph (DAG)** is a directed graph with no directed cycles.

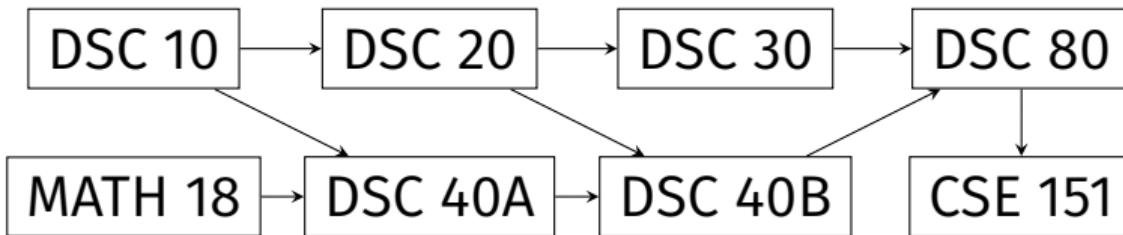
Example

- ▶ Prerequisite graphs are DAGs.
 - ▶ Or at least, they should be!

Topological Sorts

- ▶ **Given:** a DAG, $G = (V, E)$.
- ▶ **Compute:** an ordering of V such that if $(u, v) \in E$, then u comes before v in the ordering
- ▶ This is called a **topological sort** of G .

Example



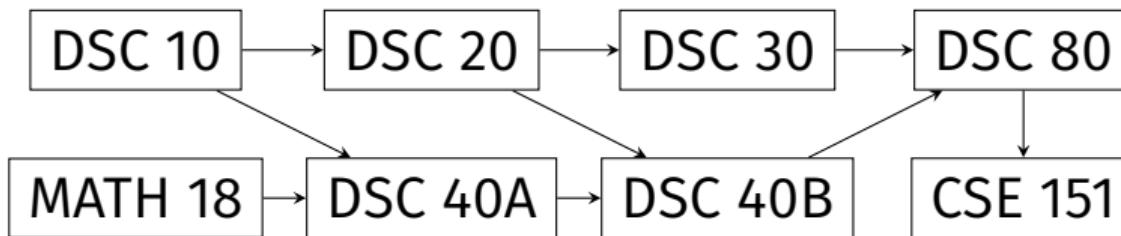
MATH 18, DSC 10, DSC 40A, DSC 20, DSC 40B, DSC 30, DSC 80, CSE 151

Key Property

- ▶ Take any two nodes u and v ($u \neq v$).
- ▶ Assume the graph is a DAG.
- ▶ **Example:** If v is reachable from u , then $\text{finish}[v] < \text{finish}[u]$.

Exercise

Compute start and finish times using DSC 10 as the source.



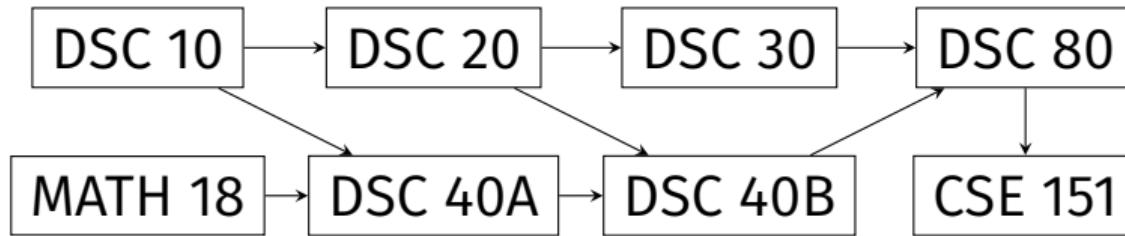
An Algorithm

- ▶ **Recall:** If v is reachable from u , then $\text{finish}[v] \leq \text{finish}[u]$.
- ▶ If v is reachable from u , u should come before v .
- ▶ Idea: nodes with later finish times should come first.

Algorithm

- ▶ To find a topological sort (if it exists):
 - ▶ Compute times with Full DFS.
 - ▶ Sort in **descending** order by finish time.
- ▶ Time complexity:

Example



Note

- ▶ There can be many valid topological sorts!

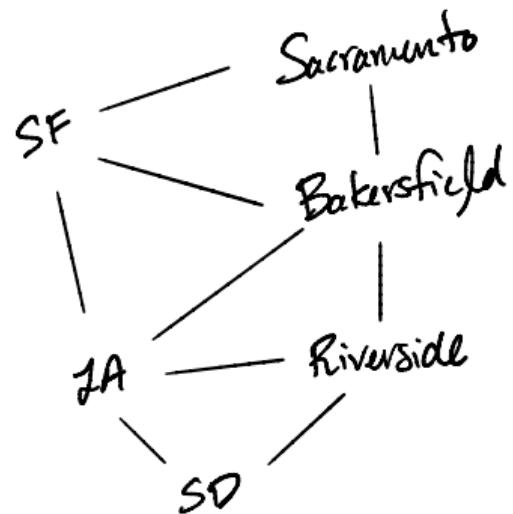
DSC 40B

Theoretical Foundations II

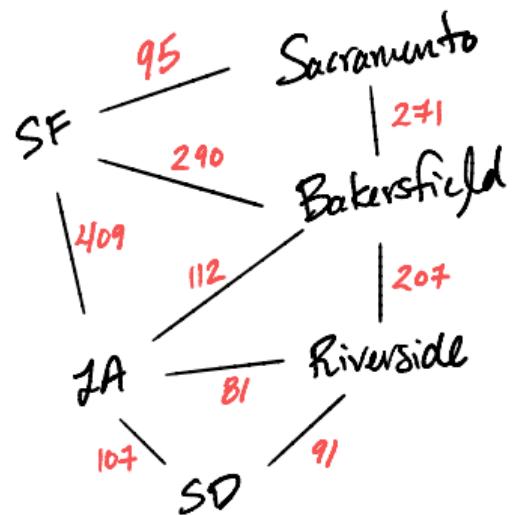
Lecture 14 | Part 1

Shortest Paths in Weighted Graphs

Google Maps



Google Maps



Weighted Graphs

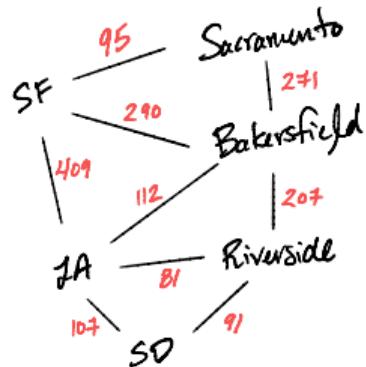
An **edge weighted graph** $G = (V, E, \omega)$ is a triple where (V, E) is a graph and $\omega : E \rightarrow \mathbb{R}$ maps each edge to a **weight**.

- ▶ Can be directed or undirected.
- ▶ In general, weights can be positive, negative, zero.
- ▶ Many uses, such as representing **metric spaces**.

Path Lengths

The **length** of a path in a **weighted graph** (usually) refers to the total weight of all edges in the path.

Example: (SD, Riverside, Bakersfield, SF)

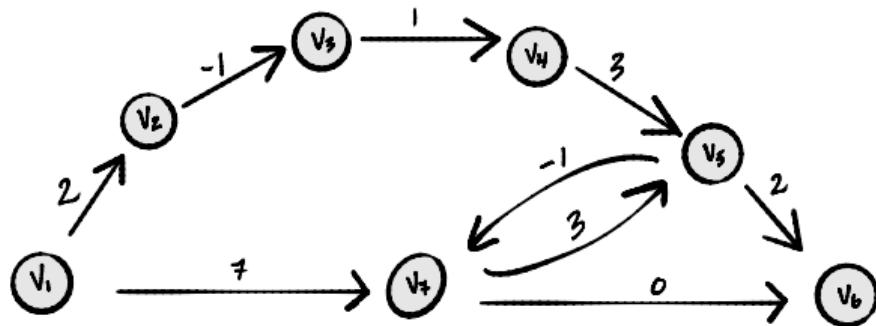


Shortest Paths

- ▶ A **shortest path** between u and v is a path between u and v with minimum length.
 - ▶ In other words, minimum total weight.

Example

What is the shortest path from v_1 to v_6 ?



Path:

Length:

Today (and next time)

How do we find shortest paths in weighted graphs?

Idea #0

- ▶ Does BFS work?
 - ▶ **No**, not really. Only if all weights are the same.
- ▶ Can we “convert” a weighted graph to an unweighted one?

Idea #0



Idea #0

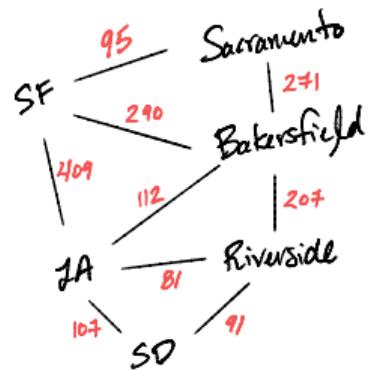


Idea #0

- ▶ Step 1: “Convert” weighted graph to unweighted one with dummy nodes.
- ▶ Step 2: Call BFS on this new graph.

Idea #0

- ▶ **Very inefficient** for large weights.



- ▶ What if edge weights are floats, or negative?

Ideas #1 and #2

- ▶ We'll look at two algorithms: **Bellman-Ford** and **Dijkstra's**.

INPUT: weighted graph, source vertex s .

OUTPUT: shortest paths from s to every other node.

- ▶ Both work by:
 - ▶ keeping track of shortest known path (**estimates**).
 - ▶ iteratively **updating** these until they're correct.

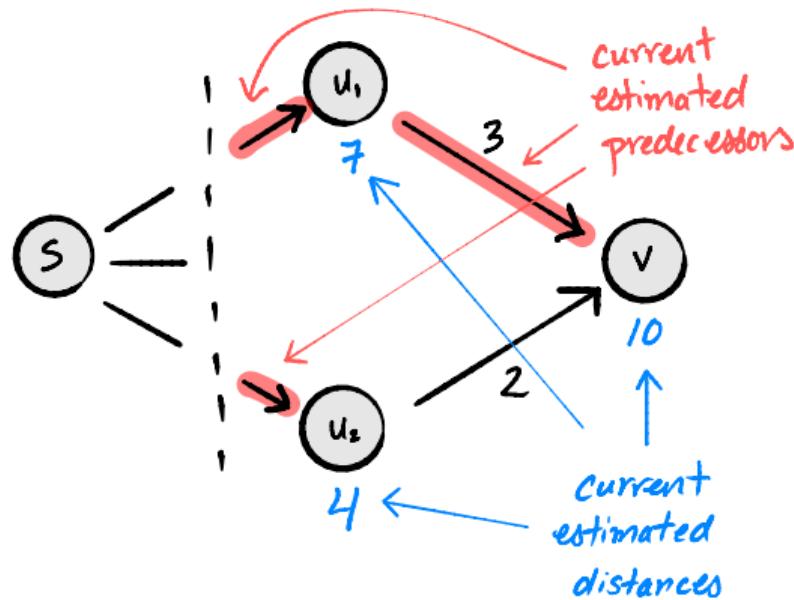
Shortest Path Estimates

- ▶ B-F and Dijkstra's keep track of the shortest paths found *so far*; we call these the **estimated shortest paths**.
- ▶ For each node u , remember u 's:
 - ▶ predecessor in estimated shortest path;
 - ▶ distance from source s in estimated shortest path.
- ▶ **Key:** estimated distance will always be \geq actual distance.

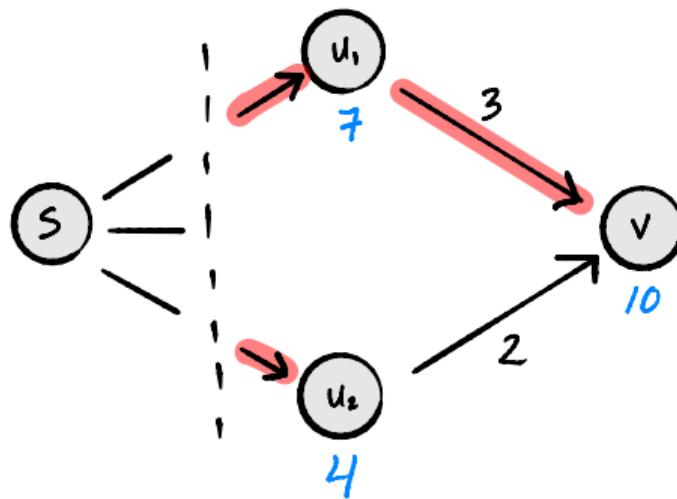
Updates

- ▶ Both algorithms work by iteratively updating their estimates.
- ▶ On each iteration, consider a new edge (u, v) .
Ask: is the best known shortest path from
 $\text{source} \rightarrow \dots \rightarrow u \rightarrow v$
shorter than the best known shortest path from
 $\text{source} \rightarrow \dots \rightarrow \text{predecessor}[v] \rightarrow v?$
- ▶ If it is, we have discovered a shorter path to v .

Example: Updating (u_2, v) :

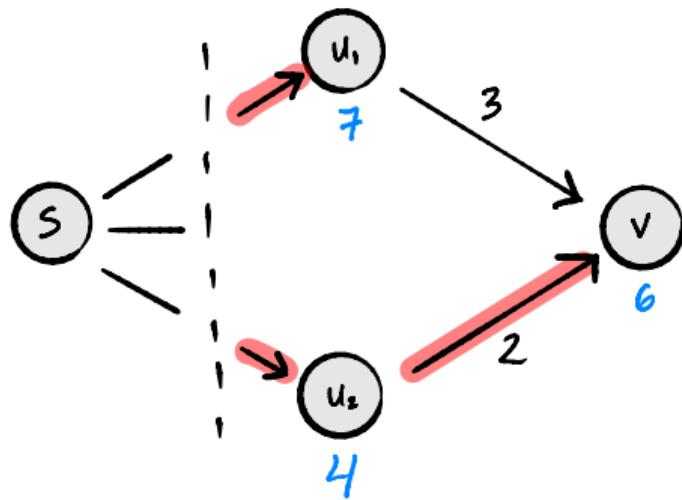


Example: Updating (u_2, v) :



$$\begin{aligned}\text{estimated length of } s \rightarrow u_2 \rightarrow v &= (\text{estimated length of } s \rightarrow u_2) + \omega(u_2, v) \\ &= 4 + 2 = 6 < 10\end{aligned}$$

Example: After Updating (u_2, v) :



A shorter path has been found.

Updating, in Code

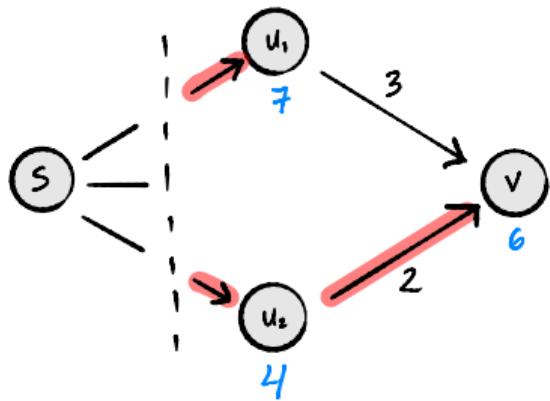
- ▶ Let:
 - ▶ `est` be a dictionary of estimated shortest path distances.
 - ▶ `predecessor` be a dictionary of estimated shortest path predecessors.
 - ▶ `weights` be a function which returns edge weights.

Updating, in Code

```
def update(u, v, weights, est, predecessor):
    """Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```

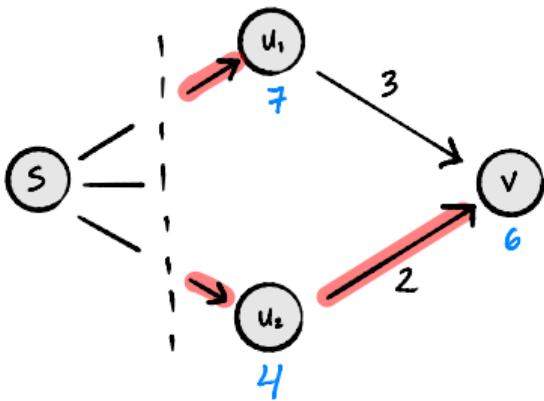
- ▶ Time complexity: _____

When does an update discover a shortest path?



- ▶ Suppose updating (u_2, v) finds a shorter path to v .
- ▶ True or False: the actual shortest path must go through u_2 .

When does an update discover a shortest path?



- ▶ Suppose updating (u_2, v) finds a shorter path to v .
- ▶ True or False: the actual shortest path must go through u_2 .
- ▶ **False:** we might later discover a better path to u_1 .

When does an update discover the shortest path?

- ▶ Let (u, v) be an edge.
- ▶ Suppose:
 - ▶ the actual shortest path to u has been found;
 - ▶ the actual shortest path to v goes through (u, v) .
- ▶ Then after updating (u, v) , the estimated shortest path to v is correct.

DSC 40B

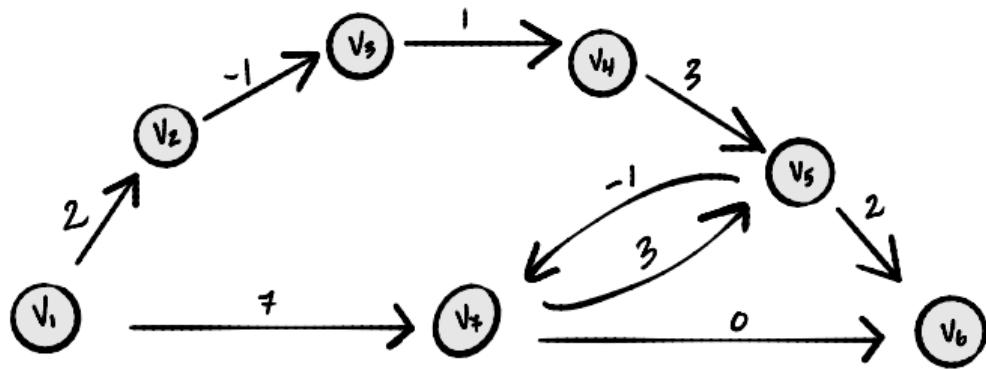
Theoretical Foundations II

Lecture 14 | Part 2

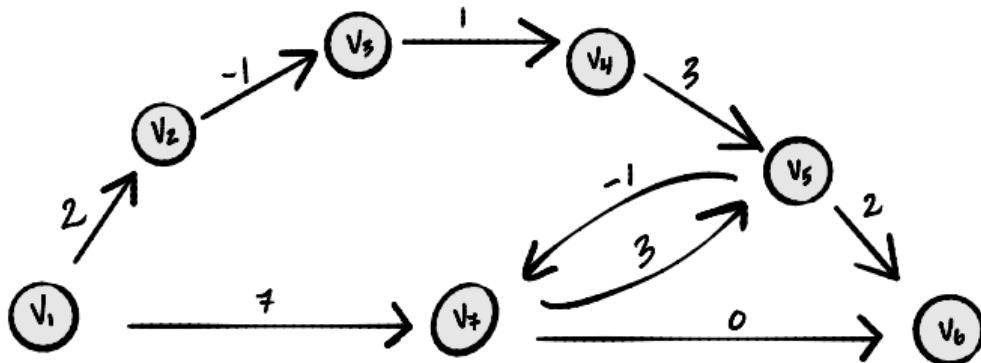
Bellman-Ford

Intuition

- ▶ Shortest paths that have many edges are “harder” to discover.
 - ▶ May require many updates.
- ▶ Shortest paths that have few edges are “easier” to discover.
- ▶ Once we’ve discovered all of the shortest paths with few edges, it makes it easier to discover the shortest paths with more edges.

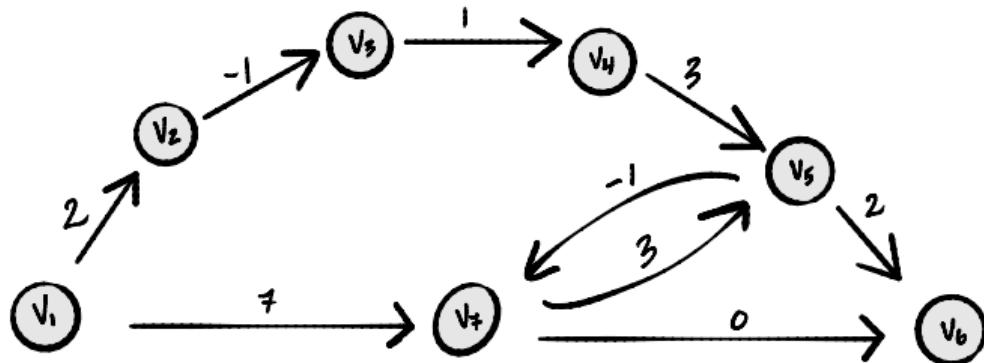


Updating All Edges



- ▶ Suppose we update all of the edges, one by one.
- ▶ Then all nodes whose shortest path from s has only **one** edge are **guaranteed** to be estimated correctly.

Updating All Edges



- ▶ Suppose we update all of the edges again.
- ▶ Then all nodes whose shortest path from s has at most **two** edges are **guaranteed** to be estimated correctly.

Loop Invariant

- ▶ One iteration: update all edges in arbitrary order.
- ▶ Loop invariant: After α iterations, all nodes whose shortest path from s has $\leq \alpha$ edges are guaranteed to be estimated correctly.

The Bellman-Ford Algorithm

```
def bellman_ford(graph, weights, source):
    """Assume graph is directed."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(?):
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)

    return est, predecessor
```

Bellman-Ford

- ▶ Claim: each node must have a shortest path which is simple¹.
- ▶ The most edges a simple path can have is $|V| - 1$
- ▶ Idea of Bellman-Ford: iteratively update all edges, repeat $|V| - 1$ times.

¹Edge case: cycles of weight zero.

The Bellman-Ford Algorithm

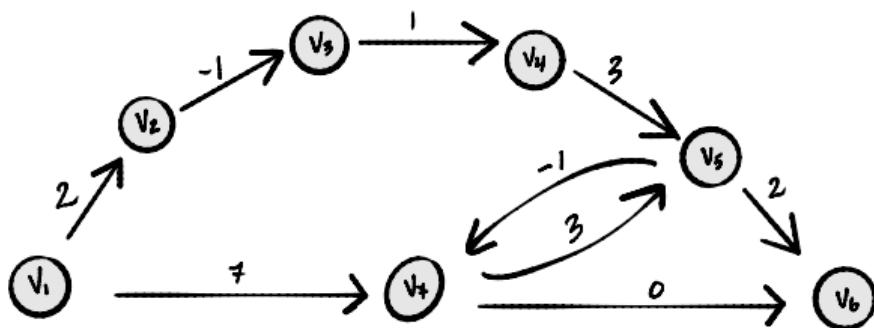
```
def bellman_ford(graph, weights, source):
    """Assume graph is directed."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes) - 1):
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)

    return est, predecessor
```

Example

Suppose `graph.edges` returns edges in following order:

$$\begin{aligned} & (v_3, v_4), (v_1, v_2), (v_2, v_3), (v_7, v_6), (v_5, v_7), \\ & (v_7, v_5), (v_4, v_5), (v_5, v_6), (v_1, v_7) \end{aligned}$$


Time Complexity

```
def bellman_ford(graph, weights, source):
    """Assume graph is directed."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes) - 1):
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)

    return est, predecessor
```

- ▶ Setup: _____ time
- ▶ Each update takes _____ time
- ▶ There are exactly _____ updates
- ▶ Total time complexity: _____

DSC 40B

Theoretical Foundations II

Lecture 14 | Part 3

Early Stopping and Negative Cycles

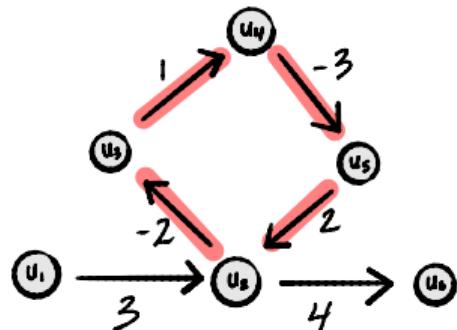
Early Stopping

- ▶ B-F may not need to run for $|V| - 1$ iterations.
- ▶ If no predecessors change, we can break:

```
def bellman_ford(graph, weights, source):  
    """Early stopping version."""  
    est = {node: float('inf') for node in graph.nodes}  
    est[source] = 0  
    predecessor = {node: None for node in graph.nodes}  
  
    for i in range(len(graph.nodes) - 1):  
        any_changes = False  
        for (u, v) in graph.edges:  
            changed = update(u, v, weights, est, predecessor)  
            any_changes = changed or any_changes  
        if not any_changes:  
            break  
    return est, predecessor
```

Negative Cycles

- ▶ A **negative cycle** is a cycle whose total edge weight is negative:



- ▶ If a graph has a negative cycle, (some) shortest paths are **not well defined**.

Detecting Negative Cycles

- ▶ If graph **does not have** negative cycles, estimated distances eventually stop changing (after at most $|V| - 1$ iterations).
- ▶ If graph **has** negative cycles, estimated distances always decrease.
- ▶ To detect them: run a $|V|$ th iteration; if distances change, a negative cycle exists.

Detecting Negative Cycles

```
def bellman_ford(graph, weights, source):
    """Early stopping version, detects negative cycles."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes)):
        any_changes = False
        for u, v in graph.edges:
            changed = update(u, v, weights, est, predecessor)
            any_changes = changed or any_changes
        if not any_changes:
            break
    # this will be True if negative cycles exist
    contains_negative_cycles = any_changes
    return est, predecessor, contains_negative_cycles
```

DSC 40B

Theoretical Foundations II

Lecture 15 | Part 1

Dijkstra's Algorithm

Shortest Path Algorithms

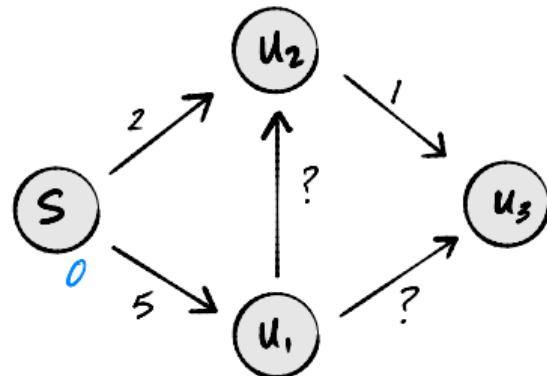
- ▶ **Bellman-Ford** and **Dijkstra's** are shortest path algorithms:
 - INPUT: weighted graph, source vertex s .
 - OUTPUT: shortest paths from s to every other node.
- ▶ Both work by:
 - ▶ keeping estimates of shortest path distances;
 - ▶ iteratively **updating** estimates until they're correct.

Shortest Path Algorithms

- ▶ We saw Bellman-Ford last time; takes time $\Theta(VE)$.
- ▶ Dijkstra's will be faster, but can't handle negative weights.

Dijkstra's Algorithm

- ▶ On every iteration, Bellman-Ford updates all edges – many don't need to be updated.
- ▶ If we **assume** all edge weights are positive, we can rule out some paths immediately:



Dijkstra's Idea

- ▶ Keep track of set C of “correct” nodes.
 - ▶ Nodes whose distance estimate is correct.
- ▶ At every step, add node outside of C with smallest estimated distance; update only its neighbors.
- ▶ A “greedy” algorithm.

Outline of Dijkstra's Algorithm

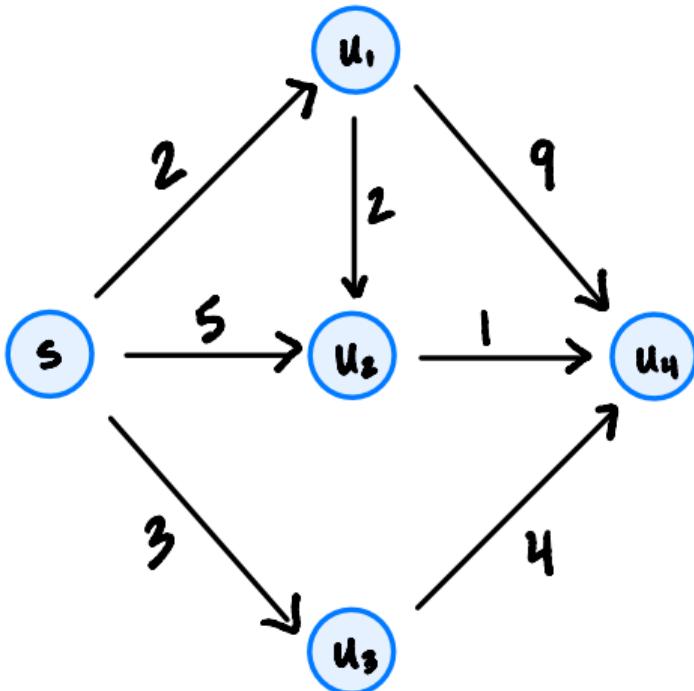
```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)
    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

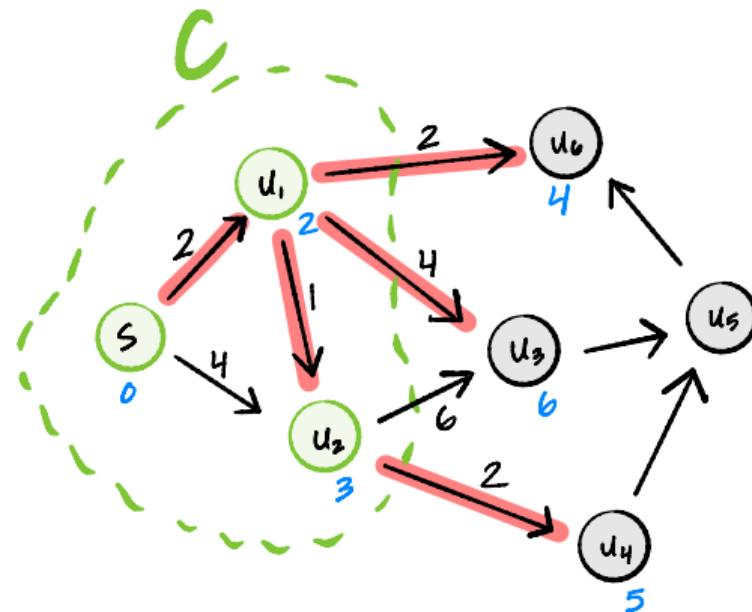
    return est, pred
```

Example



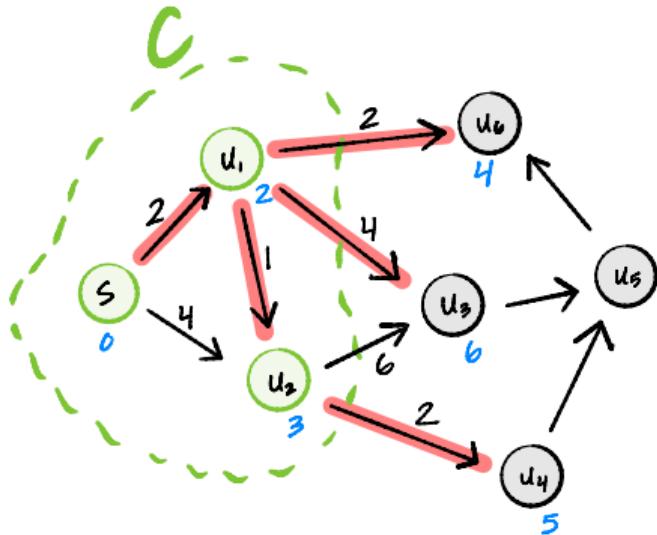
Proof Idea

- Claim: at beginning of any iteration of Dijkstra's, if u is node $\notin C$ with smallest estimated distance, the shortest path to u has been correctly discovered.



Proof Idea

- ▶ Let u be node outside of C for which $\text{est}[u]$ is smallest.
- ▶ We've discovered a path from s to u of length $\text{est}[u]$.
- ▶ Any path from s to u has to exit C somewhere.
- ▶ Any path from s to u will cost at least $\text{est}[u]$ just to exit C .



Exercise

Why do the edge weights need to be positive?
Come up with a simple example graph with some negative edge weights where Dijkstra's fails to compute the correct shortest path.

DSC 40B

Theoretical Foundations II

Lecture 15 | Part 2

Implementation

Outline of Dijkstra's Algorithm

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)
    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

return est, pred
```

Dijkstra's Algorithm: Naïve Implementation

```
1 def dijkstra(graph, weights, source):
2     est = {node: float('inf') for node in graph.nodes}
3     est[source] = 0
4     pred = {node: None for node in graph.nodes}
5
6     outside = set(graph.nodes)
7
8     while outside:
9         # find smallest with linear search
10        u = min(outside, key=est)
11        outside.remove(u)
12        for v in graph.neighbors(u):
13            update(u, v, weights, est, pred)
14
15    return est, pred
```

Priority Queues

- ▶ A **priority queue** allows us to store (key, value) pairs, efficiently return key with lowest value.
- ▶ Suppose we have a priority queue class:
 - ▶ `PriorityQueue(priorities)` will create a priority queue from a dictionary whose values are priorities.
 - ▶ The `.extract_min()` method removes and returns key with smallest value.
 - ▶ The `.change_priority(key, value)` method changes key's value.

Example

```
»> pq = PriorityQueue({  
    'w': 5,  
    'x': 4,  
    'y': 1,  
    'z': 3  
})  
»> pq.extract_min()  
'y'  
»> pq.change_priority('w', 2)  
»> pq.extract_min()
```

Dijkstra's Algorithm: Priority Queue

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

Heaps

- ▶ A priority queue can be implemented using a **heap**.
- ▶ If a **binary min-heap** is used:
 - ▶ `PriorityQueue(est)` takes $\Theta(V)$ time.
 - ▶ `.extract_min()` takes $O(\log V)$ time.
 - ▶ `.change_priority()` takes $O(\log V)$ time.

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

- ▶ Time complexity: _____

DSC 40B

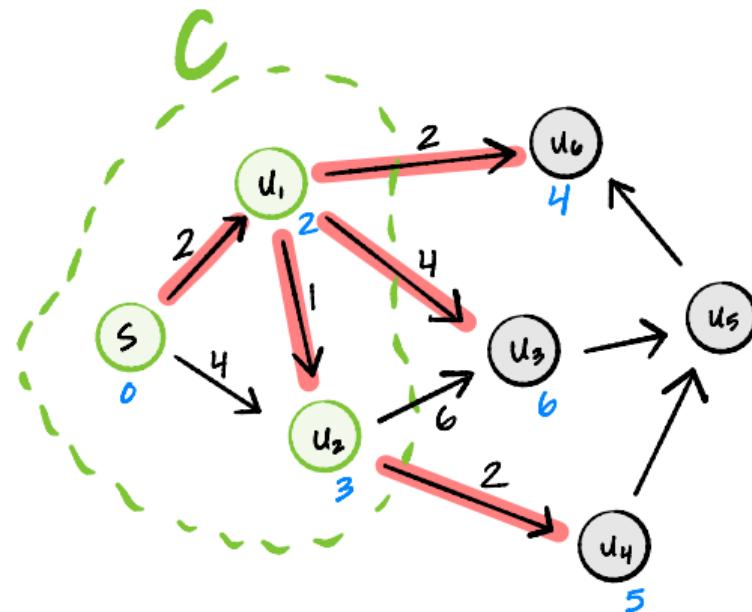
Theoretical Foundations II

Lecture 15 | Part 3

Proof

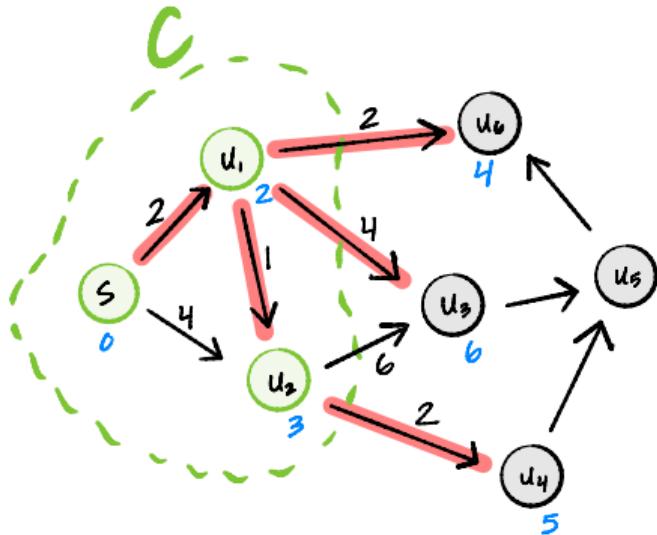
Proof Idea

- Claim: at beginning of any iteration of Dijkstra's, if u is node $\notin C$ with smallest estimated distance, the shortest path to u has been correctly discovered.



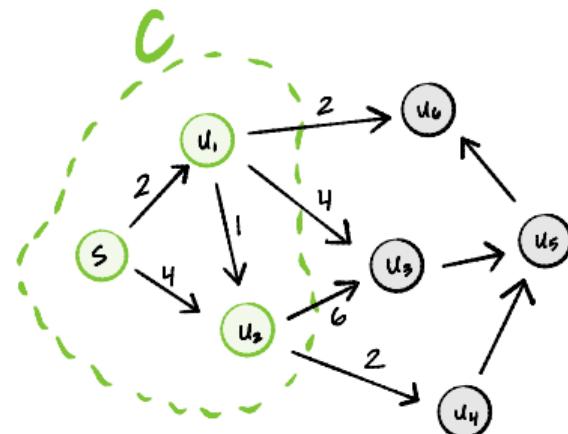
Proof Idea

- ▶ Let u be node outside of C for which $\text{est}[u]$ is smallest.
- ▶ We've discovered a path from s to u of length $\text{est}[u]$.
- ▶ Any path from s to u has to exit C somewhere.
- ▶ Any path from s to u will cost at least $\text{est}[u]$ just to exit C .



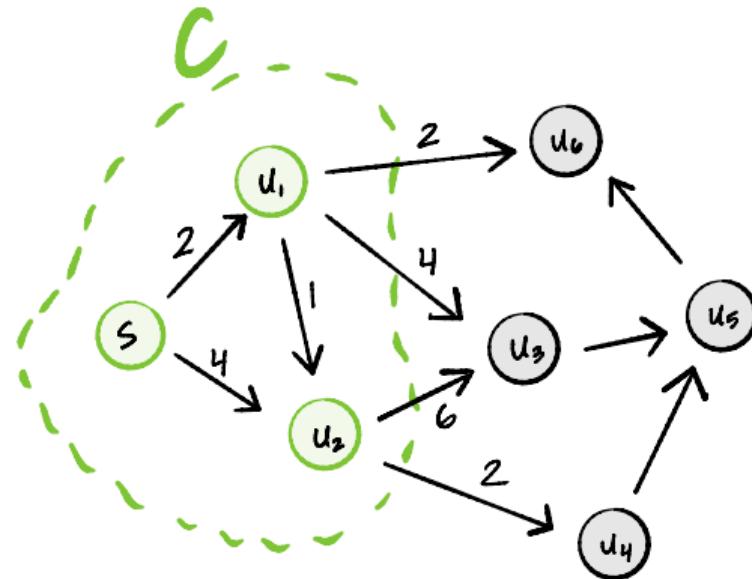
Exit Paths

- ▶ An **exit path from s through C** is a path for which:
 - ▶ the first node is s ;
 - ▶ the last node (a.k.a., the **exit node**) is **not** in C ;
 - ▶ all other nodes **are** in C .
- ▶ Example:



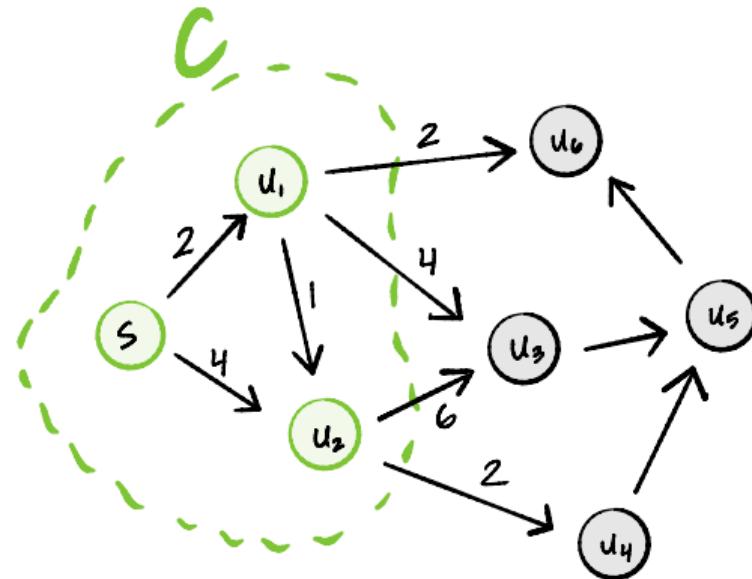
Exit Paths

- ▶ True or False: this is an exit path from s through C .



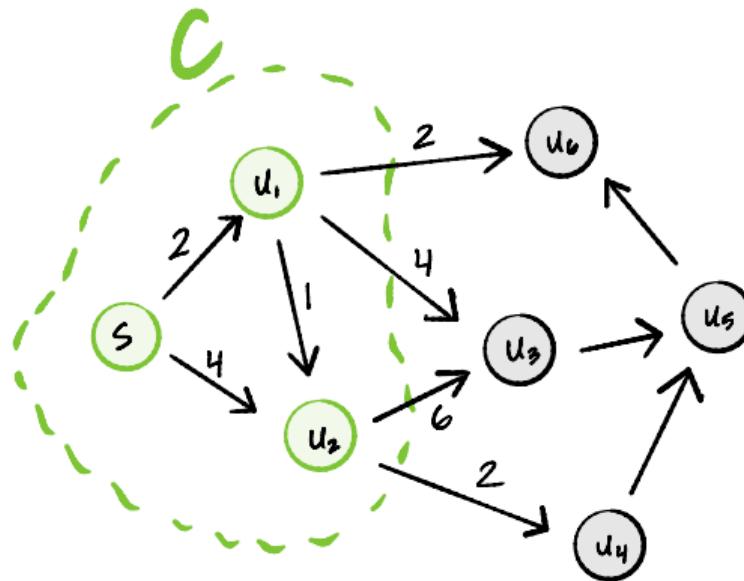
Exit Paths

- ▶ True or False: this is an exit path from s through C .



Path Decomposition

- ▶ Any path from s to a node u outside of C can be broken into two parts:
(an exit path from s) + (path from exit node to u)

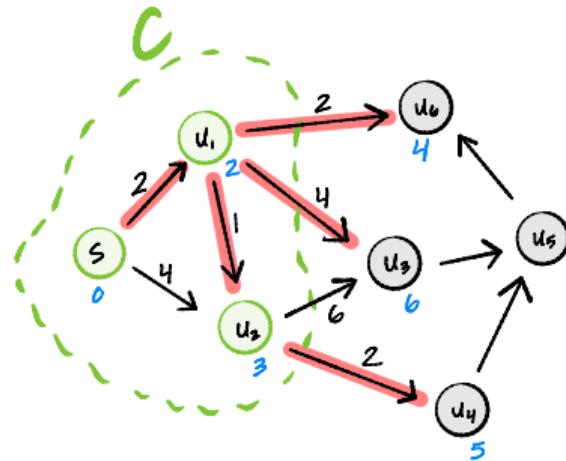


Path Decomposition

- ▶ Consider any path from s to $u \notin C$.
- ▶ Suppose e is the path's exit node.
- ▶ We have:
 - (length of the path)
 - = (length of exit path to e) + (length of path from e to u)
 - \geq (length of shortest exit path to e) + (length of path from e to u)
- ▶ Since edge weights are positive, all path lengths ≥ 0 :
 - \geq (length of shortest exit path to e) + 0

Shortest Exit Paths

- ▶ Example: What is the shortest exit path with exit node u_3 ?



- ▶ If u is outside of C , then the length of the shortest exit path with exit node e is $\text{est}[e]$.

Proof Idea

- ▶ Suppose u is a node outside of C for which $\text{est}[u]$ is smallest.
- ▶ Consider any path from s to u , and let e be the path's exit node.
- ▶ We have:

$$\begin{aligned}& (\text{length of this path from } s \text{ to } u) \\& \geq (\text{length of shortest exit path to } e) + 0 \\& = \text{est}[e] \\& \geq \text{est}[u]\end{aligned}$$

- ▶ That is, any path from s to u has length $\geq \text{est}[u]$.
- ▶ We've already found one with length $\text{est}[u]$; this proves that it is the shortest.

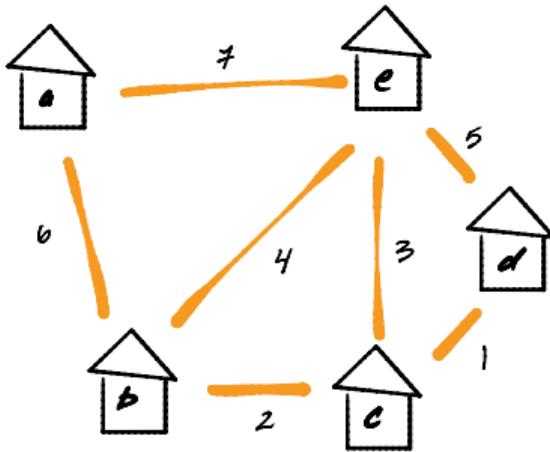
DSC 40B

Theoretical Foundations II

Lecture 16 | Part 1

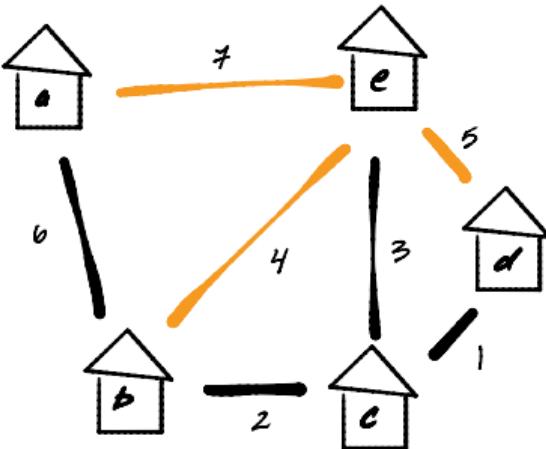
Minimum Spanning Trees

Today's Problem



- ▶ Choose a set of dirt roads to pave so that:
 - ▶ can get between any two buildings only on paved roads;
 - ▶ total cost is minimized.
- ▶ Solution: compute a **minimum spanning tree**.

Today's Problem



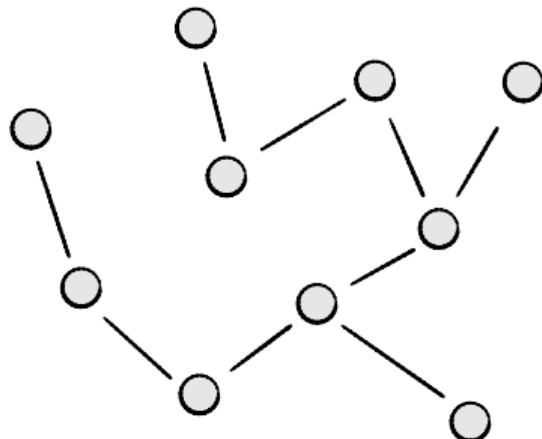
- ▶ Choose a set of dirt roads to pave so that:
 - ▶ can get between any two buildings only on paved roads;
 - ▶ total cost is minimized.
- ▶ Solution: compute a **minimum spanning tree**.

Trees

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ it is acyclic.

Example: a **tree**.

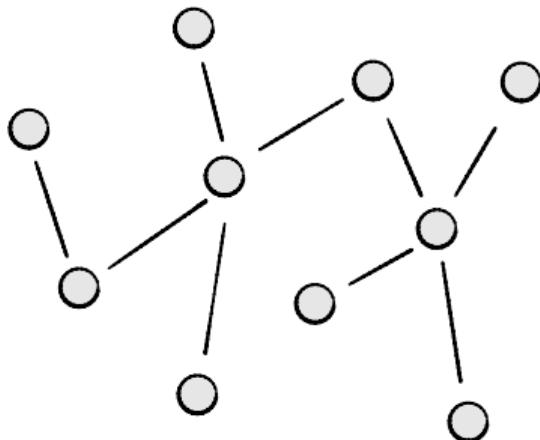


Trees

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ it is acyclic.

Example: a **tree**.

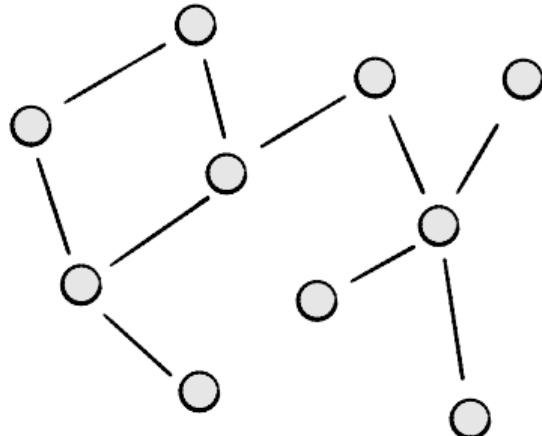


Trees

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ it is acyclic.

Example: **not** a tree.

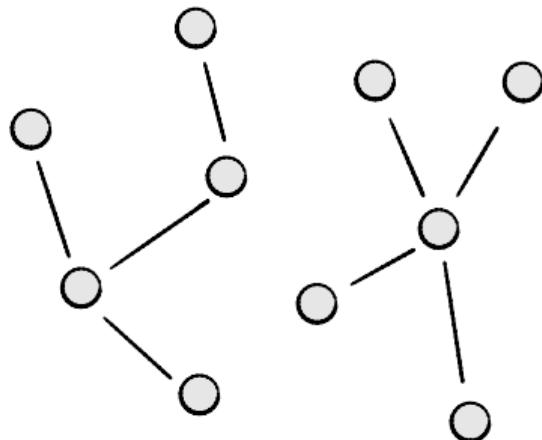


Trees

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ it is acyclic.

Example: **not** a tree.

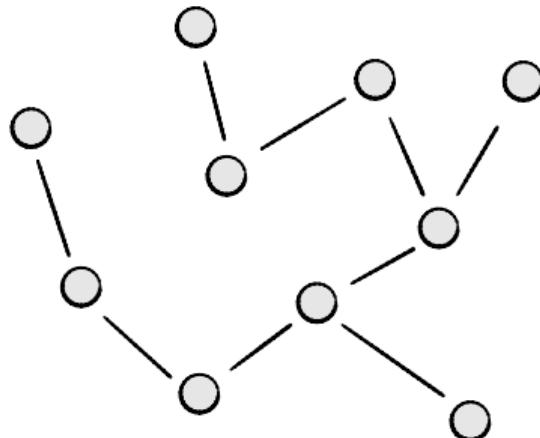


Trees: Equivalent Definition

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ $|E| = |V| - 1$.

Example: a **tree**.

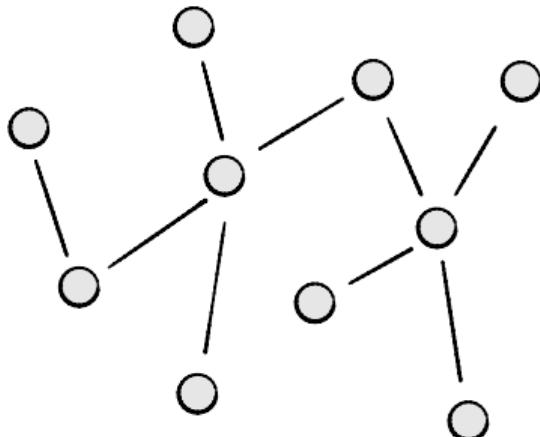


Trees: Equivalent Definition

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ $|E| = |V| - 1$.

Example: a **tree**.

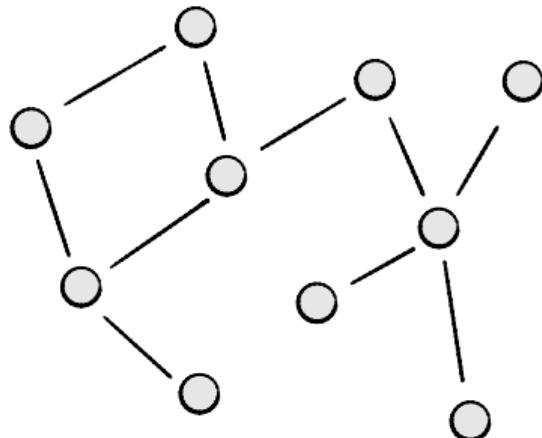


Trees: Equivalent Definition

An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ $|E| = |V| - 1$.

Example: **not** a tree.

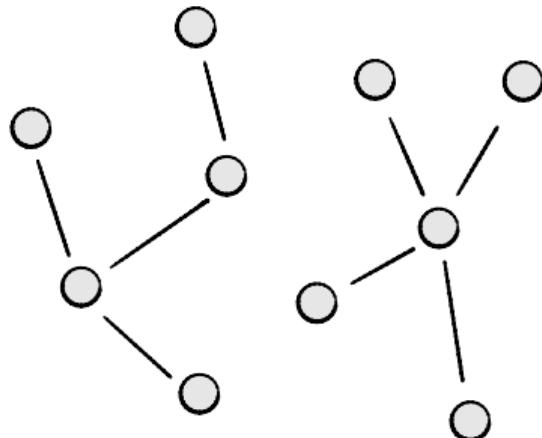


Trees: Equivalent Definition

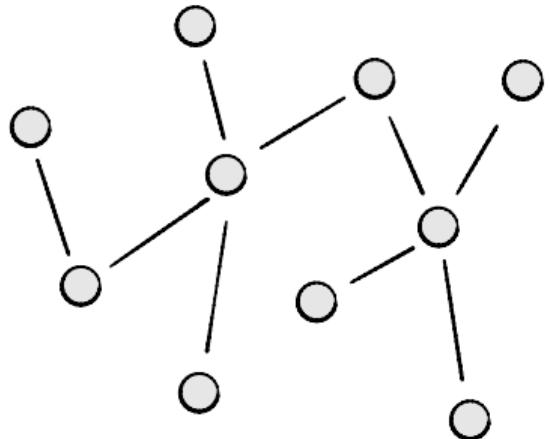
An undirected graph $T = (V, E)$ is a **tree** if

- ▶ it is connected; and
- ▶ $|E| = |V| - 1$.

Example: **not** a tree.



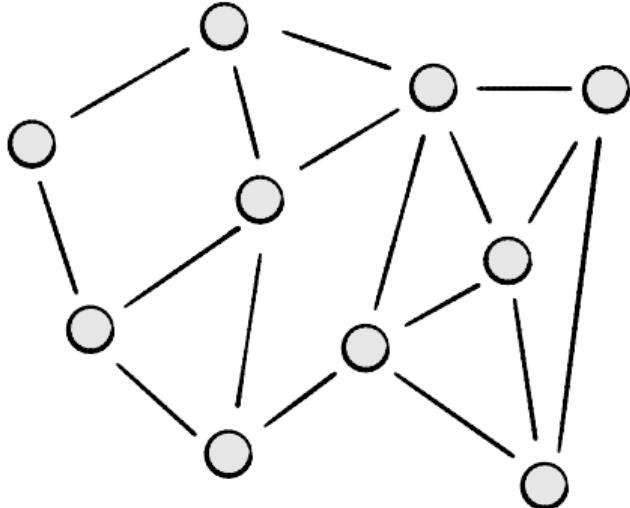
Tree Properties



- ▶ There is a unique simple path between any two nodes in a tree.
- ▶ Adding a new edge to a tree creates a cycle (no longer a tree).
- ▶ Removing an edge from a tree disconnects it (no longer a tree).

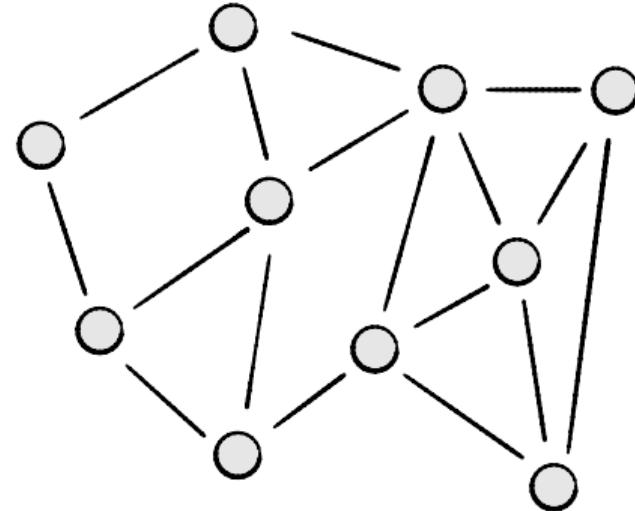
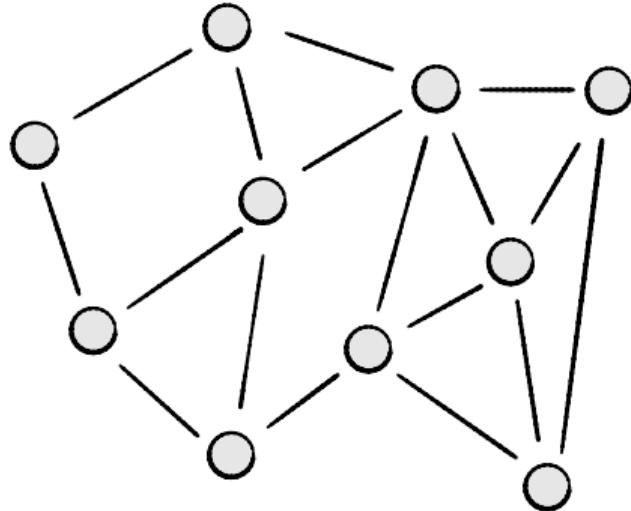
Spanning Trees

Let $G = (V, E)$ be a **connected** graph. A **spanning tree** of G is a tree $T = (V, E_T)$ with the same nodes as G , and a subset of G 's edges.



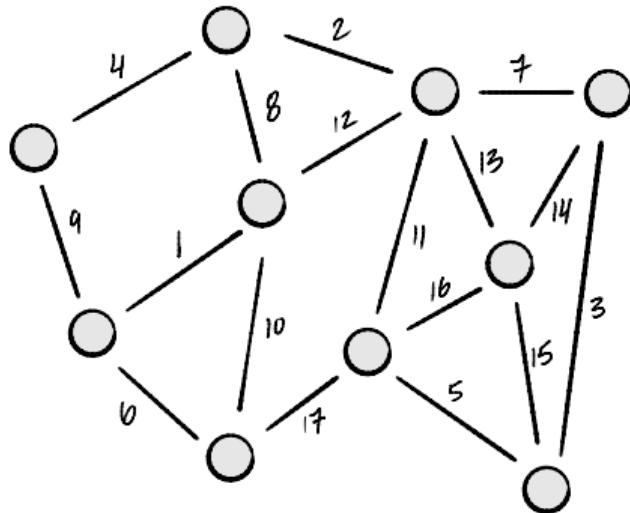
Many Spanning Trees

The same graph can have many spanning trees.



Spanning Tree Cost

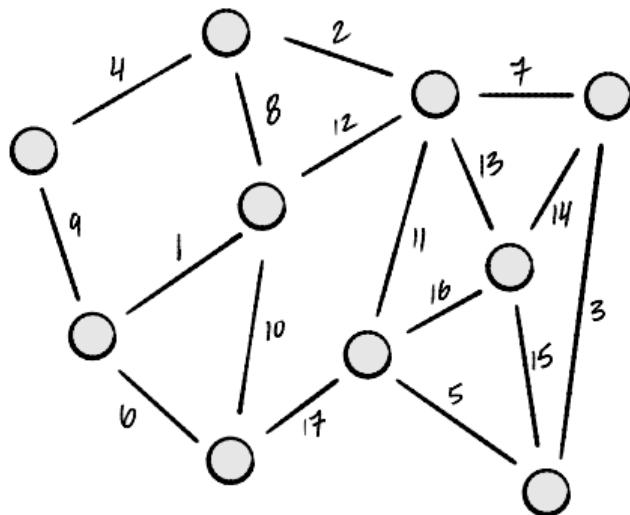
If $G = (V, E, \omega)$ is a weighted undirected graph, the **cost** (or **weight**) of a spanning tree is the total weight of the edges in the spanning tree.



Cost:

Spanning Tree Cost

Different spanning trees of the same graph can have different costs.



Cost:

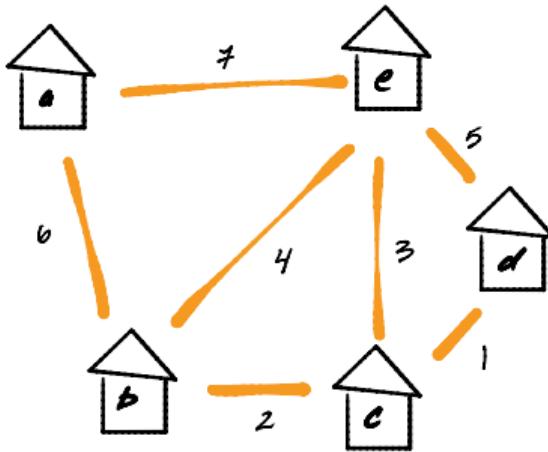
Minimum Spanning Tree

- ▶ The **minimum spanning tree** problem is as follows:
 - ▶ GIVEN: A weighted, undirected graph $G = (V, E, \omega)$.
 - ▶ COMPUTE: a spanning tree of G with minimum cost (i.e., minimum total edge weight).
- ▶ For a given graph, the MST may not be unique.

Exercise

Suppose the edges of a graph $G = (V, e, \omega)$ all have the same weight. How can we compute an MST of the graph?

Today's Problem



- ▶ Choose a set of dirt roads to pave so that:
 - ▶ can get between any two buildings only on paved roads;
 - ▶ total cost is minimized.
- ▶ Solution: compute a **minimum spanning tree**.

MSTs in Data Science?

- ▶ Do we need to find MSTs in data science?
- ▶ Actually, yes! (Next lecture)

DSC 40B

Theoretical Foundations II

Lecture 16 | Part 2

Prim's Algorithm

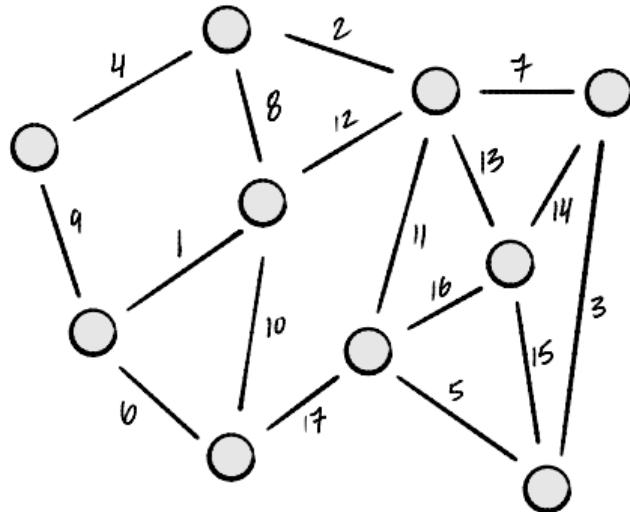
Building MSTs

- ▶ How do we build a MST efficiently?
- ▶ We'll adopt a **greedy** approach.
 - ▶ Build a tree edge-by-edge.
 - ▶ At every step, doing what looks best at the moment.
- ▶ This strategy isn't guaranteed to work in all of life's situations, but it works for building MSTs.

Two Greedy Approaches

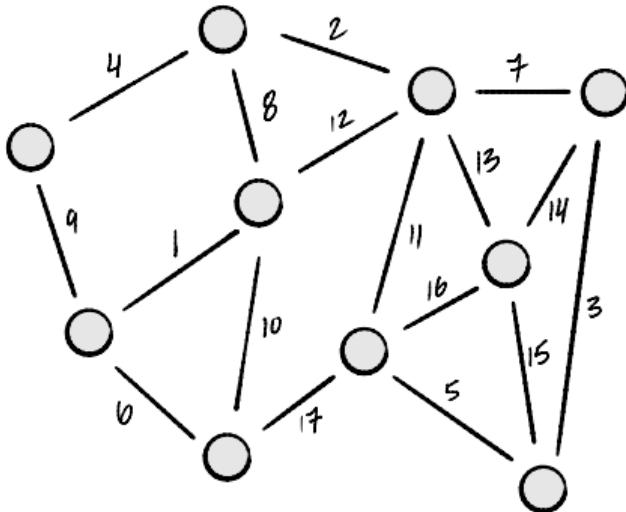
- ▶ We'll look at two greedy algorithms:
 - ▶ Today: Prim's Algorithm
 - ▶ Next time: Kruskal's Algorithm
- ▶ Differ in the order in which edges are added to tree.
- ▶ Also differ in time complexity.

Prim's Algorithm, Informally



- ▶ Start by picking any node to add to “tree”, T .
- ▶ While T is not a spanning tree, greedily add **lightest** edge from a node in T to a node not in T .
 - ▶ “lightest” = edge of smallest weight

Prim's Algorithm, Informally

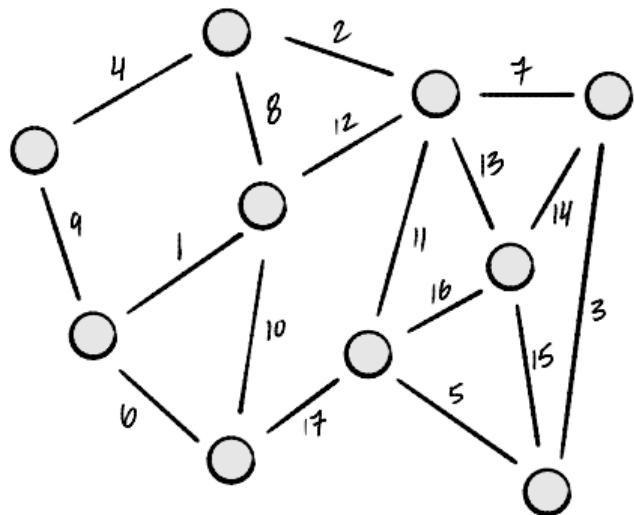


- ▶ Start by picking any node to add to “tree”, T .
- ▶ While T is not a spanning tree, greedily add **lightest** edge from a node in T to a node not in T .
 - ▶ “lightest” = edge of smallest weight
- ▶ **Is this guaranteed to work?** Yes, as we'll see.

Prim's Algorithm, Equivalently

- ▶ For each node u , store:
 - ▶ estimated cost of adding node to tree;
 - ▶ estimated “predecessor” v in the tree.
- ▶ At each step,
 - ▶ Find node with smallest estimated cost.
 - ▶ Add to tree T by including edge with estimated “predecessor”.
 - ▶ Update cost of neighbors.
- ▶ Same as adding lightest edge from T to outside T at every step!

Prim's Algorithm, Equivalently



- ▶ While T is not a tree:
 - ▶ find the node $u \notin T$ with smallest cost
 - ▶ add the edge between u and its estimated “predecessor” to T
 - ▶ update estimated cost/pred. of u 's neighbors which aren't already in tree.

Recall: Priority Queues

- ▶ How do we efficiently find node with smallest cost?
- ▶ Priority Queues:
 - ▶ `PriorityQueue(priorities)`: creates priority queue from dictionary whose values are priorities.
 - ▶ `.extract_min()`: removes and returns key with smallest value.
 - ▶ `.decrease_priority(key, value)`: changes key's value.
- ▶ We'll use a priority queue to hold nodes not yet added to tree.

```
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u

    return tree
```

Prim and Dijkstra

- ▶ This is a lot like Dijkstra's Algorithm for s.p.d.!
- ▶ Both: at each step, extract node with smallest cost, update its edges. (Prim: only those edges to nodes not in tree).
- ▶ Dijkstra update of (u, v) :

$$\text{cost}[v] = \min(\text{cost}[v], \text{cost}[u] + \text{weight}(u, v))$$

- ▶ Prim update of (u, v) :

$$\text{cost}[v] = \min(\text{cost}[v], \text{weight}(u, v))$$

DSC 40B

Theoretical Foundations II

Lecture 16 | Part 3

Time Complexity

Time Complexity

- ▶ A priority queue can be implemented using a **heap**.
- ▶ If a **binary min-heap** is used:
 - ▶ `PriorityQueue(est)` takes $\Theta(V)$ time.
 - ▶ `.extract_min()` takes $O(\log V)$ time.
 - ▶ `.decrease_priority()` takes $O(\log V)$ time.

Time Complexity

```
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Time Complexity

- ▶ Using a **binary heap**...
- ▶ Overall: $\Theta(V \log V + E \log V)$.
- ▶ Since graph is assumed connected, $E = \Omega(V)$.
- ▶ So this simplifies to $\Theta(E \log V)$.

Fibonacci Heaps

- ▶ A priority queue can be implemented using a **heap**.
- ▶ If a **Fibonacci min-heap** is used:
 - ▶ `PriorityQueue(est)` takes $\Theta(V)$ time.
 - ▶ `.extract_min()` takes $\Theta(\log V)$ time¹.
 - ▶ `.decrease_priority()` takes $O(1)$ time.

¹Amortized.

Time Complexity

```
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Time Complexity

- ▶ Using a **Fibonacci heap**...
- ▶ Overall: $\Theta(V \log V + E)$.

Fibonacci vs. Binary Heaps

- ▶ Using Fibonacci heaps improves time complexity when graph is dense.
- ▶ E.g., if $E = \Theta(V^2)$:
 - ▶ Prim's with Fibonacci: $\Theta(E) = \Theta(V^2)$
 - ▶ Prim's with binary: $\Theta(E \log E) = \Theta(V^2 \log V)$.
- ▶ But Fibonacci heaps are **hard** to implement; have large constants.
- ▶ Binary heaps used more in practice despite complexity.

DSC 40B

Theoretical Foundations II

Lecture 16 | Part 4

Correctness of Prim's Algorithm

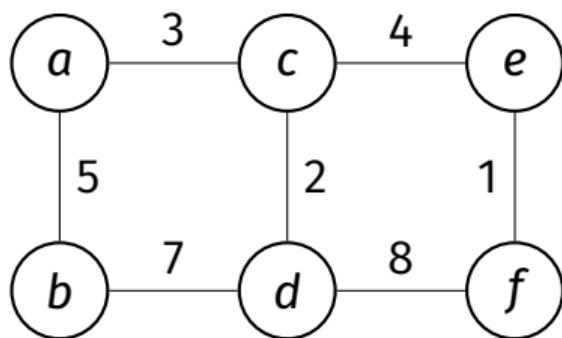
Being Greedy

- ▶ At every step, we add the lightest edge.
- ▶ Is this “safe”?

Being Greedy

- ▶ At every step, we add the lightest edge.
- ▶ Is this “safe”?
- ▶ Yes! This is guaranteed to find an MST.

Promising Subtrees



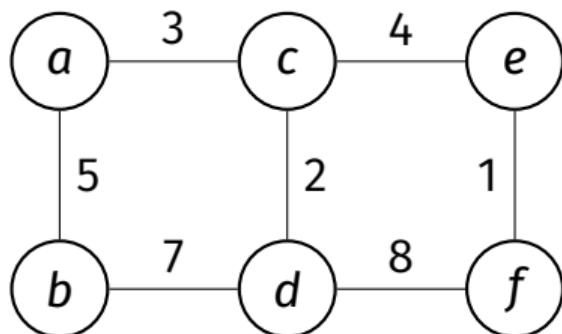
- ▶ Let $G = (V, E, \omega)$ be a weighted graph.
- ▶ A subgraph $T' = (V', E')$ is **promising** if it is “part” of some MST.
 - ▶ That is, it is an “MST in progress”
 - ▶ Not necessarily a tree!
- ▶ That is, there exists an MST $T = (V, E_{\text{mst}})$ such that $E' \subset E_{\text{mst}}$.
- ▶ Hint: a “promising subtree” where $V' = V$ is an MST!

Main Idea

Prim's starts with a promising subtree T . At each step, adds lightest edge from a node within T to a node outside of T .

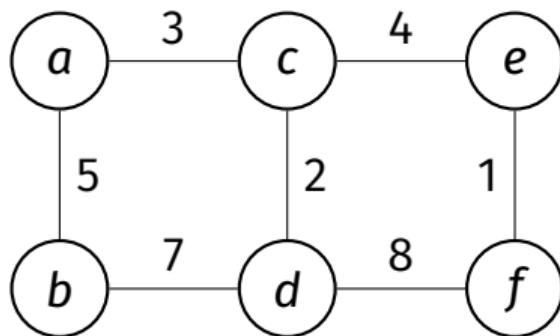
We'll show each new edge results in a larger promising subtree. Eventually the promising subtree becomes a full MST.

Claim



- ▶ Let $G = (V, E, \omega)$ be a weighted graph.
- ▶ Suppose $T' = (V', E')$ is a promising subtree for an MST of G .
- ▶ Let $e = (u, v)$ be a **lightest edge** from a node in T' to a node outside of T' . (Prim).
- ▶ Then adding (u, v) to T' results in another **promising subtree**.

Proof



- ▶ Suppose T_{mst} is an MST that includes T' .
- ▶ If T_{mst} includes e , we're done: $T' + e$ is promising.
- ▶ If it doesn't include e , it must have an edge f that connects T' to rest of the graph.
- ▶ Swap f with e in T_{mst} . The result is a tree, and it must be a MST since $\omega(e) \leq \omega(f)$.
- ▶ So there is an MST that contains $T' + e$.

DSC 40B

Theoretical Foundations II

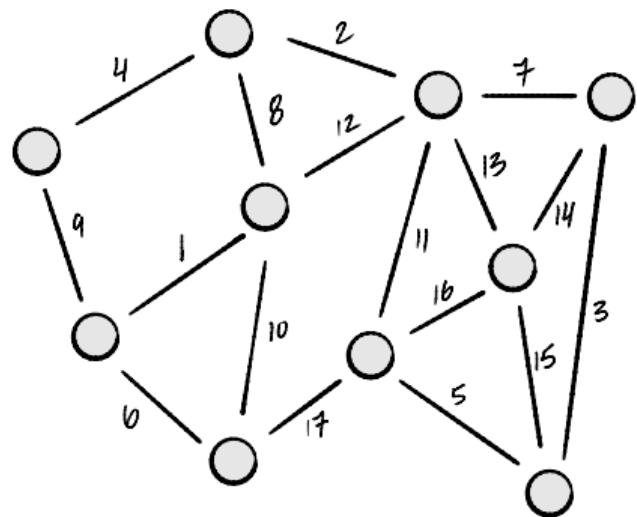
Lecture 17 | Part 1

Kruskal's Algorithm

Last Time: Minimum Spanning Tree

- ▶ The **minimum spanning tree** problem is as follows:
 - ▶ GIVEN: A weighted, undirected graph $G = (V, E, \omega)$.
 - ▶ COMPUTE: a spanning tree of G with minimum cost (i.e., minimum total edge weight).

Example



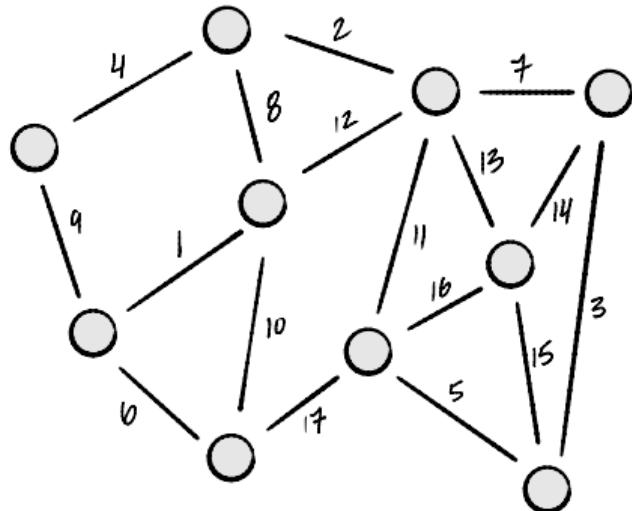
Last Time: Building MSTs

- ▶ How do we build a MST efficiently?
- ▶ We'll adopt a **greedy** approach.
 - ▶ Build a tree edge-by-edge.
 - ▶ At every step, doing what looks best at the moment.
- ▶ This strategy isn't guaranteed to work in all of life's situations, but it works for building MSTs.

Two Greedy Approaches

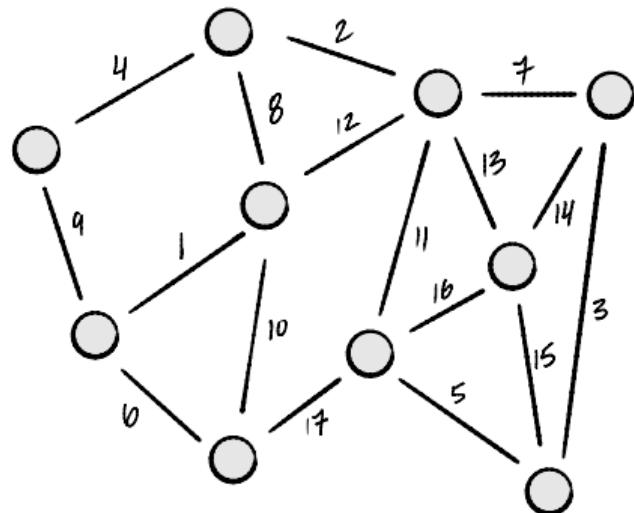
- ▶ We'll look at two greedy algorithms:
 - ▶ Last Time: Prim's Algorithm
 - ▶ Today: Kruskal's Algorithm
- ▶ Differ in the order in which edges are added to tree.
- ▶ Also differ in time complexity.

Prim's Algorithm, Informally



- ▶ Start by picking any node to add to “tree”, T .
- ▶ While T is not a spanning tree, greedily add **lightest** edge from a node in T to a node not in T .
 - ▶ “lightest” = edge of smallest weight

Kruskal's Algorithm, Informally



- ▶ Start with empty forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \emptyset$.
- ▶ Loop through edges in increasing order of weight.
 - ▶ If edge does not create a cycle in T , add it to T .
 - ▶ If T is a spanning tree, break.

Being Greedy

- ▶ Prim: add the **node** with smallest estimated cost and update neighbors.
 - ▶ Works locally, “grows” a connected tree.
- ▶ Kruskal: add the **edge** with smallest weight.
 - ▶ As long as it doesn’t make a cycle.
 - ▶ Edge can be anywhere in graph.

Kruskal's Algorithm (Pseudocode)

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        # if u and v are not already connected
        if ...:
            mst.add_edge(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Checking for Connectivity

- ▶ Each iteration: check if u and v are connected in $T = (V, E_{\text{mst}})$.
- ▶ We *could* do a DFS/BFS on each iteration...
 - ▶ $\Theta(V + E_{\text{mst}}) = \Theta(V)$ each time.
 - ▶ **Expensive!**
- ▶ Remember:
 - ▶ If you're computing something once, use a fast algorithm.
 - ▶ If you're computing it repeatedly, consider a **data structure**.

Disjoint Set Forests

- ▶ Represent a collection of disjoint sets.

$\{\{1, 5, 6\}, \{2, 3\}, \{0\}, \{4\}\}$

- ▶ `.union(x, y)`: Union the sets containing x and y .
- ▶ `.in_same_set(x, y)`: Return **True/False** if x and y are in the same set.¹

¹Usually implemented as a `.find(x)` method returning representative of set containing x .

Example

```
»> # create a DSF with {{0}, {1}, {2}, {3}, {4}, {5}}
»> dsf = DisjointSetForest([0, 1, 2, 3, 4, 5])
»> dsf.union(0, 3)
»> dsf.union(1, 4)
»> dsf.union(3, 1)
»> dsf.union(2, 5)
»> # dsf now represents {{0, 1, 3, 4}, {2, 5}}
»> dsf.in_same_set(0, 3)
True
»> dsf.in_same_set(0, 2)
False
```

Disjoint Set Forests

- ▶ Operations take $\Theta(\alpha(n))$ time, where n is number of objects in collection.
- ▶ $\alpha(n)$ is the **inverse Ackermann function**.
- ▶ It grows very, **very** slowly.
- ▶ Essentially constant time.

Disjoint Set Forests

- ▶ Can be used to keep track of CCs of a **dynamic graph**.
- ▶ Nodes of CCs are disjoint sets.
 - ▶ Add an edge (u, v) : `.union(u, v)`
 - ▶ Check if u and v are connected:
`.in_same_set(u, v)`
- ▶ To check if u, v are already connected:
 - ▶ BFS/DFS: $\Theta(V)$ each time.
 - ▶ DSF: $\Theta(\alpha(V))$ each time (essentially $\Theta(1)$).

Kruskal's Algorithm

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

    # (optional) if mst is now a spanning tree, break
    if len(mst.edges) == len(graph.nodes) - 1:
        break

return mst
```

Time Complexity

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

        # (optional) if mst is now a spanning tree, break
        if len(mst.edges) == len(graph.nodes) - 1:
            break

    return mst
```

Time Complexity

- ▶ Assume graph is connected. Then $E = \Omega(V)$.
- ▶ Kruskal's takes $\Theta(E \log E) = \Theta(E \log V)$ time.
 - ▶ Dominated by sorting the edges.
- ▶ Note: if graph disconnected, Kruskal's produces a **minimum spanning forest**.

DSC 40B

Theoretical Foundations II

Lecture 17 | Part 2

Kruskal v. Prim

Kruskal v. Prim

- ▶ Both algorithms for computing MSTs.
- ▶ Which is “better”?
- ▶ There’s no clear winner.

Time Complexity

- ▶ Prim:
 - ▶ Binary heap: $\Theta(V \log V + E \log V)$
 - ▶ Fibonacci heap: $\Theta(V \log V + E)$
- ▶ Kruskal: $\Theta(E \log V)$
- ▶ If the graph is dense, $E = \Theta(V^2)$, and Prim's with Fibonacci heap “wins”.
 - ▶ $\Theta(V^2)$ versus $\Theta(V^2 \log V)$.

Not so fast...

- ▶ Fibonacci heaps are hard to implement, high overhead.
- ▶ Prim's will be faster for very large dense graphs.
- ▶ But Kruskal's may be faster for smaller dense graphs.
- ▶ The right choice depends on your application.

Main Idea

Asymptotic time complexity isn't everything. For small inputs, the “inefficient” algorithm may beat the “efficient” one. There's also ease of implementation to consider.

DSC 40B

Theoretical Foundations II

Lecture 17 | Part 3

MSTs and Clustering

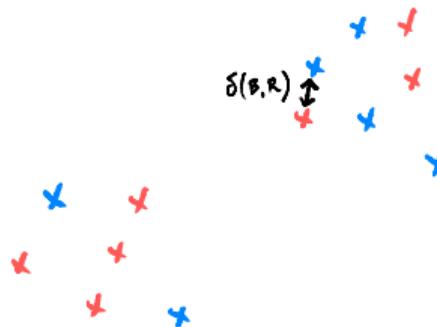
Clustering

Goal: identify the groups in data. Example:



Clustering, Formalized

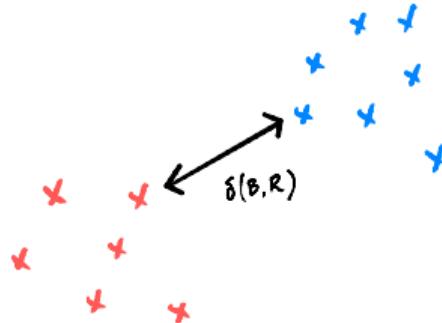
- We frame as an optimization problem.
 - GIVEN: n data points.
 - GOAL: assign color to each point (red or blue) to maximize the distance between the closest pair of red and blue points.



Bad Clustering

Clustering, Formalized

- We frame as an optimization problem.
 - GIVEN: n data points.
 - GOAL: assign color to each point (red or blue) to maximize the distance between the closest pair of red and blue points.



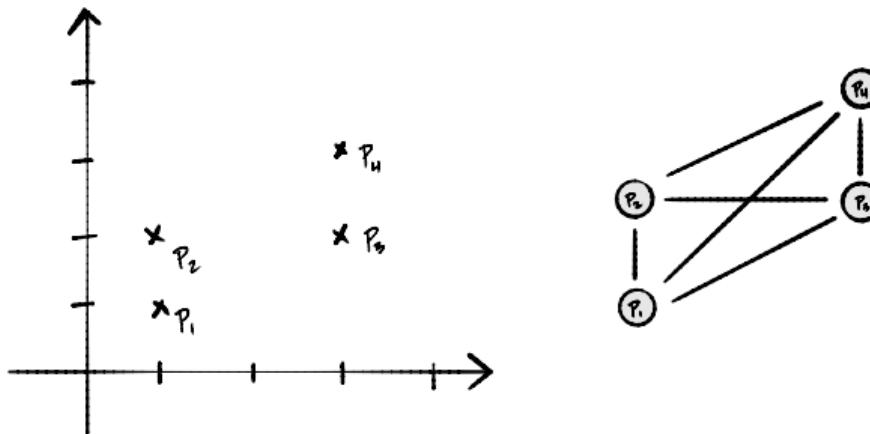
Good Clustering

Brute Force Solution

- ▶ Try all possible assignments; return best.
- ▶ If there are n data points, there are $\Theta(2^n)$ assignments.
- ▶ Exponential time; very slow. Practical only for ~ 50 data points.
- ▶ Instead, we will turn it into a graph problem.

Distance Graphs

- ▶ Given n data points, p_1, p_2, \dots, p_n , create complete graph with $V = \{p_1, \dots, p_n\}$.
- ▶ Set weight of edge $(p_i, p_j) = \text{dist}(p_i, p_j)$.
- ▶ The result is a weighted, undirected **distance graph**.



Main Idea

We can always think of a set of points in a (metric) space as a weighted distance graph. This is a **very** important idea, because it allows us to use our graph algorithms!

Clustering with MSTs

- ▶ Given n data points and a number of clusters, k :
 - ▶ Create distance graph G .
 - ▶ Run Kruskal's Algorithm on G until there are only k components.



- ▶ The resulting connected components are the **clusters**.
- ▶ This is known as **single-linkage clustering**.

Example

Single-Linkage Clustering

- ▶ Time complexity of single-linkage is determined by Kruskal's Algorithm: $\Theta(E \log E)$.
- ▶ Since distance graph is complete, $E = \Theta(V^2)$, and so
$$\Theta(E \log E) = \Theta(V^2 \log V) = \Theta(n^2 \log n)$$
- ▶ Practically, can cluster $\sim 10,000$ points.

Summary

- ▶ We started the quarter with a brute force solution.
 - ▶ Took $\Theta(2^n)$ time, only feasible for a few dozen points.
- ▶ We've now reframed the problem using graph theory.
 - ▶ Now only $\Theta(n^2 \log n)$ time!
 - ▶ Feasible for tens of thousands of points.

Why Algorithms?

- ▶ Data scientists use computers as tools.
- ▶ But solving a problem isn't just about coding it up.
- ▶ You need to know how to analyze your code and use the right algorithms and data structures to make your solution efficient.

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 1

Complexity Theory

The quest for efficient algorithms is about finding clever ways to avoid taking exponential time. So far we have seen the most brilliant successes of this quest; now we meet the quest's most embarrassing and persistent failures.

- paraphrased from *Algorithms* by Dasgupta, Papadimitriou, Vazirani

Exponential to Polynomial

- ▶ Many problems have brute force solutions which take exponential time.
- ▶ Example: clustering to maximize separation
- ▶ The challenge of algorithm design: find a more “efficient” solution.

Polynomial Time

- ▶ If an algorithm's worst case time complexity is $O(n^k)$ for some k , we say that it runs in **polynomial time**.
 - ▶ Example: $\Theta(n \log n)$, since $n \log n = O(n^2)$.
- ▶ Any polynomial is much faster than exponential for big n .
 - ▶ But not necessarily for small n .
 - ▶ Example: n^{100} vs 1.0001^n .
- ▶ We therefore think of polynomial as “efficient”.

Question

- ▶ Is every problem solvable in polynomial time?

Question

- ▶ Is every problem solvable in polynomial time?
- ▶ **No!** Problem: print all permutations of n numbers.

Question

- ▶ Is every problem solvable in polynomial time?
- ▶ **No!** Problem: print all permutations of n numbers.
- ▶ **No!** Problem: given $n \times n$ checkerboard and current pieces, determine if red can force a win.

Ok, then...

- ▶ What problems can be solved in polynomial time?
- ▶ What problems can't?
- ▶ How can I tell if I have a hard problem?

Ok, then...

- ▶ What problems can be solved in polynomial time?
- ▶ What problems can't?
- ▶ How can I tell if I have a hard problem?
- ▶ Core questions in **computational complexity theory**.

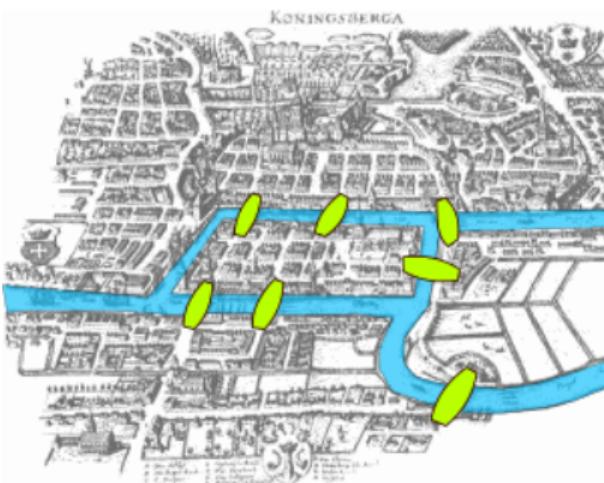
DSC 40B

Theoretical Foundations II

Lecture -1 | Part 2

Eulerian and Hamiltonian Cycles

Example: Bridges of Königsberg



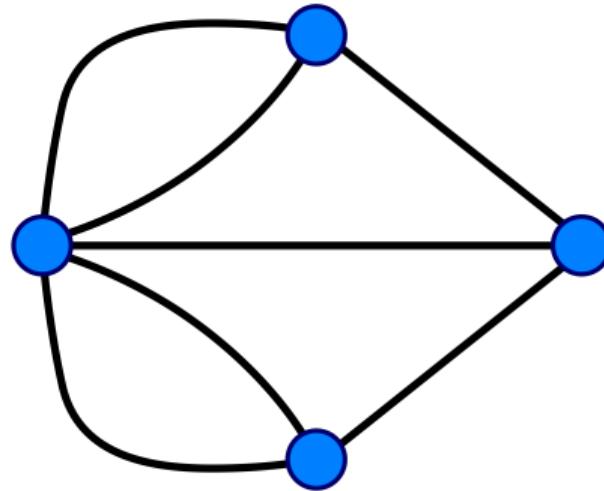
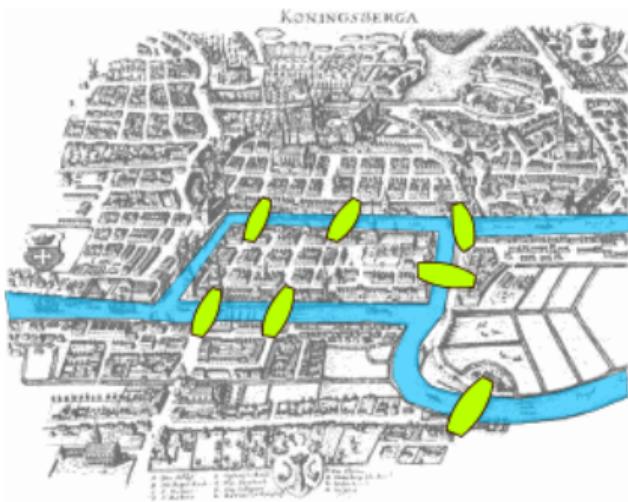
- ▶ **Problem:** Is it possible to start and end at same point while crossing each bridge exactly once?

Leonhard Euler



1707 - 1783

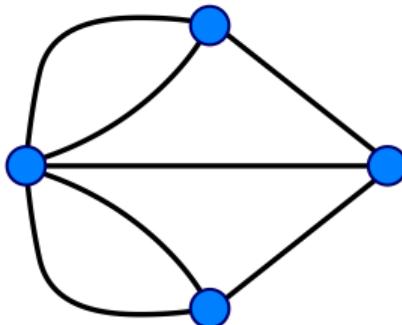
Eulerian Cycle



Is there a cycle which uses each edge exactly once?

Necessary conditions

- ▶ Graph must be connected.
- ▶ Each node must have even degree.
- ▶ Answer for Königsberg answer: it is **impossible**.



In General...

- ▶ These conditions are **necessary** and **sufficient**.
- ▶ A graph has a Eulerian cycle **if and only if**:
 - ▶ it is connected;
 - ▶ each node has even degree.

Exercise

Can we determine if a graph has an Eulerian cycle in time that is polynomial in the number of nodes?

Remember, an Eulerian cycle exists iff the graph is connected and each node has even degree.

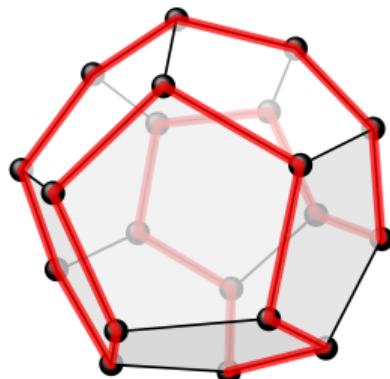
Answer

- ▶ We can check if it is connected in $\Theta(V + E)$ time.
- ▶ Compute every node's degree in $\Theta(V)$ time with adjacency list.
- ▶ Total: $\Theta(V + E) = O(V^2)$. **Yes!**

Gaming in the 19th Century

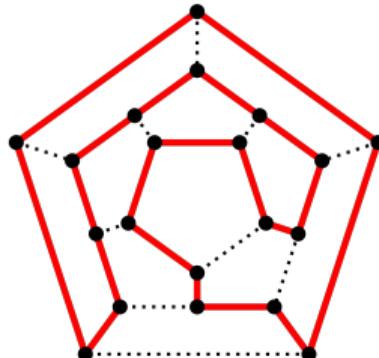
I have found that some young persons have been much amused by trying a new mathematical game which the Icosian furnishes [...]

- W.R. Hamilton, 1856



Hamiltonian Cycles

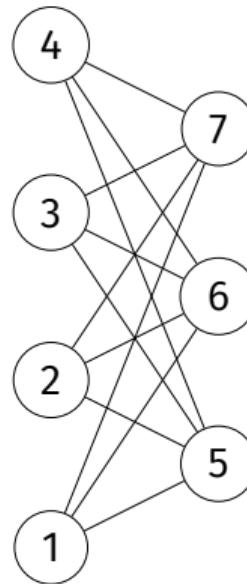
- ▶ A **Hamiltonian cycle** is a cycle which visits each *node* exactly once (except the starting node).
- ▶ Game: find a Hamiltonian cycle on the graph below:



Exercise

Can we determine whether a general graph has a Hamiltonian cycle in polynomial time?

Some cases are easy



In General

- ▶ Could brute-force.
- ▶ How many possible cycles are there?

Hamiltonian Cycles are Difficult

- ▶ This is a **very difficult** problem.
- ▶ No polynomial algorithm is known for general graphs.
- ▶ In special cases, there may be a fast solution.
But in general, worst case is hard.

Note

- ▶ Determining if a graph has a Hamiltonian cycle is **hard**.
- ▶ But if we're given a "hint" (i.e., (v_1, v_2, \dots, v_n) is possibly a Hamiltonian cycle), we can check it very quickly!
- ▶ Hard to solve; but easy to verify "hints".

Similar Problems

- ▶ Eulerian: polynomial algorithm, “**easy**”.
- ▶ Hamiltonian: no polynomial algorithm known, “**hard**”.

Main Idea

Computer science is littered with pairs of similar problems where one easy and the other very hard.

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 3

Shortest and Longest Paths

Problem: SHORTPATH

- ▶ **Input:** Graph¹ G , source u , dest. v , number k .
- ▶ **Problem:** is there a path from u to v of length $\leq k$?
- ▶ **Solution:** BFS or Dijkstra/Bellman-Ford in polynomial time.
- ▶ **Easy!**

¹Weighted with no negative cycles, or unweighted.

Problem: LONGPATH

- ▶ **Input:** Graph² G , source u , dest. v , number k .
- ▶ **Problem:** is there a **simple** path from u to v of length $\geq k$?
- ▶ Naïve solution: try all $V!$ path candidates.

²Weighted or unweighted.

Long Paths

- ▶ There is no known polynomial algorithm for this problem.
- ▶ It is a **hard problem**.
- ▶ But given a “hint” (a possible long path), we can verify it very quickly!

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 4

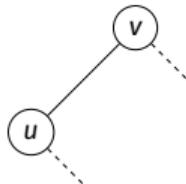
Reductions

Reductions

- ▶ HAMILTONIAN and LONGPATH are related.
- ▶ We can “convert” HAMILTONIAN into LONGPATH in polynomial time.
- ▶ We say that HAMILTONIAN **reduces** to LONGPATH.

Reduction

- ▶ Suppose we have an algorithm for LONGPATH.
- ▶ We can use it to solve HAMILTONIAN as follows:



- ▶ Pick arbitrary node u .
- ▶ For each neighbor v of u :
 - ▶ Create graph G' by copying G , deleting (u, v)
 - ▶ Use algorithm to check if a simple path of length $\geq |V| - 1$ from u to v exists in G' .
 - ▶ If yes, then there is a Hamiltonian cycle.

Reductions

- ▶ If Problem A reduces³ to Problem B, it means “we can solve A by solving B”.
- ▶ Best possible time for A \leq best possible time for B + polynomial
- ▶ “A is no harder than B”
- ▶ “B is at least as hard as A”

³We'll assume reduction takes polynomial time.

Relative Difficulty

- ▶ If Problem A reduces to Problem B, we say B is **at least as hard** as A.
- ▶ Example: HAMILTONIAN reduces to LONGPATH.
LONGPATH is at least as hard as HAMILTONIAN.

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 5

P $\stackrel{?}{=}$ NP

Decision Problems

- ▶ All of today's problems are **decision problems**.
 - ▶ Output: yes or no.
 - ▶ Example: Does the graph have an Euler cycle?

P

- ▶ Some problems have polynomial time algorithms.
 - ▶ SHORTPATH, EULER
- ▶ The set of decision problems that can be solved in polynomial time is called P.
- ▶ Example: SHORTPATH and EULER are in P.

NP

- ▶ The set of decision problems with “hints” that can be verified in polynomial time is called **NP**.
- ▶ All of today’s problems are in NP.
 - ▶ All problems in P are also in NP.
- ▶ Example: SHORTPATH, EULER, HAMILTONIAN, LONGPATH are all in NP.

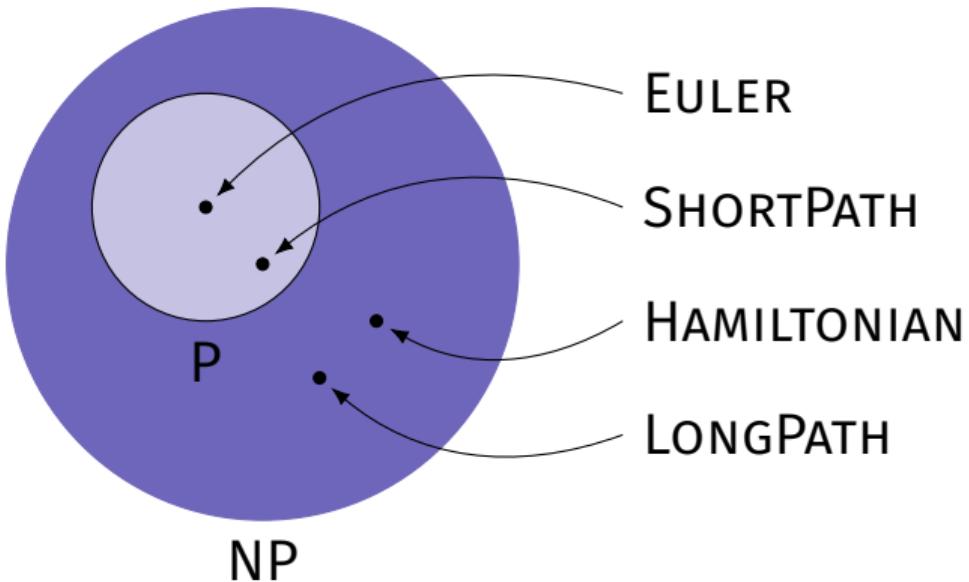
$P \subset NP$

- ▶ P is a subset of NP .
- ▶ It *seems* like some problems in NP aren't in P .
 - ▶ Example: HAMILTONIAN, LONGPATH.
 - ▶ We don't know polynomial time algorithms for these problems.
- ▶ But that doesn't such an algorithm is impossible!

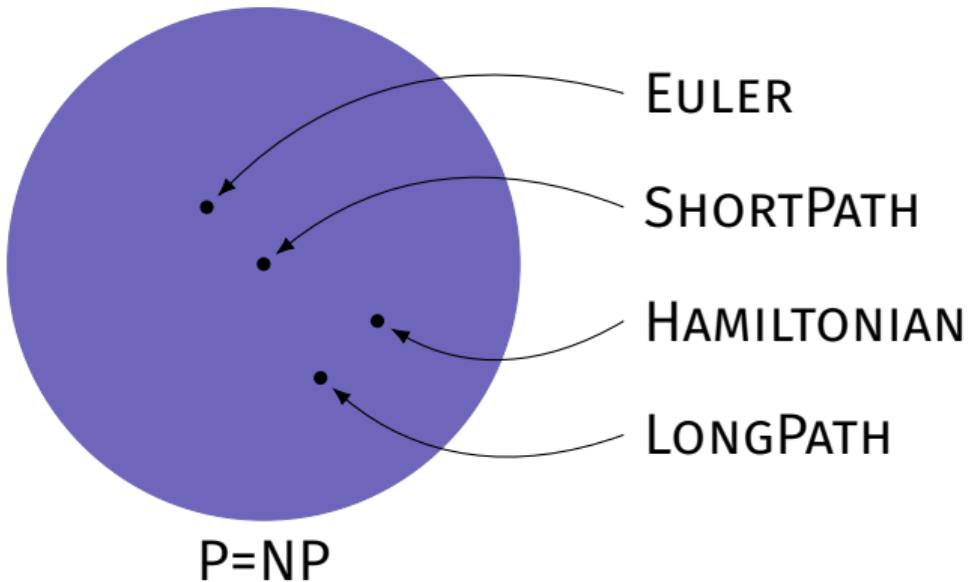
P = NP?

- ▶ Are there problems in NP that aren't in P?
 - ▶ That is, is $P \neq NP$?
- ▶ Or is any problem in NP also in P?
 - ▶ That is, is $P = NP$?

P \neq NP



P = NP



P = NP?

- ▶ Is P = NP?

⁴If you solve it, you'll be rich and famous.

P = NP?

- ▶ Is P = NP?
- ▶ **No one knows!**
- ▶ Biggest open problem in Math/CS.⁴
- ▶ Most think P \neq NP.

⁴If you solve it, you'll be rich and famous.

What if P = NP?

- ▶ Possibly Earth-shattering.
 - ▶ Almost all cryptography instantly becomes obsolete;
 - ▶ Logistical problems solved exactly, quickly;
 - ▶ *Mathematicians* become obsolete.
- ▶ But maybe not...
 - ▶ Proof could be non-constructive.
 - ▶ Or, constructive but really inefficient. E.g., $\Theta(n^{10000})$

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 6

NP-Completeness

Problem: 3-SAT

- ▶ Suppose x_1, \dots, x_n are boolean variables
(True, False)
- ▶ A **3-clause** is a combination made by **or-ing** and possibly negating three variables:
 - ▶ $x_1 \text{ or } x_5 \text{ or } (\text{not } x_7)$
 - ▶ $(\text{not } x_1) \text{ or } (\text{not } x_2) \text{ or } (\text{not } x_4)$

Problem: 3-SAT

- ▶ **Given:** m clauses over n boolean variables.
- ▶ **Problem:** Is there an assignment of x_1, \dots, x_n which makes all clauses true simultaneously?
- ▶ No polynomial time algorithm is known.
- ▶ But it is easy to verify a solution, given a hint.
 - ▶ 3-SAT is in NP.

Cook's Theorem

Every problem in NP is polynomial-time reducible to 3-SAT.

- ▶ ...including Hamiltonian, long path, etc.
- ▶ 3-SAT is at least as hard as every problem in NP.
- ▶ “hardest problem in NP”

Cook's Theorem (Corollary)

- ▶ If 3-SAT is solvable in polynomial time, then all problems in NP are solvable in polynomial time.
 - ▶ ...including Hamiltonian, long path, etc.

NP-Completeness

- ▶ We say that a problem is **NP-complete** if:
 - ▶ it is in NP;
 - ▶ every problem in NP is reducible to it.
- ▶ HAMILTONIAN, LONGPATH, 3-SAT are all NP-complete.
- ▶ NP-complete problems are the “hardest” in NP.

Equivalence

- ▶ In some sense, NP-complete problems are equivalent to one another.
- ▶ E.g., a fast algorithm for HAMILTONIAN gives a fast algorithm for 3-SAT, LONGPATH, and all problems in NP.

Who cares?

- ▶ Complexity theory is a fascinating piece of science.
- ▶ But it's practically useful, too, for recognizing hard problems when you stumble upon them.

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 7

Hard Optimization Problems

Hard Optimization problems

- ▶ NP-completeness refers to **decision problems**.
- ▶ What about optimization problems?
- ▶ We can typically state a similar decision problem.
- ▶ If that decision problem is hard, then optimization is at least as hard.

Problem: bin packing

- ▶ Optimization problem:
 - ▶ **Given:** bin size B , n objects of size $\alpha_1, \dots, \alpha_n$..
 - ▶ **Problem:** find minimum number of bins k that can contain all n objects.
- ▶ Decision problem version:
 - ▶ **Given:** bin size B , n objects of size $\alpha_1, \dots, \alpha_n$, integer k .
 - ▶ **Problem:** is it possible to pack all n objects into k bins?
- ▶ Decision problem is NP-complete, reduces to optimization problem.

Example: traveling salesperson

- ▶ Optimization problem:
 - ▶ **Given:** set of n cities, distances between each.
 - ▶ **Problem:** find shortest Hamiltonian cycle.
- ▶ Decision problem:
 - ▶ **Given:** set of n cities, distance between each, length ℓ .
 - ▶ **Problem:** is there a Hamiltonian cycle of length $\leq \ell$?
- ▶ Decision problem is NP-complete, reduces to optimization problem.

NP-complete problems in machine learning

- ▶ Many machine learning problems are NP-complete.
- ▶ Examples:
 - ▶ Finding a linear decision boundary to minimize misclassifications in non-separable regime.
 - ▶ Minimizing k -means objective.

So now what?

- ▶ Just because a problem is NP-Hard, doesn't mean you should give up.
- ▶ Usually, an approximation algorithm is fast, "good enough".
- ▶ Some problems are even hard to *approximate*.

Summary

- ▶ Not every problem can be solved efficiently.
- ▶ Computer scientists are able to categorize these problems.

DSC 40B

Theoretical Foundations II

Lecture -1 | Part 8

The Halting Problem

Really hard problems

- ▶ Some decision problems are harder than others.
- ▶ That is, it takes more time to solve them.
- ▶ Given enough time, all decision problems can be solved, right?

Alan Turing



1912-1954

Turing's Halting Problem

- ▶ **Given:** a function f and an input x .
- ▶ **Problem:** does $f(x)$ halt, or run forever?
- ▶ Algorithm must work for all functions/inputs!

Turing's Argument

- ▶ Turing says: no such algorithm can exist.
- ▶ Suppose there is a function `halts(f, x)`:
 - ▶ Returns **True** if $f(x)$ halts.
 - ▶ Returns **False** if $f(x)$ loops forever.

Turing's Argument

```
def evil_function(f):
    if halts(f, f):
        # loop forever
    else: # it runs forever
        return
```

- ▶ Consider `evil_function(evil_function)`.
 - ▶ Does it halt or not?

Turing's Argument

```
def evil_function(f):
    if halts(f, f):
        # loop forever
    else: # it runs forever
        return
```

- ▶ Consider `evil_function(evil_function)`.
 - ▶ Does it halt or not?
- ▶ Assuming that `halt` works leads to logical impossibility!
 - ▶ So a working `halt` cannot exist.

Undecidability

- ▶ The halting problem is **undecidable**.
- ▶ Fact of the universe: there can be no algorithm for solving it which works on all functions/inputs.
- ▶ All of these problems are undecidable:
 - ▶ Does the program terminate?
 - ▶ Does this line of code ever run?
 - ▶ Does this function compute what its specification says?
 - ▶ Many others...

Reality

- ▶ **Physics:** can't go faster than the speed of light.
- ▶ **Computer science:**
 - ▶ There's a speed limit for certain problems, too.
 - ▶ And some problems can't even be solved!

The End