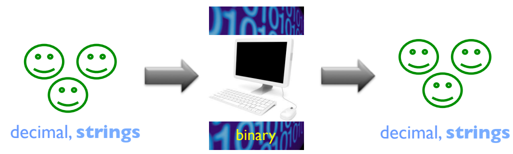
Objectives

- Data Compression

Review: Encoding Problem

- 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*

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

Example: Calculating ABL

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

```
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 * 2
- = 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

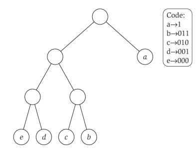
- Given an alphabet and a set of frequencies for the letters, produce optimal (most efficient) prefix code
  - Minimizes average # of bits per letter (ABL)

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

Binary Trees to Represent Prefix Codes

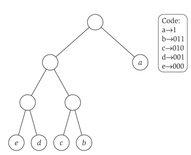
- Exposes structure better than list of mappings
  - Each leaf node is a letter
  - Follow path to the letter
    - Going left: 0
    - Going right: 1



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

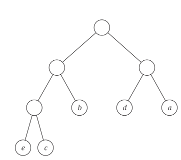
Code:
a→1
b→011
c→010
d→001
e→000

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

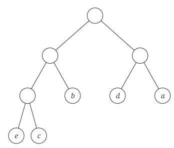
Tree Properties

- What is the length of a letter's encoding?
- Define our optimal goal in tree terms



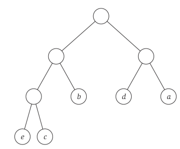
Tree Properties

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



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?

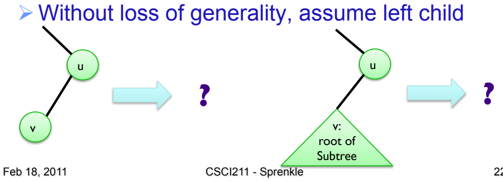


Tree Properties

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

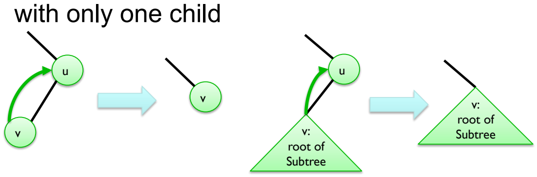
Tree Properties

- 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



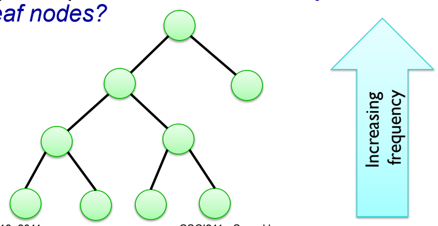
Replace *u* with *v* → decrease depth → original wasn't optimal

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?*



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

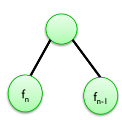
What does this mean the bottom of our tree looks like?

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:



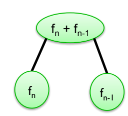
Could be flipped

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 $f_n + f_{n-1}$

Meta-letter:

2 letters with least frequency:



Could be flipped

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:
   Encode one letter as 0 and the other letter as 1

else:
   Let y* and z* be the two lowest-frequency letters
   Form a new alphabet S' by deleted y* and z* and replacing
      them with a new letter w of freq f_y* + f_z*
   Recursively construct a prefix code y' for S' with tree T'
   Define a prefix code for S as follows:
      Start with T'
      Take the leaf labeled w and add two children below it
        labeled y* and z*
```

Replace lowest-freq letters with meta letter

Reduce

Build up

Alternative Description

1. Create a leaf node for each symbol, labeled by its frequency, and add to a queue
2. While there is more than one node in the queue
   a) Remove the two nodes of lowest frequency
   b) Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' probabilities
   c) Add the new node to the queue
3. The remaining node is the tree's root node

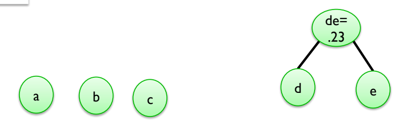
Creating the Optimal Prefix Code

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

Creating the Optimal Prefix Code

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

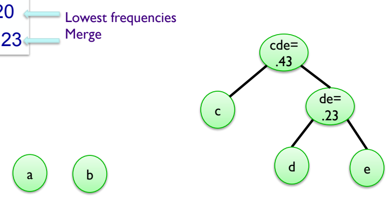
Lowest frequencies
Merge



Creating the Optimal Prefix Code

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_{de} = .23$

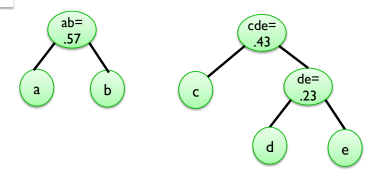
Lowest frequencies
Merge



Creating the Optimal Prefix Code

$f_a = .32$
$f_b = .25$
$f_{cde} = .43$

Lowest frequencies
Merge

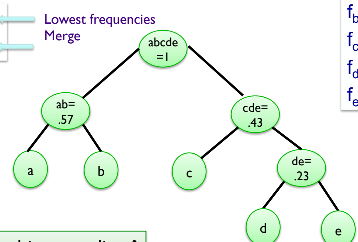


Creating the Optimal Prefix Code

$f_{ab} = .57$
$f_{cde} = .43$

Lowest frequencies
Merge

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

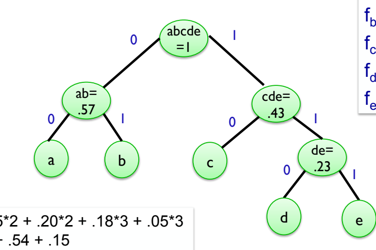


What are the resulting encodings?
What is the ABL?

Creating the Optimal Prefix Code

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

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



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

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

Implementation

- What data structures do we need?

Implementation

- What data structures do we need?
  - Binary tree for the prefix codes
  - Priority queue for choosing the node with lowest frequency
- Where are the costs?

Running Time

- Costs
  - Inserting and extracting node into PQ: O(log n)
  - Number of insertions and extractions: O(n)
  - O(n log n)

Analysis of Algorithm's Optimality

- 2 page proof in book

Real-life Compression

- Text can be compressed well because of known frequencies
- Algorithms can be optimized to languages
  - More than just "z doesn't happen very often"
    - "z doesn't happen after q"

Assignments

- Wiki due Wednesday
  - 4.5-4.8
- PS5 due Friday after break

Assignments

- Read, summarize Chapter 4.5-4.8
- PS5 due Friday after break
  - No class because of SSA (EC opportunity)
  - Submit somehow

Feb 18, 2011 CSCI211 - Sprenkle 43