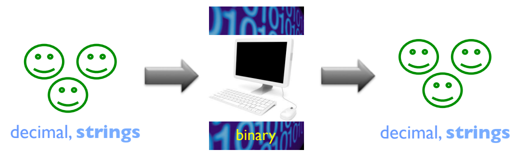
Objectives

- 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

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

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)

Review: Building the Binary Tree

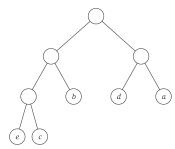
- How do we build the binary tree for this mapping?

Code:
a→1
b→011
c→010
d→001
e→000

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

Tree Properties

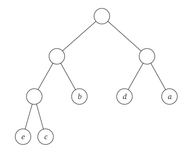
- What is the length of a letter's encoding?
- Define our optimal goal in tree terms



Feb 29, 2012 CSCI211 - Sprenkle 7

Tree Properties

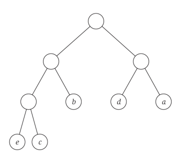
- What is the length of a letter's encoding?
  - Length of path from root to leaf → its *depth*
- Define our optimal goal in tree terms
  - **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)| = \Sigma_{x \in S} f_x \, \text{depth}(x)$



Feb 29, 2012 CSCI211 - Sprenkle 8

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?



Feb 29, 2012 CSCI211 - Sprenkle 9

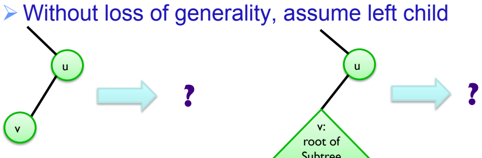
Tree Properties

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

Feb 29, 2012 CSCI211 - Sprenkle 10

Tree Properties

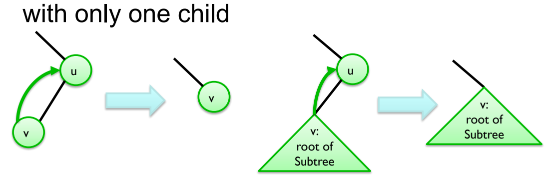
- Claim. The binary tree *T* corresponding to the optimal prefix code is *full*, i.e., each internal node has two children.
- Proof. Assume that *T* has an internal node with only one child
  - Without loss of generality, assume left child



Feb 29, 2012 CSCI211 - Sprenkle 11

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

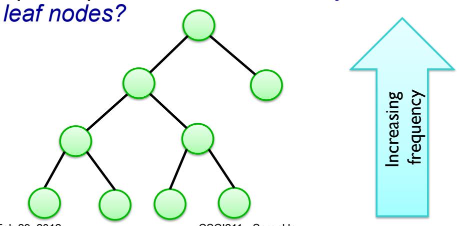
Feb 29, 2012 CSCI211 - Sprenkle 12

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?*



Drawing Conclusions from 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

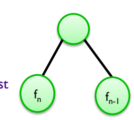
What does this mean the bottom of our tree looks like?

Drawing Conclusions from 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:



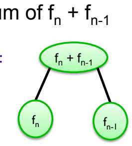
Could be flipped

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 deleting 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

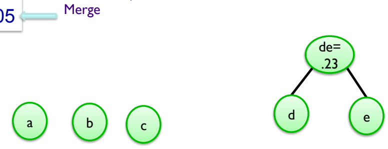
Creating the Optimal Prefix Code

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

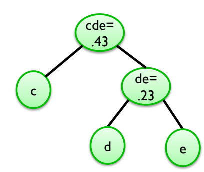
Lowest frequencies
Merge



Creating the Optimal Prefix Code

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

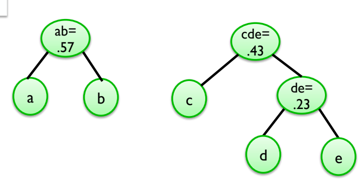
Lowest frequencies
Merge



Creating the Optimal Prefix Code

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

Lowest frequencies
Merge

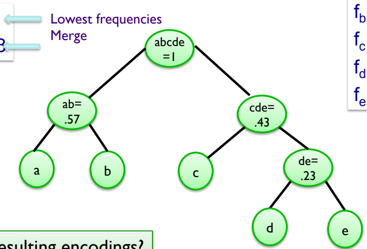


Creating the Optimal Prefix Code

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

Lowest frequencies
Merge

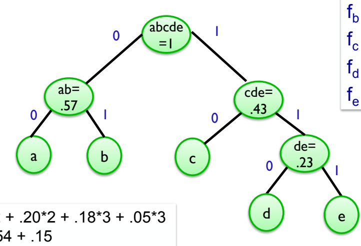
f_a = .32
f_b = .25
f_c = .20
f_d = .18
f_e = .05



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?

Alternative Description

What data structures do we need?

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

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?

Alternative Description

What 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

Alternative Description

1. Create a leaf node for each symbol, labeled by its frequency, and add to a queue O(nlog n)
2. While there is more than one node in the queue O(n)
   a) Remove the two nodes of lowest frequency O(log n)
   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 O(log n)
3. The remaining node is the tree's root node

Total: O(n logn)

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

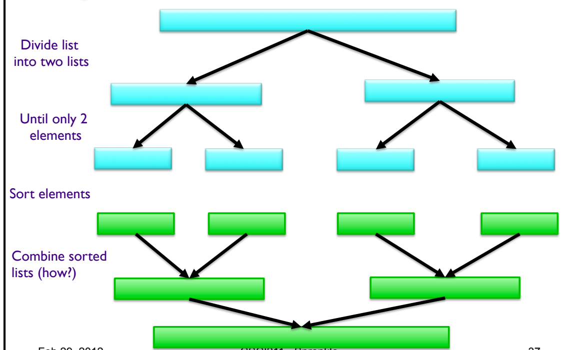
- What is a well-known divide and conquer algorithm?

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

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

For some constant c,
$T(n) \le 2\,T(n/2) + \underbrace{cn}_{O(n)}$ when $n > 2$,
$T(2) \le c$

Solve T(n) to come up with explicit bound

Approaches to Solving Recurrences

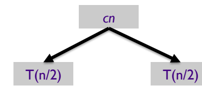
1. Unroll recursion
   - Look for patterns in runtime at each level
   - Sum up running times over all levels
2. Substitute guess solution into recurrence
   - Check that it works
   - Induction on n

Unrolling Recurrence: T(n)

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

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

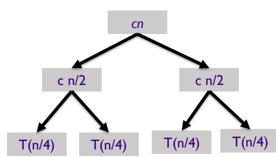
- First level: 2 T(n/2) + *cn*



How does the next level break down?

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

- Next level:



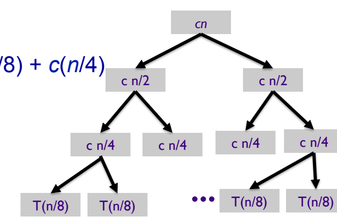
Each one is 2 T(n/4) + *c*(*n*/2)

Next level?

Unrolling Recurrence

- Next level:

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

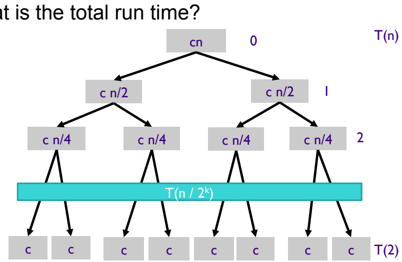


And so on…

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?



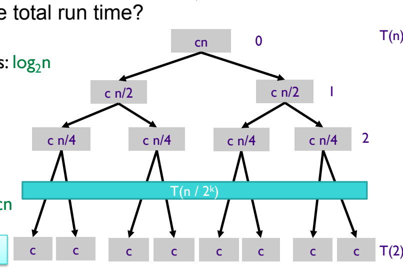
Unrolling Recurrence

- 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?

Number of levels: $\log_2 n$

2^k problems
Size: $n/2^k$

Each level takes
$2^k * c * (n/2^k) = cn$



O(n log n)

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 $T(2n) = 2cn \log_2 (2n)$

$$\begin{aligned} T(2n) &= 2T(n) + c2n \\ &= 2cn \log_2 n + 2cn \quad \text{Replace w/ induction hypothesis} \\ &= 2cn (\log_2(2n)-1) + 2cn \\ &= 2cn \log_2(2n) - 2cn + 2cn \\ &= 2cn \log_2(2n) \checkmark \end{aligned}$$

Looking Ahead

- Wiki due tonight
- Problem Set 5 due Monday in class
- Next Wiki: Chapter 4.7-4.8, Chapter 5
  - Due next Wednesday
- Problem Set 6 due next Friday