Objectives

- Review Huffman Codes
- Introducing Divide and Conquer Algorithms

Towards Huffman Codes

- What problem are we trying to solve?
- Binary 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

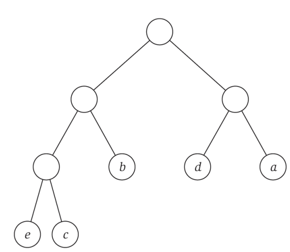
Given the mapping, how do you build the binary tree for this mapping?

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

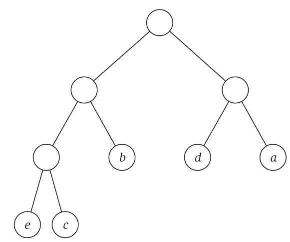
Tree Properties

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



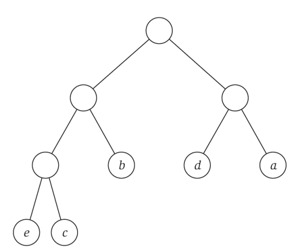
Tree Properties

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



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?

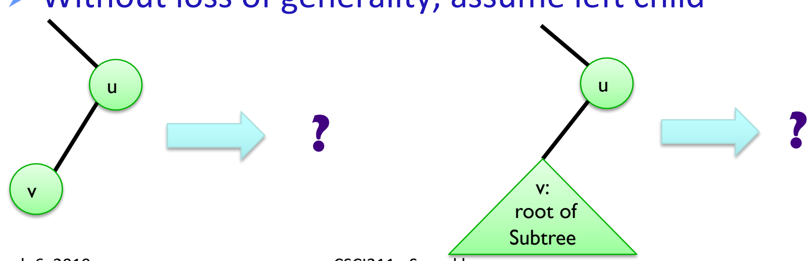


Tree Properties

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

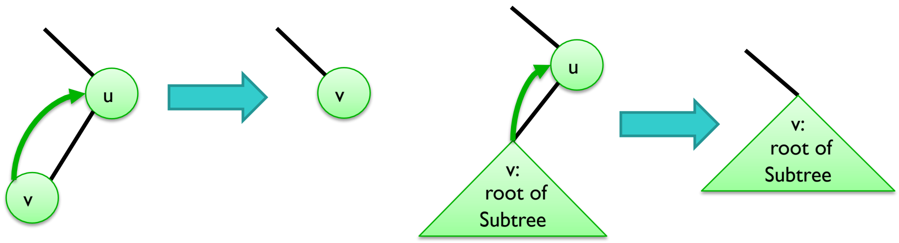
Tree Properties

- 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



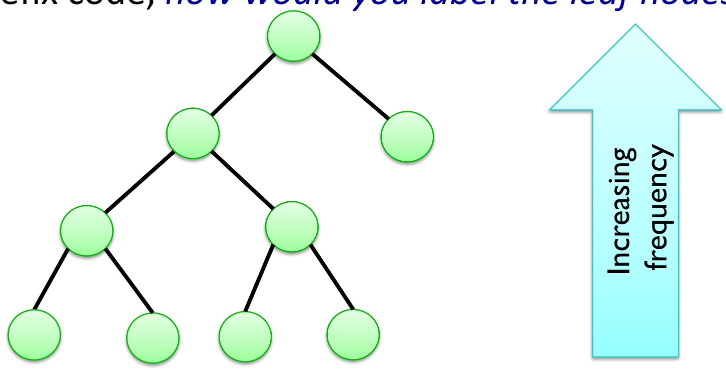
Replace *u* with *v* → decrease depth → original wasn't optimal

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

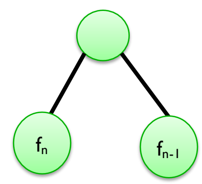
What does this mean the bottom of our tree should look like?

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 should look like?

2 letters with least frequency:



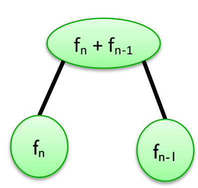
Could be flipped

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 fy* + fz*
   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

Constructing an Optimal Prefix Code: 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

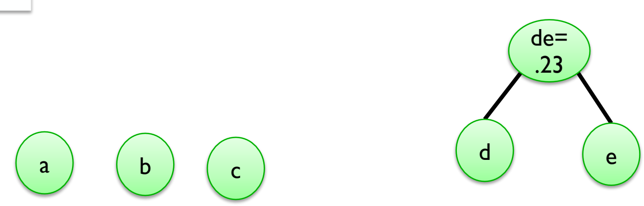
Creating the Optimal Prefix Code

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

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

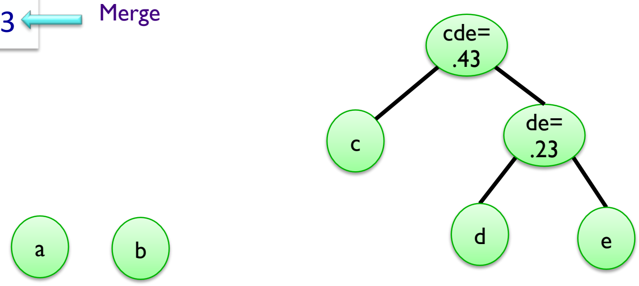
$f_a = .32$

$f_b = .25$

$f_c = .20$

$f_{de} = .23$

Lowest frequencies
Merge



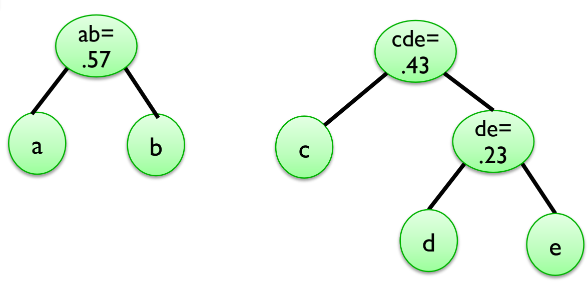
Creating the Optimal Prefix Code

$f_a = .32$

$f_b = .25$

$f_{cde} = .43$

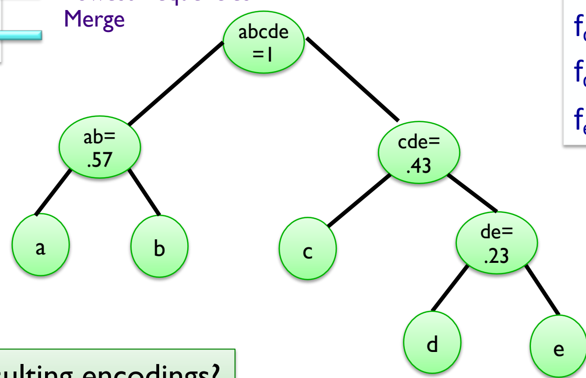
Lowest frequencies
Merge



Creating the Optimal Prefix Code

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

Lowest frequencies
Merge

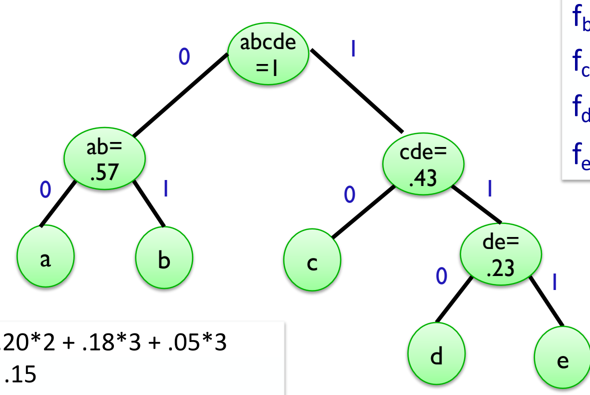


f_a = .32
f_b = .25
f_c = .20
f_d = .18
f_e = .05

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”

DIVIDE AND CONQUER ALGORITHMS

Divide-and-Conquer

Divide et impera.
Veni, vidi, vici.
- *Julius Caesar*

- Divide-and-conquer process
  - **Break up** problem into **several parts**
  - Solve each part **recursively**
  - **Combine** solutions to sub-problems into overall solution
- Most common usage:
  - Break up problem of size n into two equal parts of size ½n
  - Solve two parts recursively
  - Combine two solutions into overall solution

Discussion

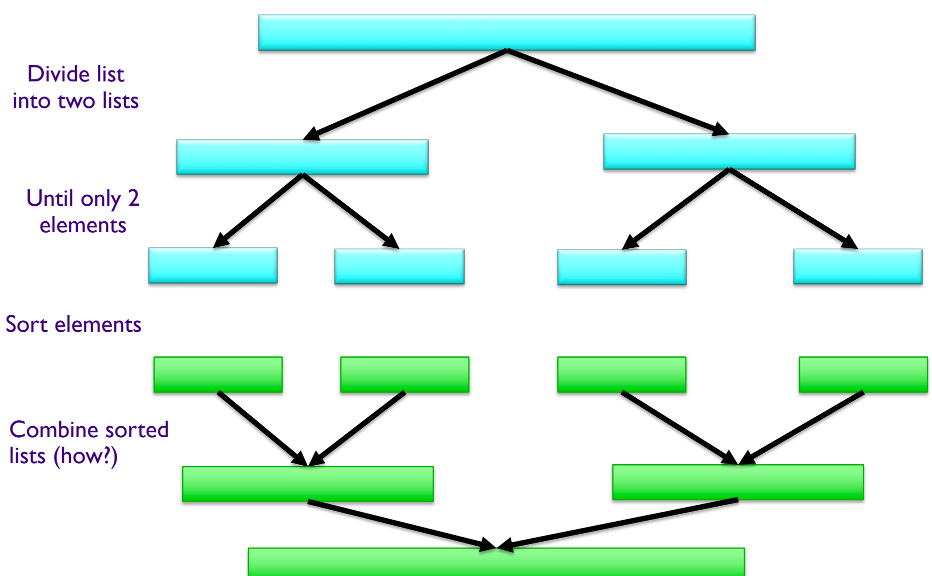
- What is a well-known divide and conquer algorithm?

Merge Sort

Merge Sort

- How does Merge Sort work?
- When do we stop?

Merge Sort



RECURRENCE RELATIONS

Analyzing Merge Sort

General Template

- Break up problem of size n into two equal parts of size ½n
- Solve two parts recursively
- Combine two solutions into overall solution

- **Def.** T(n) = number of comparisons to mergesort an input of size *n*
- Want to say a bit more about what T(n) is
  - Break it down more…

What can we say about the running time w.r.t. to the different parts of the above template?

Analyzing Merge Sort

General Template

- Break up problem of size n into two equal parts of size ½n **O(1)**
- Solve two parts recursively **T(n/2) + T(n/2)**
- Combine two solutions into overall solution **O(n)**

- **Def.** T(n) = number of comparisons to mergesort an input of size *n*
- Want to say a bit more about what T(n) is
  - Break it down more…

What is the base case? Its running time?

Merge Sort's Recurrence Relation

```
MergeSort( L[1…n] ):
     if len(L) == 1:
          return L          Base cases
     if len(L) == 2:
          compare the two entries in L,
          swap if necessary
          return L
     A = MergeSort(L[:n/2])    T(n/2)
     B = MergeSort(L[n/2+1:])  T(n/2)
     M = Merge(A, B)           O(n)
     return M
```

T(n) = 2T(n/2) + O(n)

Looking Ahead

- Problem Set 6 due Friday