## Objectives

Greedy Algorithms

- Data Compression



## Recap

Looking at various problems with *greedy* solutions

- Greedy: at each step, make locally optimal choice and build from there
- How to prove optimal: stays ahead, exchange

Solving problems using greedy solutions

- Shortest path
- Minimal spanning tree
- Applying minimal spanning tree
  - Clustering
- Today: data compression



## The Plan

Friday:

- Problem Set 3 due
- SSA – Extra credit opportunities
  - Added to homework grade
  - Answer easy for 1 pt, harder for 3 pts

Monday: Wrap-up Chapter 4, Review

Tue-Fri: Open-book midterm

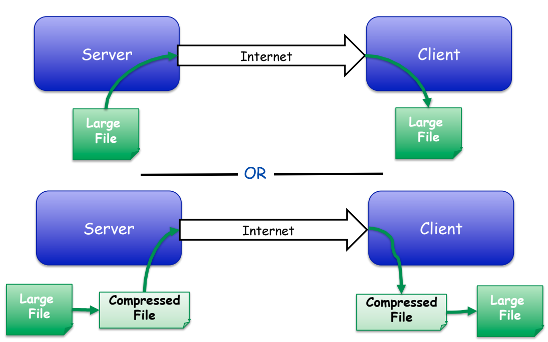
- Covers chapters 1—4 of book
- Similar to problem set
- Turned into my mailbox in CS office by Friday
- I'll be at a conference Tuesday through Saturday
  - Available by email



## DATA COMPRESSION



## Which is Better?






## Which is Better?

Depends on your metrics, compression time/amount

Case 1 requires

- More network resources
- Less CPU time (server: compress; client: uncompress)

Case 2 requires

- Less network resources
- More CPU time (client and server)

Overall best

- Depends on file size, network speed, compression time/amount

➡Bigger files → Case 2

## Problem: Encoding Symbols

Computers use bits: 0s and 1s

Need to represent what we (humans) know as 0s and 1s

- Map symbol to unique sequence of 0s and 1s
- Process is called *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 them?



## 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?



## 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



## 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

- eta
- aa
- etet
- aet

What's the problem?



## Problem

Ambiguity caused by encoding of one character is a *prefix* of encoding of another



## Prefix Codes

Problem: Encoding of one character is a *prefix* of encoding of another

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:

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

- Verify that no encoding is a prefix of another
- What is this? 0010000011101



## Prefix Codes

Problem: Encoding of one character is a *prefix* of encoding of another

Better name: *prefix-free?*

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:

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

- Verify that no encoding is a prefix of another
- What is this? 0010000011101 ⟶ cecab



## 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?



## 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?

- '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

$\gamma(x)$: encoding of $x$

- $|\gamma(x)|$: length of encoding of $x$

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



## Example: Calculating ABL

| | |
|---|---|
| $f_a = .32$ | a: 11 |
| $f_b = .25$ | b: 01 |
| $f_c = .20$ | c: 001 |
| $f_d = .18$ | d: 10 |
| $f_e = .05$ | e: 000 |

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

handout



## Example: Calculating ABL

| | |
|---|---|
| $f_a = .32$ | a: 11 |
| $f_b = .25$ | b: 01 |
| $f_c = .20$ | c: 001 |
| $f_d = .18$ | d: 10 |
| $f_e = .05$ | e: 000 |

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

= .32 * 2 + .25 * 2 + .20 * 3 + .18 * 2 + .05 * 2

= 2.25

What about a fixed-length encoding?

- Is it a prefix code? What is ABL?



## Example: Calculating ABL

Consider a fixed-length encoding

- Is it a prefix code?
- What is its ABL?



## Example: Calculating ABL

Consider a fixed-length encoding

- Is it a prefix code?
  - Yes. Always look at fixed number of characters
- What is its ABL?
  - ABL is the length of the encoding

| |
|---|
| a: 11 |
| b: 01 |
| c: 001 |
| d: 10 |
| e: 000 |

For 5 characters, ABL is 3

Variable-length prefix code's ABL (2.25) is an improvement



## Can We Improve On This?

| | |
|---|---|
| $f_a = .32$ | a: 11 |
| $f_b = .25$ | b: 01 |
| $f_c = .20$ | c: 001 |
| $f_d = .18$ | d: 10 |
| $f_e = .05$ | e: 000 |

## Can We Improve On This?

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$

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

*Swap these because c occurs more frequently than d. So, 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 number of bits per letter



## 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

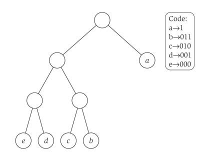


## Binary Trees to Represent Prefix Codes

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?

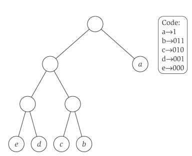





## Binary Trees to Represent Prefix Codes

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





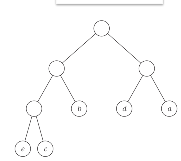

## Building the Binary Tree

How do we build the binary tree for this mapping?

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

Tree Rules:

- Each leaf node is a letter
- Follow path to the letter
  - Going left: 0
  - Going right: 1



## Recursively Generate Tree

All letters are in root node

For all letters in node

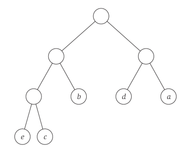
- If encoding begins with 0, letter belongs in left subtree
- Otherwise (encoding begins with 1), letter belongs in right subtree
- If last bit of encoding, make the letter a leaf node of that subtree
- Shift encoding one bit
- Process left and right children



## Tree Properties

What is the length of a letter's encoding?

Define our optimal goal in tree terms





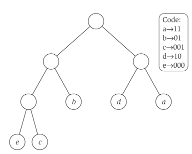

## Tree Properties

What is the length of a letter's encoding?

- Length of path from root to leaf → its *depth*

Define our optimal goal in tree terms

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

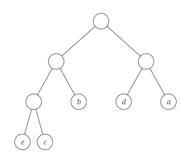





## Tree Properties

What do we want our tree to look like for the optimal solution?

- How many leaves?
- How many internal nodes?
  - Think about parent nodes vs child nodes
- When uniform frequencies?
- Nonuniform frequencies?






## Tree Properties

Claim. The binary tree corresponding to the optimal prefix code is *full*, i.e., each internal node has two children.

Proof?

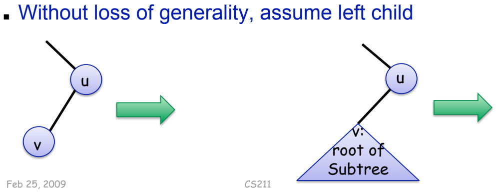


## Tree Properties

Claim. The binary tree *T* corresponding to the optimal prefix code is *full*, i.e., each internal node has two children.

Proof. Assume that *T* has an internal node with only one child

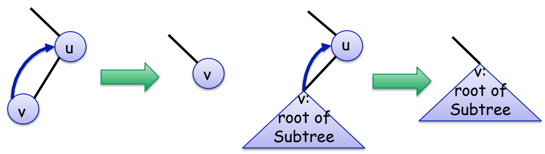
- Without loss of generality, assume left child



## Tree Properties

Claim. The binary tree *T* corresponding to the optimal prefix code is *full*, i.e., each internal node has two children.

Proof. Assume that *T* has an internal node with only one child




Replace *u* with *v* → decrease depth → original wasn't optimal



## Toward a Solution…

Two problems to solve:

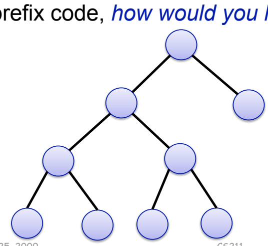
- Creating the prefix code tree
- Labeling the prefix code tree with alphabet/frequencies



## Simplifying: Know Optimal Prefix Code

Process: assume knowledge of optimal solution to gain insight into finding solution

Assume we knew the tree structure of the optimal prefix code, *how would you label the leaf nodes?*

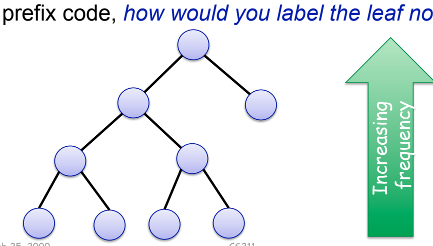




## Simplifying: Know Optimal Prefix Code

Process: assume knowledge of optimal solution to gain insight into finding solution

Assume we knew the tree structure of the optimal prefix code, *how would you label the leaf nodes?*






## Combining Our Conclusions

The binary tree corresponding to the optimal prefix code is *full*, i.e., each internal node has two children

We want to label the leaf nodes of the binary tree corresponding to the optimal prefix code such that nodes with *greatest* depth have *least* frequency

What does this mean the bottom of our tree looks like?



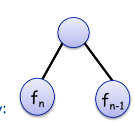
## Combining Our Conclusions

The binary tree corresponding to the optimal prefix code is *full*, i.e., each internal node has two children

We want to label the leaf nodes of the binary tree corresponding to the optimal prefix code such that nodes with *greatest* depth have *least* frequency

What does this mean the bottom of our tree looks like?

*2* letters with least frequency:

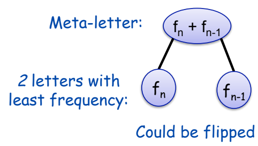



Could be flipped

## How Can We Use This?

Two letters with least frequency are definitely going to be siblings

- Tie them together
- Their parent is a "meta-letter"
  - Frequency is sum of $f_n + f_{n-1}$






## Constructing an Optimal Prefix Code

Huffman's Algorithm:

```
To construct a prefix code for an alphabet S, with given frequencies:
  If S has two letters then
    Encode one letter using 0 and the other letter using 1
  Else
    Let y* and z* be the two lowest-frequency letters
    Form a new alphabet S' by deleting y* and z* and
      replacing them with a new letter ω of frequency f_y* + f_z*
    Recursively construct a prefix code γ' for S', with tree T'
    Define a prefix code for S as follows:
      Start with T'
      Take the leaf labeled ω and add two children below it
        labeled y* and z*
  Endif
```

Reduce

Build up

Replace lowest-freq letters with meta letter



## Alternative Description

Create a leaf node for each symbol, labeled by its frequency, and add to a queue

While there is more than one node in the queue

- Remove the two nodes of lowest frequency
- Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' probabilities
- Add the new node to the queue

The remaining node is the tree's root node



## Creating the Optimal Prefix Code

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$



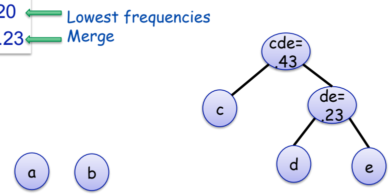
## Creating the Optimal Prefix Code

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$ ⟸ Lowest frequencies
$f_e = .05$ ⟸ Merge



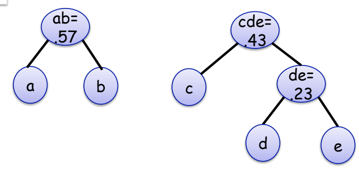

## Creating the Optimal Prefix Code

$f_a = .32$
$f_b = .25$
$f_c = .20$ ⟸ Lowest frequencies
$f_{de} = .23$ ⟸ Merge



## Creating the Optimal Prefix Code

$f_a = .32$ ← Lowest frequencies
$f_b = .25$ ← Merge
$f_{cde} = .43$

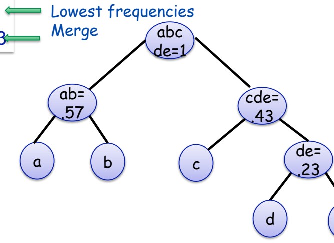





## Creating the Optimal Prefix Code

$f_{ab} = .57$ ← Lowest frequencies
$f_{cde} = .43$ ← Merge

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$




What are the resulting encodings?
What is the ABL?

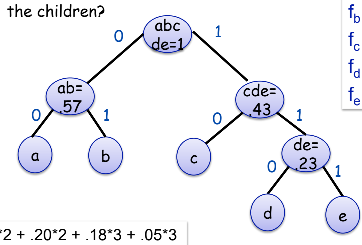


## Creating the Optimal Prefix Code

```
a: 00
b: 01
c: 10
d: 110
e: 111
```

I chose to build the tree this way. What if I had switched the order of the children?

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$




ABL=.32*2 + .25*2 + .20*2 + .18*3 + .05*3
= .64 + .5 + .4 + .54 + .15
= 2.23



## Implementation

What are the data structures we need?



## Implementation

What are the data structures we need?

- Binary tree for the prefix codes
- Priority queue for choosing the node with lowest frequency

Where are the costs?



## Next Time: Analysis

Can we prove that the solution is optimal?

Reminders:

- Problem Set, Friday at 5 p.m. under my door or in my mailbox
- SSA EC Opportunities