Objectives

- Wrap up Huffman Codes
- Divide and Conquer Algorithms

Review

- What was the problem we were trying to solve on Monday?
  - What was our optimization goal?
- What was our solution to the problem?

Review: Problem and Goal

- 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*

Review: 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

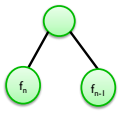
What does this mean the bottom of our tree looks like?

Review: 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:

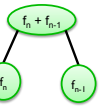


Could be flipped

Review: 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:



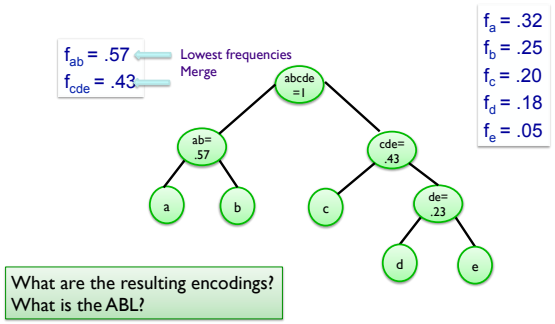
2 letters with least frequency:

Could be flipped

Review: Huffman Coding Algorithm

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

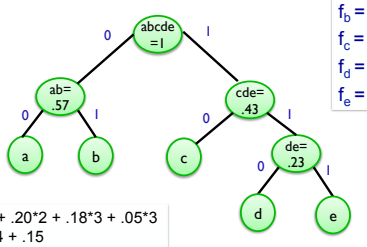
Creating the Optimal Prefix Code



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?

Implementing Huffman Coding Algorithm

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

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

Analyzing Runtime of Huffman Coding Algorithm

Where are the costs?

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

Running Time

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

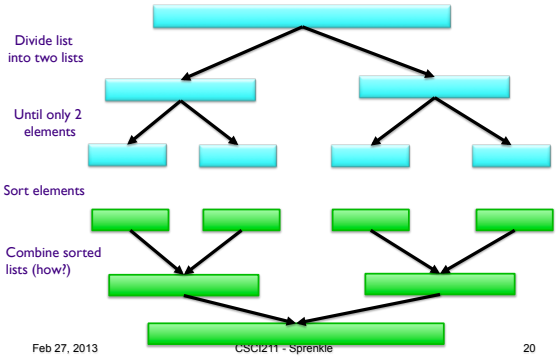
- What is a well-known divide and conquer algorithm?

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

- Put an *upperbound* on T(n):

O(n)

For some constant c,
$T(n) \leq 2\,T(n/2) + cn$ when $n > 2$,
$T(2) \leq c$

Solve T(n) to come up with explicit bound

Approaches to Solving Recurrences

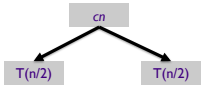
1. Unroll recursion
   - Look for patterns in runtime at each level
   - Sum up running times over all levels
2. Substitute guess solution into recurrence
   - Check that it works
   - Induction on n

Unrolling Recurrence: T(n)

T(n) = 2 T(n/2) + *cn*

Unrolling Recurrence: 2 T(n/2) + cn

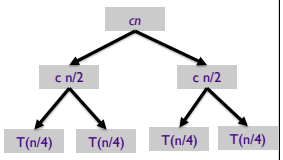
- First level: 2 T(n/2) + *cn*



How does the next level break down?

Unrolling Recurrence: 2 T(n/2) + cn

- Next level:



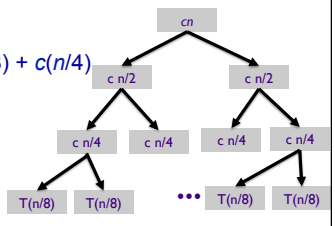
Each one is 2 T(n/4) + *c*(*n*/2)

Next level?

Unrolling Recurrence

- Next level:

  Each one is 2 T(n/8) + *c*(*n*/4)

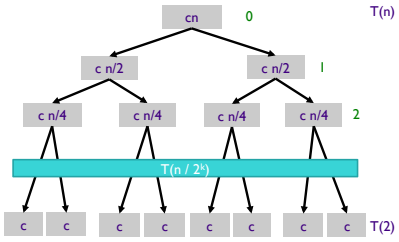


And so on…

What does the final level look like?

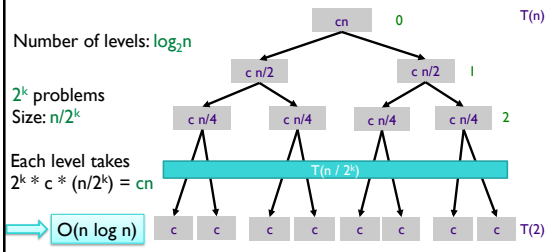
Unrolling Recurrence

- How much does each level cost, in terms of the level?
- How many levels are there (assuming *n* is a power of 2)?
- What is the total run time?



Unrolling Recurrence

- How many levels are there (assuming *n* is a power of 2)?
- How much does each level cost, in terms of the level?
- What is the total run time?



Alternative: Proof by Induction

- Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.
  - Recall: $T(n) = 2\,T(n/2) + cn$
- Pf. (by induction on n)
  - Base case: n = 2
  - Inductive hypothesis: $T(n) \le cn \log_2 n$
  - Goal: show that $T(2n) = 2cn \log_2 (2n)$

Why doubling n?

Proof by Induction

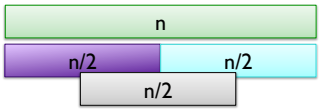
- Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.
  - Recall: $T(n) = 2\,T(n/2) + cn$
- Pf. (by induction on n)
  - Inductive hypothesis: $T(n) \le cn \log_2 n$
  - Goal: show that $T(2n) = 2cn \log_2 (2n)$

$T(2n) = 2T(n) + c2n$
$= 2cn \log_2 n + 2cn$ Replace w/ induction hypothesis
$= 2cn (\log_2(2n)-1) + 2cn$
$= 2cn \log_2(2n) - 2cn + 2cn$
$= 2cn \log_2(2n)$ ✔

Another Recurrence Relation

- Instead of recursively solving 2 problems, solve *q* problems
  - Size of problems is still n/2
- Combining solutions is still O(n)

Example: q=3:



What is the recurrence relation?

Another Recurrence Relation

- Instead of recursively solving 2 problems, solve *q* problems
  - Size of problems is still n/2
- Combining solutions is still O(n)
- Recurrence relation:
  - For some constant *c*,

$T(n) \le q\,T(n/2) + cn$ when $n > 2$
$T(2) \le c$

Intuition about running time?

Looking Ahead

- Problem Set 5 due Friday