Objectives

- Clustering
- Data Compression: Huffman Codes

Implementing Kruskal's Algorithm

- Using the **union-find** data structure
  - Build set T of edges in the MST
  - Maintain set for each connected component

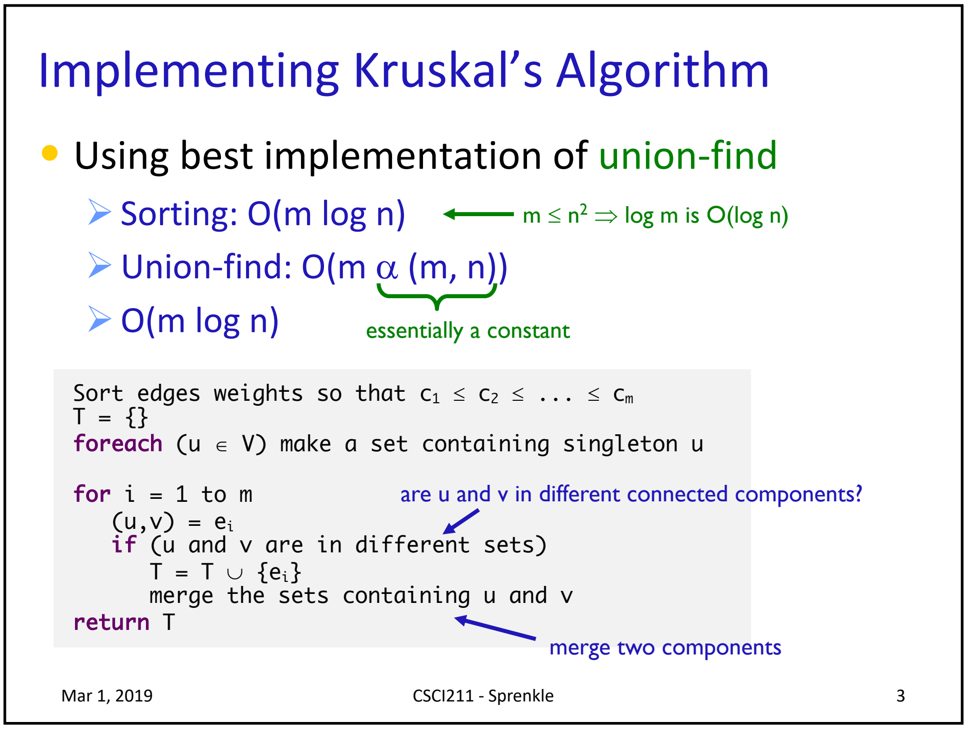
Costs?

```
Sort edge weights so that $c_1 \le c_2 \le \ldots \le c_m$
T = {}
foreach ($u \in V$) make a set containing singleton u

for i = 1 to m
   (u,v) = $e_i$
   if (u and v are in different sets)
      T = T $\cup$ {$e_i$}
      merge the sets containing u and v
return T
```

are u and v in different connected components?

merge two components



Implementing Kruskal's Algorithm

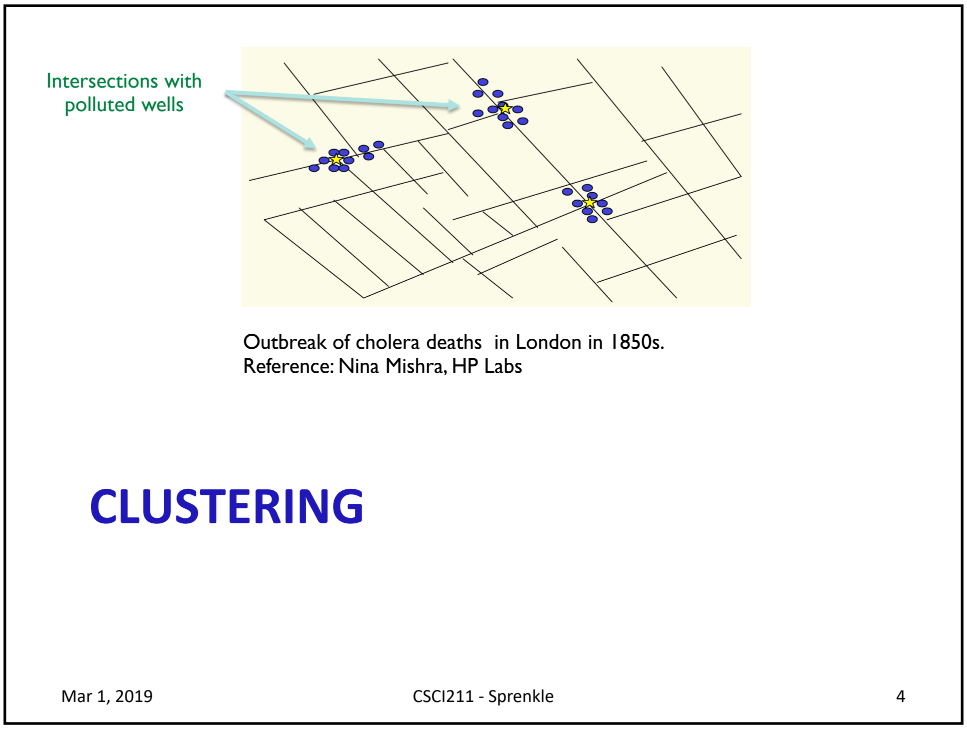
- Using best implementation of union-find
  - Sorting: O(m log n) ← $m \le n^2 \Rightarrow \log m$ is O(log n)
  - Union-find: $O(m\, \underbrace{\alpha\ (m, n)}_{\text{essentially a constant}})$
  - O(m log n)

```
Sort edges weights so that $c_1 \le c_2 \le \dots \le c_m$
T = {}
foreach (u $\in$ V) make a set containing singleton u

for i = 1 to m
   (u,v) = $e_i$
   if (u and v are in different sets)
      T = T $\cup$ {$e_i$}
      merge the sets containing u and v
return T
```

are u and v in different connected components?

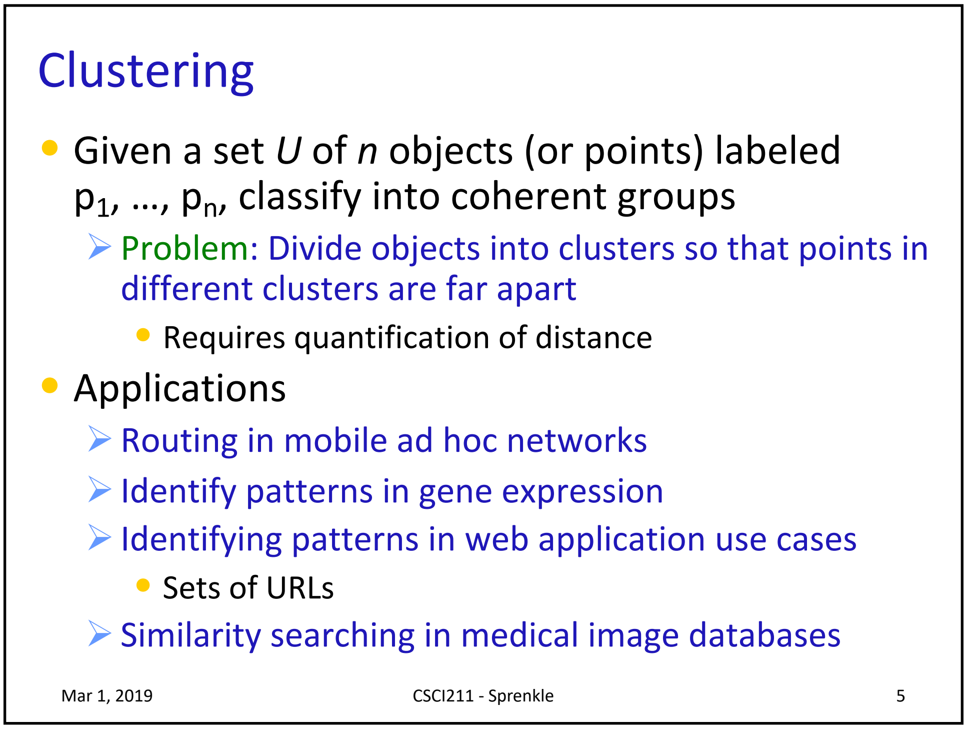
merge two components



Intersections with polluted wells

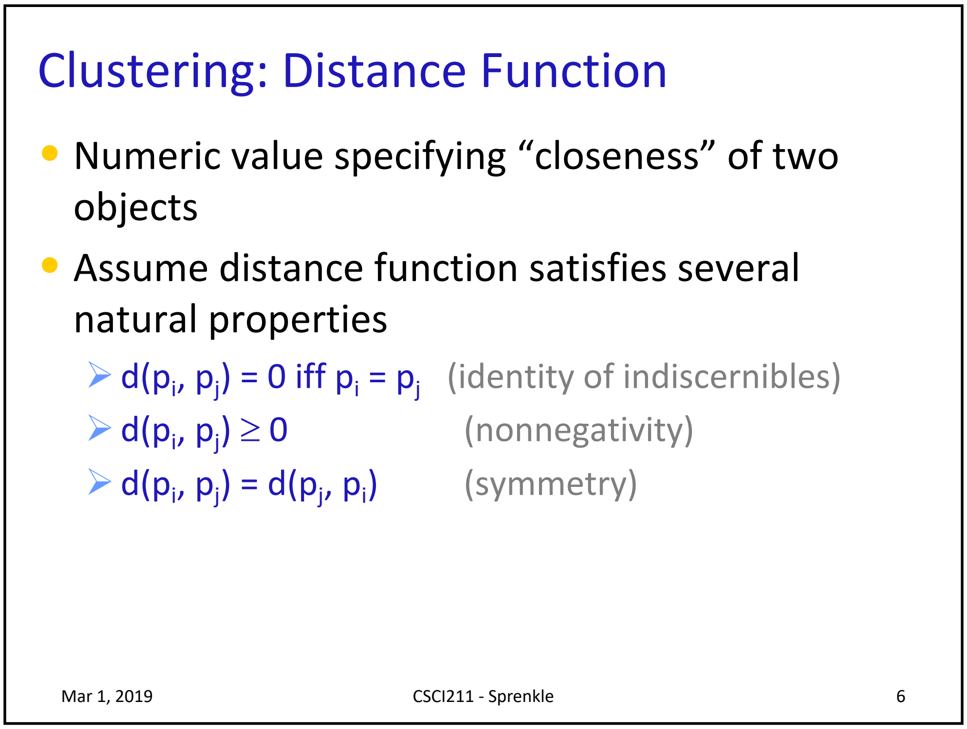
Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

CLUSTERING



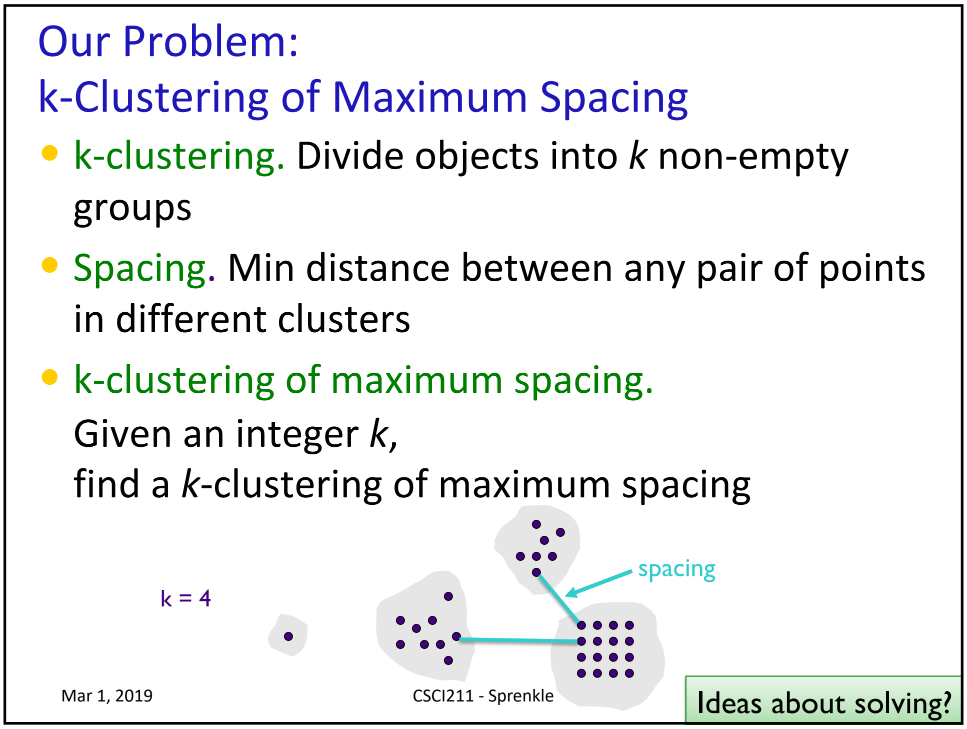
Clustering

- Given a set *U* of *n* objects (or points) labeled $p_1, \ldots, p_n$, classify into coherent groups
  - Problem: Divide objects into clusters so that points in different clusters are far apart
    - Requires quantification of distance
- Applications
  - Routing in mobile ad hoc networks
  - Identify patterns in gene expression
  - Identifying patterns in web application use cases
    - Sets of URLs
  - Similarity searching in medical image databases



Clustering: Distance Function

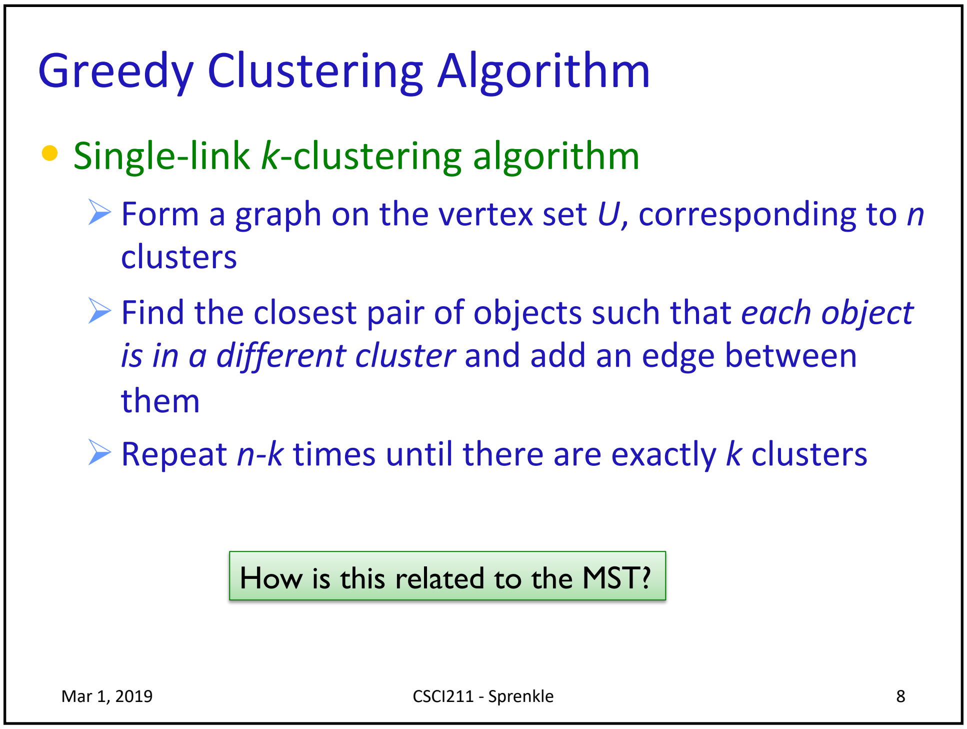
- Numeric value specifying "closeness" of two objects
- Assume distance function satisfies several natural properties
  - $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
  - $d(p_i, p_j) \geq 0$ (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)



Our Problem: k-Clustering of Maximum Spacing

- k-clustering. Divide objects into *k* non-empty groups
- Spacing. Min distance between any pair of points in different clusters
- k-clustering of maximum spacing.
  Given an integer *k*,
  find a *k*-clustering of maximum spacing

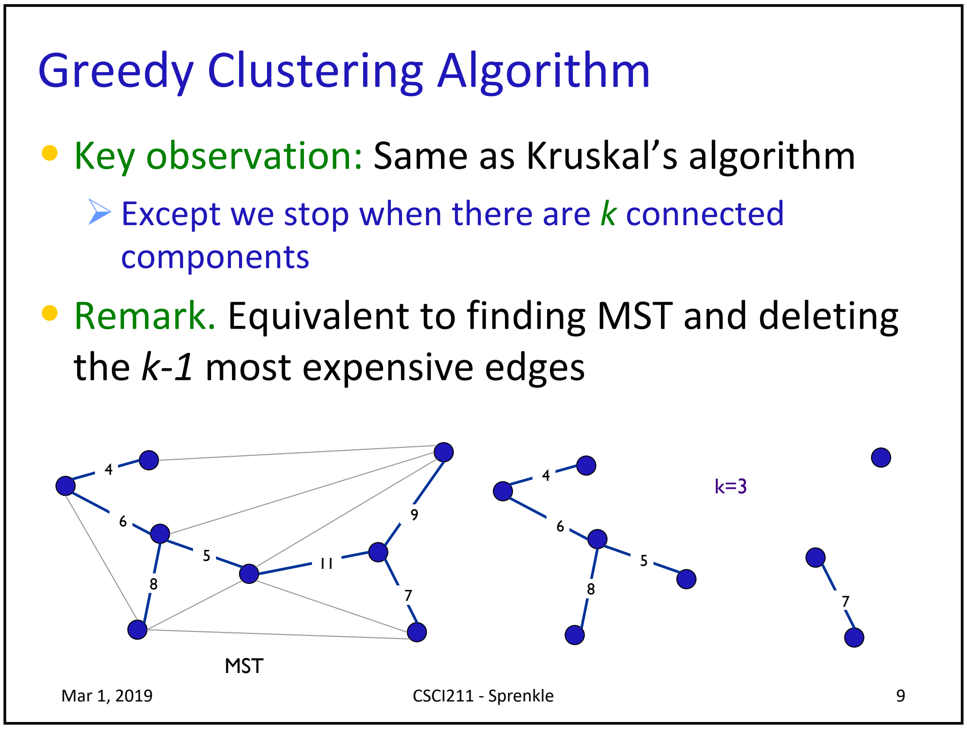
Ideas about solving?



Greedy Clustering Algorithm

- Single-link *k*-clustering algorithm
  - Form a graph on the vertex set *U*, corresponding to *n* clusters
  - Find the closest pair of objects such that *each object is in a different cluster* and add an edge between them
  - Repeat *n-k* times until there are exactly *k* clusters

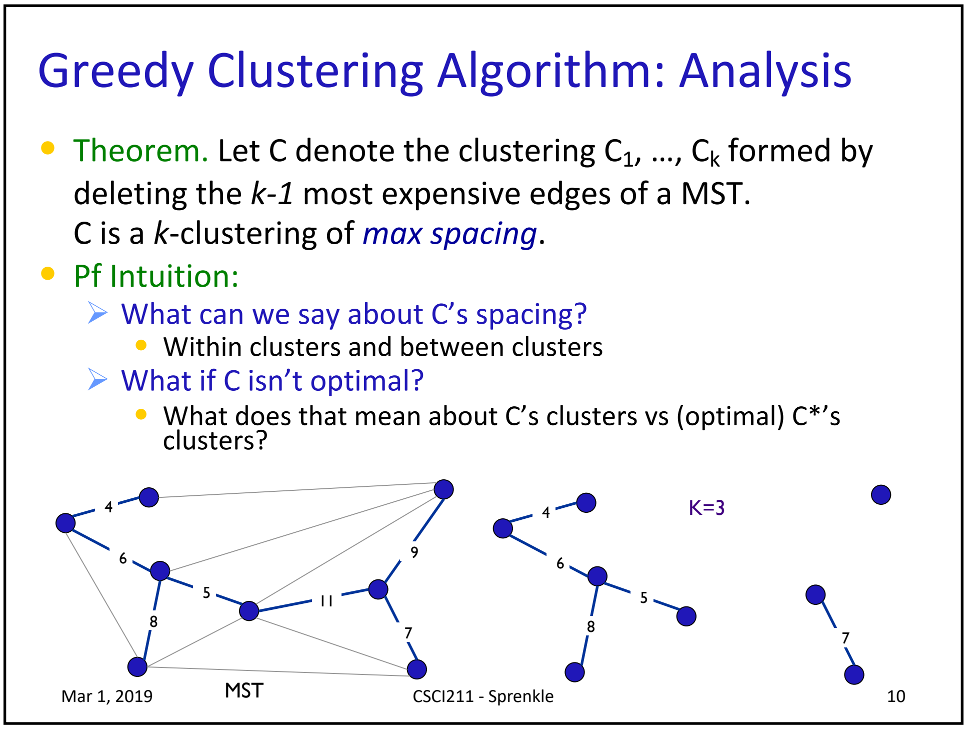
How is this related to the MST?



Greedy Clustering Algorithm

- Key observation: Same as Kruskal's algorithm
  - Except we stop when there are *k* connected components
- Remark. Equivalent to finding MST and deleting the *k-1* most expensive edges

4
6
5
8
11
9
7
MST
4
6
5
8
k=3
7



Greedy Clustering Algorithm: Analysis

- Theorem. Let C denote the clustering C_1, …, C_k formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of *max spacing*.
- Pf Intuition:
  - What can we say about C's spacing?
    - Within clusters and between clusters
  - What if C isn't optimal?
    - What does that mean about C's clusters vs (optimal) C*'s clusters?

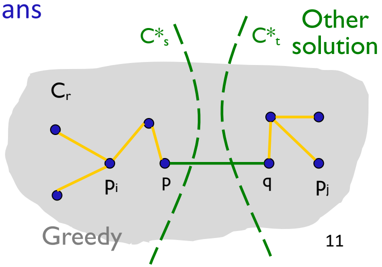
4
6
5
8
11
9
7
MST
4
6
5
8
K=3
7

Greedy Clustering Algorithm: Analysis

- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of *maximum spacing*.

- Pf Sketch. Let C* denote some other clustering $C^*_1, \ldots, C^*_k$. C* and C must be different; otherwise we're done.
  - The spacing of *C* is length *d* of (k-1)st most expensive edge
  - Let p_i, p_j be in the same cluster in Greedy solution *C* (say C_r) but different clusters in other solution C*, say C^*_s and C^*_t
  - Some edge (*p*, *q*) on p_i-p_j path in C_r spans two different clusters in *C**

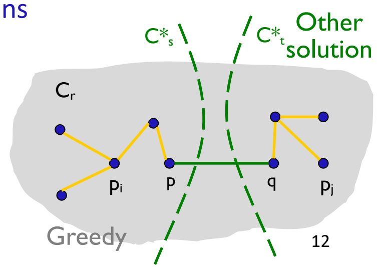
What do we know about *(p, q)*?



Greedy Clustering Algorithm: Analysis

- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of *maximum spacing*.

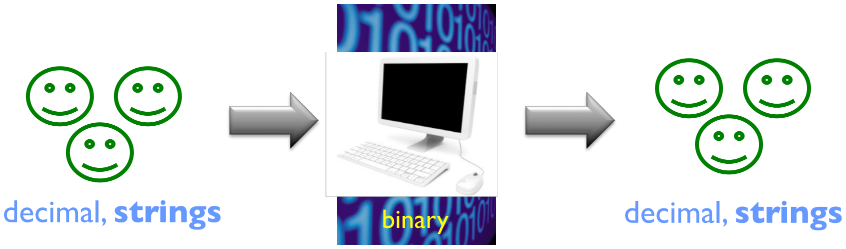
- Pf. Let C* denote some other clustering $C^*_1, \ldots, C^*_k$. C* and C must be different; otherwise we're done.
  - The spacing of *C* is length *d* of (k-1)st most expensive edge
  - Let p_i, p_j be in the same cluster in *C* (say C_r) but different clusters in C*, say C^*_s and C^*_t
  - Some edge (*p*, *q*) on p_i-p_j path in C_r spans two different clusters in *C**
  - All edges on p_i-p_j path have length $\leq d$ since Kruskal chose them
  - Spacing of C* is at most $\leq$ d since *p* and *q* are in different clusters



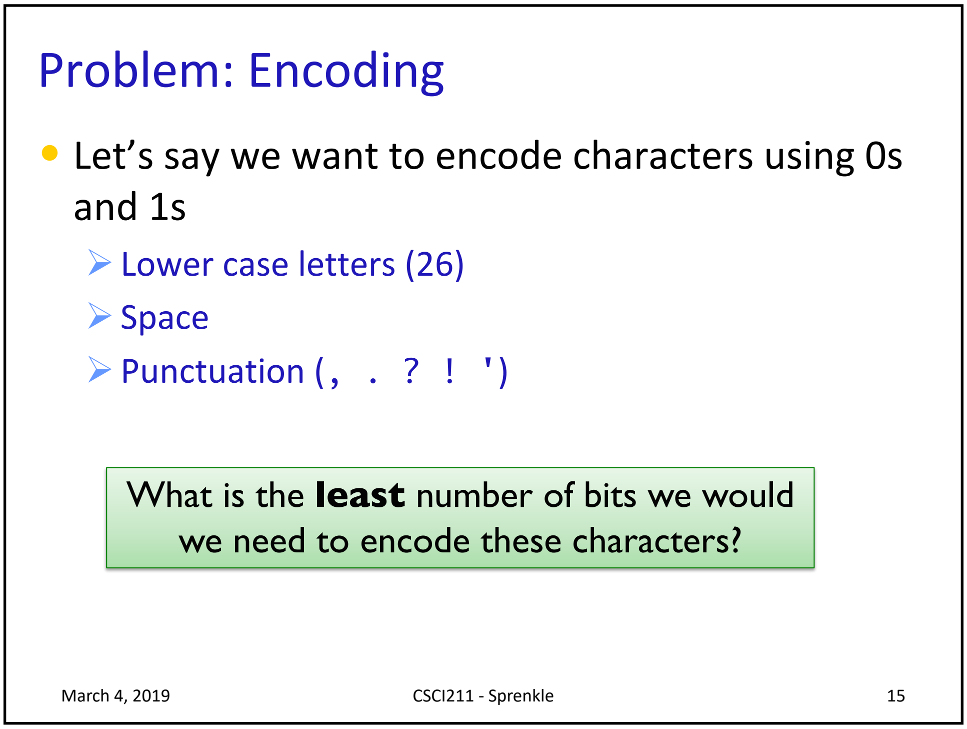
ENCODING

Problem: Encoding

- Computers use bits: 0s and 1s
- Need to represent what we (humans) know to what computers know



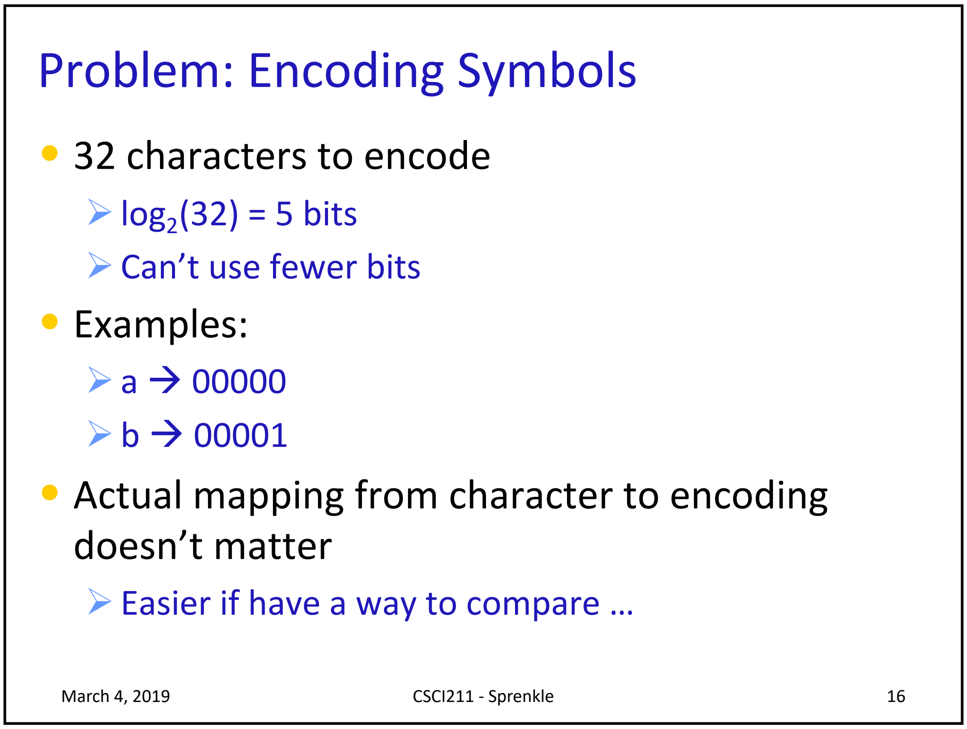
- ➢ Map **symbol** → unique sequence of 0s and 1s
- ➢ Process is called *encoding*



Problem: Encoding

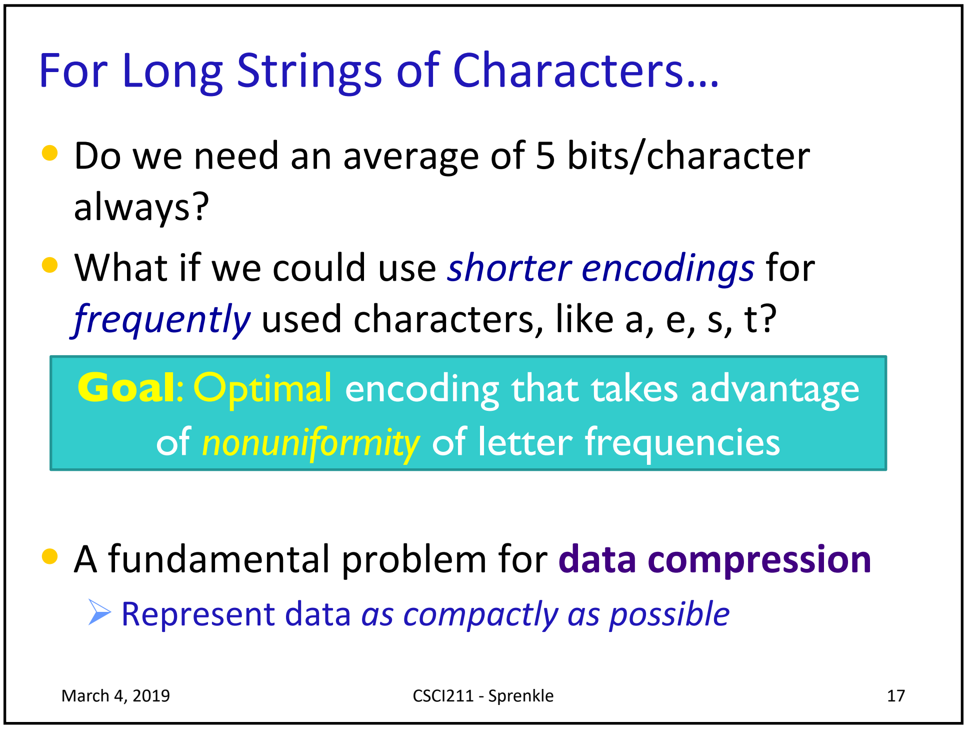
- Let’s say we want to encode characters using 0s and 1s
  - Lower case letters (26)
  - Space
  - Punctuation (`,  .  ?  !  '`)

What is the **least** number of bits we would we need to encode these characters?



Problem: Encoding Symbols

- 32 characters to encode
  - $\log_2(32) = 5$ bits
  - Can’t use fewer bits
- Examples:
  - a → 00000
  - b → 00001
- Actual mapping from character to encoding doesn’t matter
  - Easier if have a way to compare …

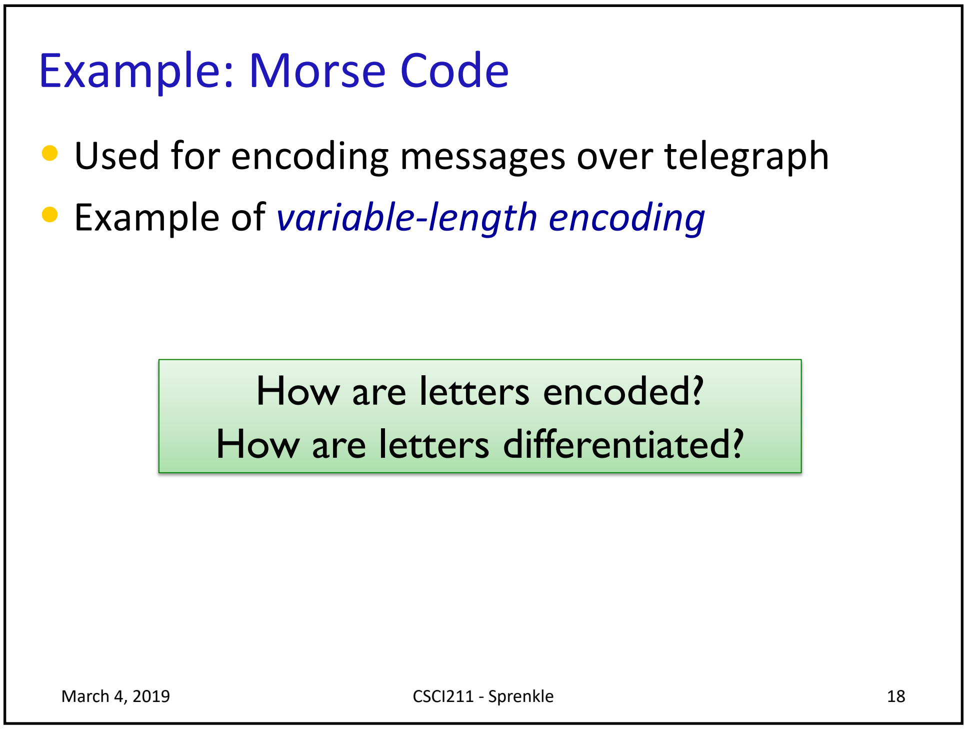


For Long Strings of Characters…

- Do we need an average of 5 bits/character always?
- What if we could use *shorter encodings* for *frequently* used characters, like a, e, s, t?

Goal: Optimal encoding that takes advantage of *nonuniformity* of letter frequencies

- A fundamental problem for **data compression**
  - Represent data *as compactly as possible*



Example: Morse Code

- Used for encoding messages over telegraph
- Example of *variable-length encoding*

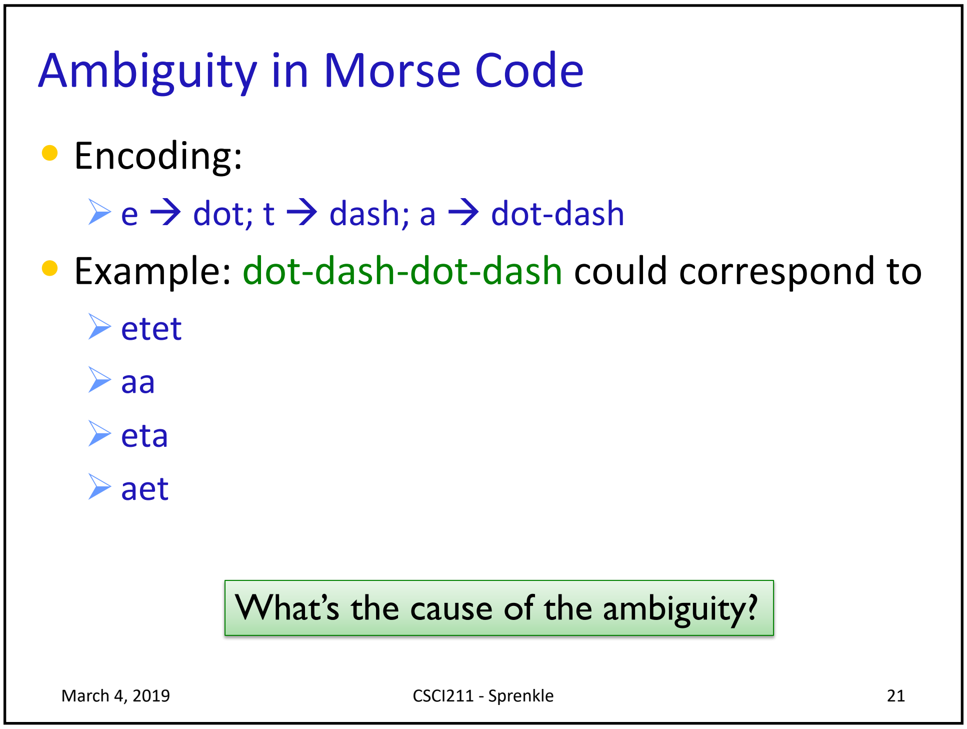
How are letters encoded?
How are letters differentiated?

Example: Morse Code

- Used for encoding messages over telegraph
- Example of *variable-length encoding*
- How are letters encoded?
  - Dots, dashes
  - Most frequent letters use shorter sequences
    - e → dot; t → dash; a → dot-dash
- How are letters differentiated?
  - Spaces in between letters
    - Otherwise, ambiguous
    - adds one more character to each letter

Ambiguity in Morse Code

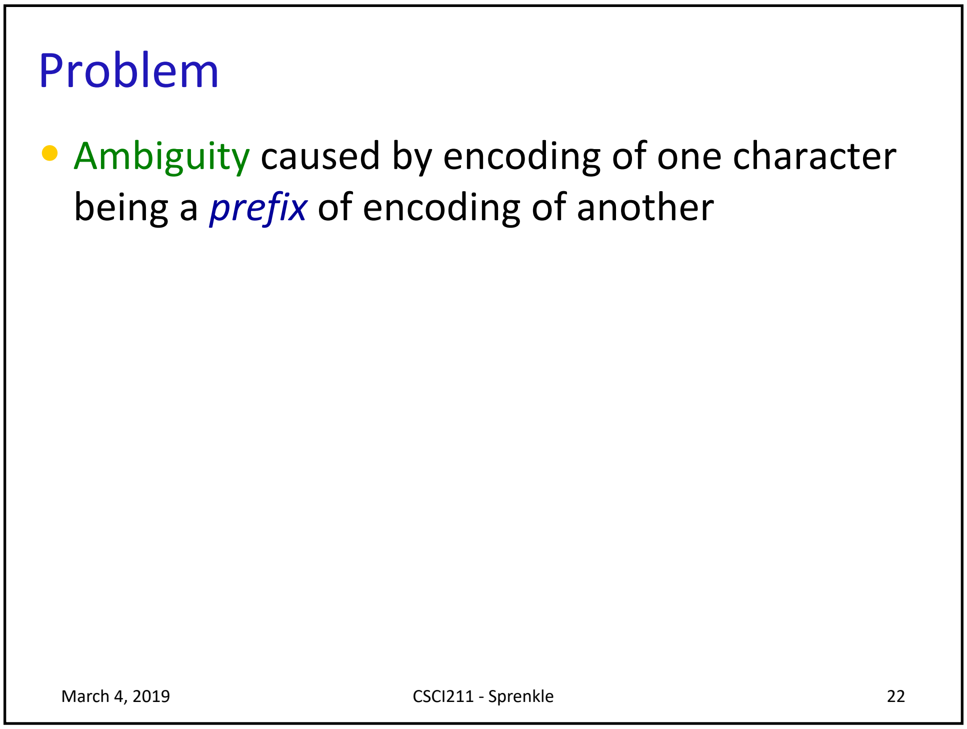
- Encoding:
  - e → dot; t → dash; a → dot-dash
- Example: dot-dash-dot-dash could correspond to:



Ambiguity in Morse Code

- Encoding:
  - e → dot; t → dash; a → dot-dash
- Example: dot-dash-dot-dash could correspond to
  - etet
  - aa
  - eta
  - aet

What's the cause of the ambiguity?



Problem

- Ambiguity caused by encoding of one character being a *prefix* of encoding of another

Prefix Codes

- Problem: Encoding of one character being a *prefix* of encoding of another → ambiguity
- Solution: **Prefix Codes**: map letters to bit strings such that *no encoding is a prefix of any other*
  - Won't need artificial devices like spaces to separate characters
- Example encodings:
  - Verify that no encoding is a prefix of another
  - What is 0010000011101?

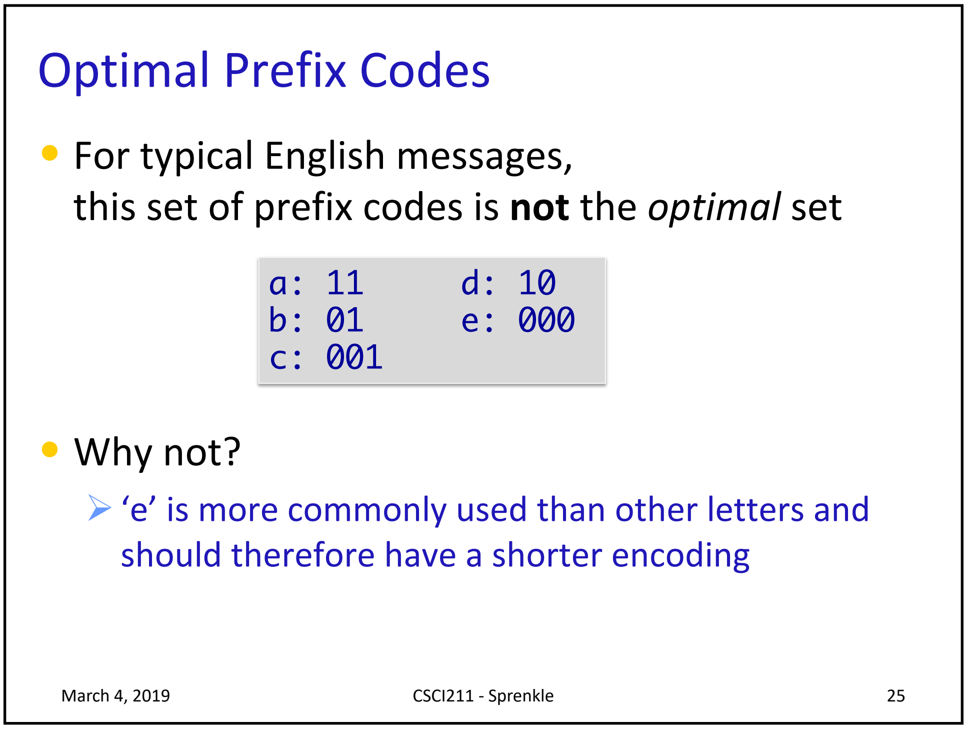
```
a: 11      d: 10
b: 01      e: 000
c: 001
```

Optimal Prefix Codes

- For typical English messages,
  this set of prefix codes is **not** the *optimal* set

```
a: 11      d: 10
b: 01      e: 000
c: 001
```

- Why not?

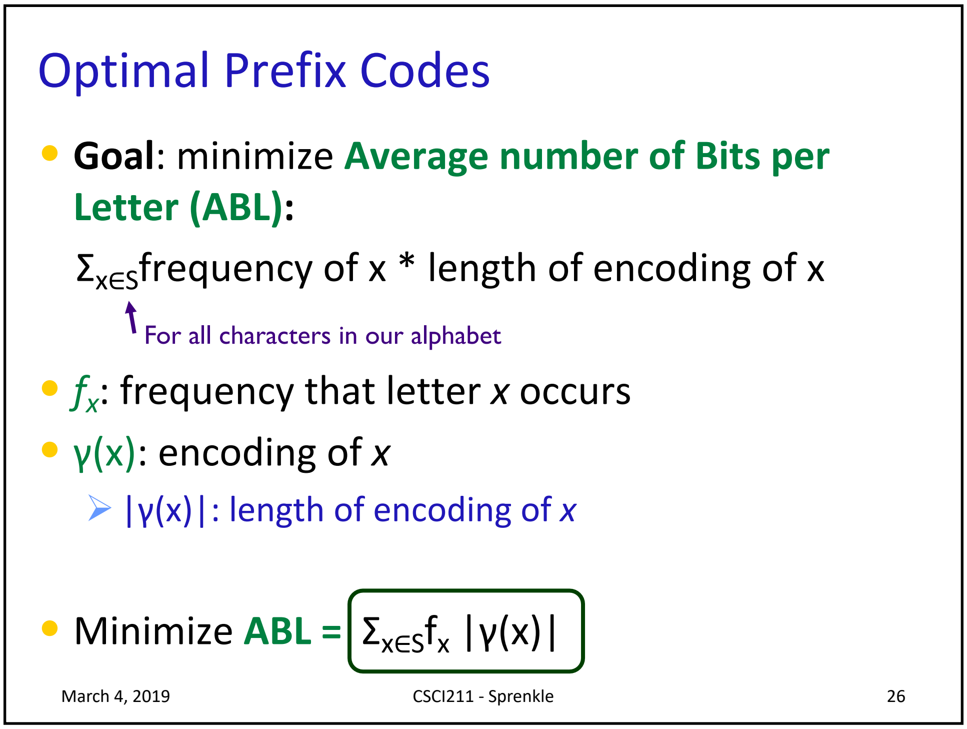


Optimal Prefix Codes

- For typical English messages, this set of prefix codes is **not** the *optimal* set

```
a: 11      d: 10
b: 01      e: 000
c: 001
```

- Why not?
  - ‘e’ is more commonly used than other letters and should therefore have a shorter encoding



Optimal Prefix Codes

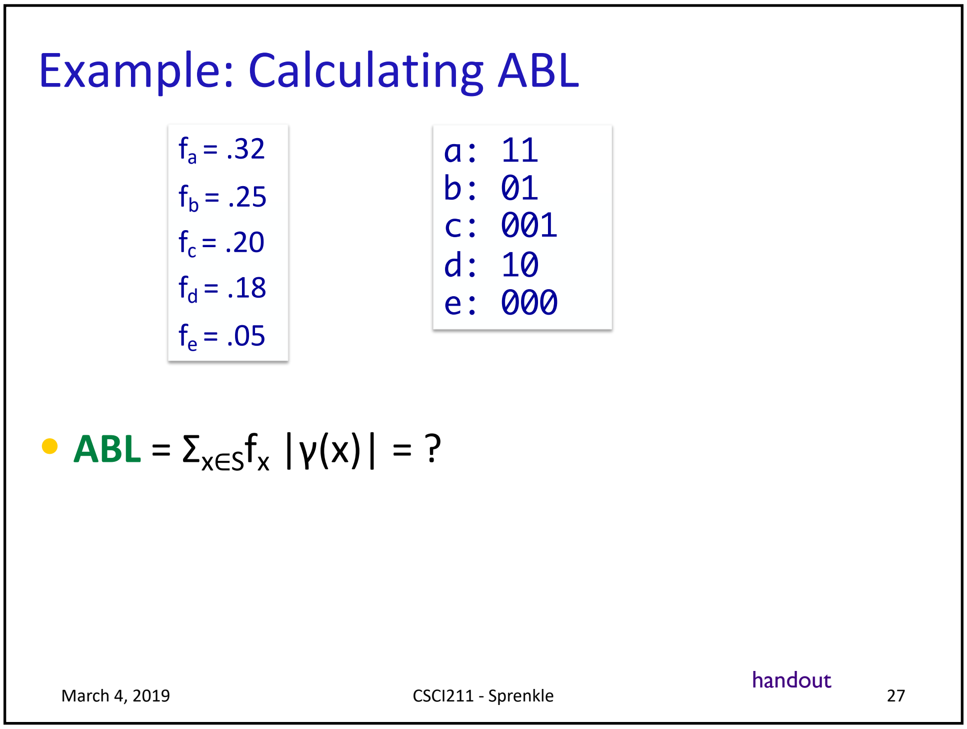
- **Goal**: minimize **Average number of Bits per Letter (ABL)**:

  $\Sigma_{x \in S}$frequency of x * length of encoding of x

  ↑ For all characters in our alphabet

- f_x: frequency that letter *x* occurs
- γ(x): encoding of *x*
  - |γ(x)|: length of encoding of *x*

- Minimize **ABL =** $\boxed{\Sigma_{x \in S} f_x \, |\gamma(x)|}$



Example: Calculating ABL

$f_a = .32$

$f_b = .25$

$f_c = .20$

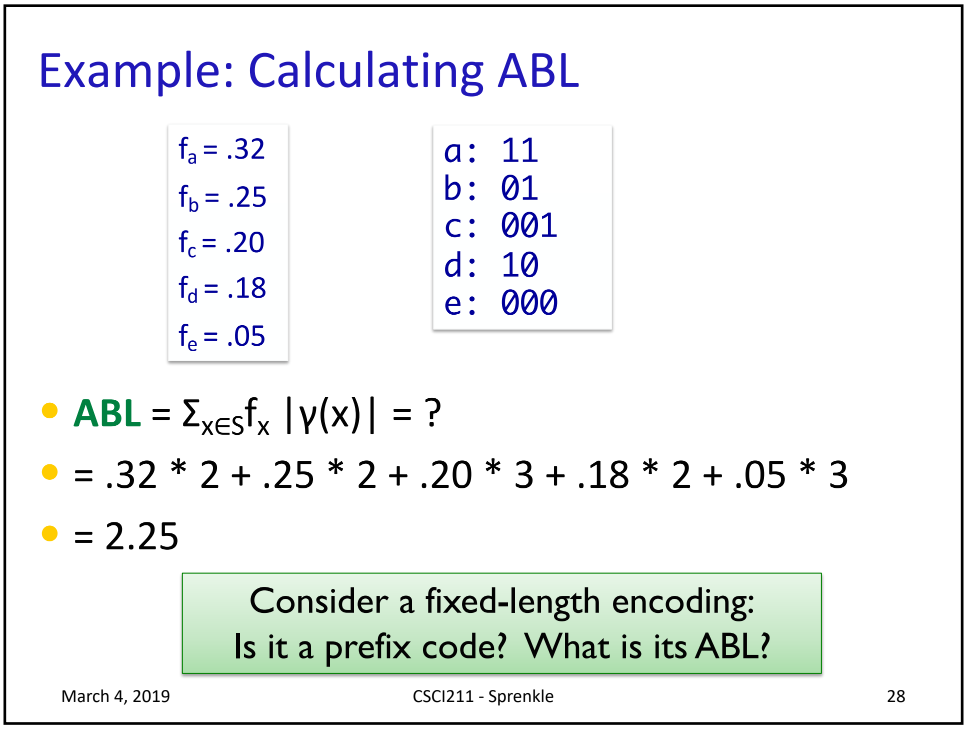
$f_d = .18$

$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

- **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)|$ = ?

handout



Example: Calculating ABL

$f_a = .32$

$f_b = .25$

$f_c = .20$

$f_d = .18$

$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

- **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)|$ = ?
- = .32 * 2 + .25 * 2 + .20 * 3 + .18 * 2 + .05 * 3
- = 2.25

Consider a fixed-length encoding:
Is it a prefix code? What is its ABL?

Fixed-Length Encodings

- Is it a prefix code?
  - Yes. Always look at fixed number of characters
- What is its ABL?
  - ABL is the length of the encoding
- For 5 characters, ABL is 3
- Variable-length prefix code's ABL (2.25) is an improvement

Can We Improve the ABL?

$f_a = .32$

$f_b = .25$

$f_c = .20$

$f_d = .18$

$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

Can We Improve the ABL?

$f_a = .32$

$f_b = .25$

$f_c = .20$

$f_d = .18$

$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

Swap these because *c* occurs more frequently than *d.*
Give *c* the shorter encoding

- **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)|$ = 2.23

Problem Statement

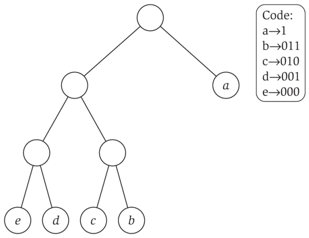
- Given an alphabet and a set of frequencies for the letters, produce optimal (most efficient) prefix code
  - Minimizes average # of bits per letter (ABL)

Approaches to Solution

- Brute force
  - Search space is complicated → all ways to map letters to bit strings that adhere to prefix code property
- Build towards greedy approach
  - Start: representing prefix codes
    - Given we know the codes, how do we represent them?

Binary Trees to Represent Prefix Codes

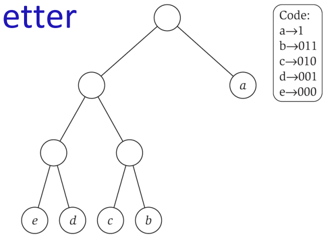
- Exposes structure better than list of mappings
  - Each leaf node is a letter
  - Follow path to the letter
    - Going left: 0
    - Going right: 1



Are these really prefix codes?
How could we show they weren't?

Binary Trees to Represent Prefix Codes

- Structure: Each leaf node is a letter
  - ➢ Follow path to the letter
    - Going left: 0; Going right: 1
- Proof. If it weren't:
  a letter's encoding is a prefix of another letter
  - ➢ Letter is in the path of another letter
  - ➢ But, all letters are leaf nodes
    - Contradiction



Looking Ahead

- Wiki: 4.5-4.7
- Problem Set 6 due Friday