Objectives

- Wrap up Data Compression/Huffman Codes
- Divide and conquer
  - Recurrence relations

Review: 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 00100000011101?

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


Review: Optimal Prefix Codes

- **Goal**: minimize **Average number of Bits per Letter (ABL)**:

  $\Sigma_{x \in S}$ frequency of x * length of encoding of x

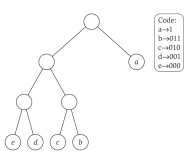
  ↑ For all characters in our alphabet
- f_x: frequency that letter *x* occurs
- γ(x): encoding of *x*
  - |γ(x)|: length of encoding of *x*
- Minimize **ABL** = $\Sigma_{x \in S} f_x |\gamma(x)|$

Review: 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

Review: 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

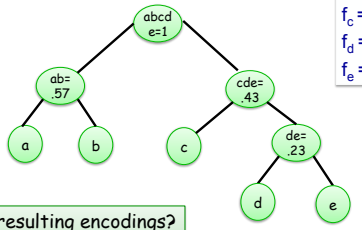


Review: Huffman Algorithm

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

Example: Creating the Optimal Prefix Code

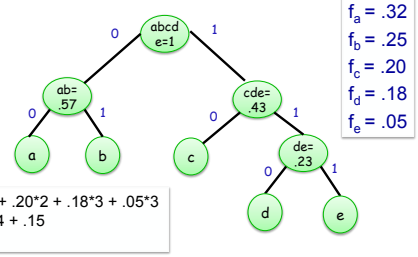
$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$



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

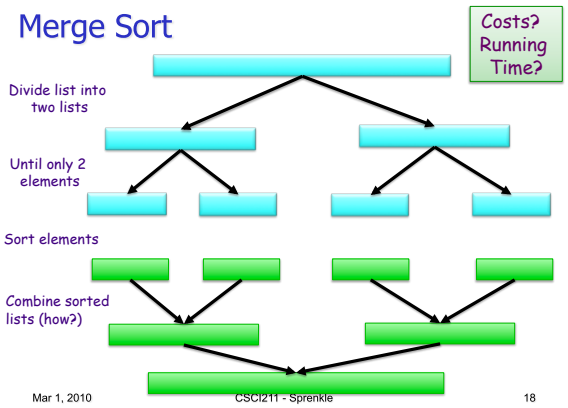
- What is a well-known divide and conquer algorithm?

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

What is the base case's running time?

Merge Sort's Recurrence Relation

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

O(n)

For some constant c,
T(n) ≤ 2 T(n/2) + *cn* when n > 2,
T(2) ≤ c.

Why is this constant?

Solve T(n) to come up with explicit bound

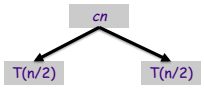
Approaches to Solving Recurrences

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)

Unrolling Recurrence

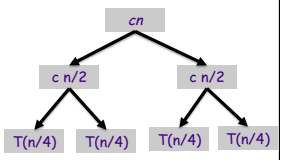
- First level: 2 T(n/2) + *cn*



How does the next level break down?

Unrolling Recurrence

- Next level:



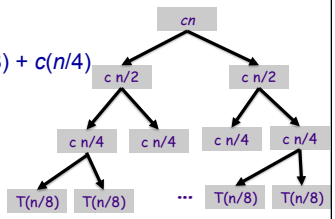
Each one is 2 T(n/4) + *c*(*n*/2)

Next level?

Unrolling Recurrence

- Next level:

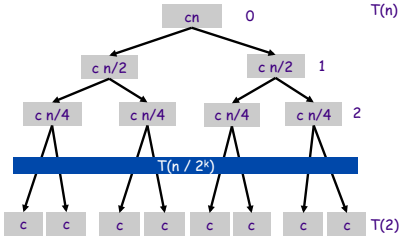
  Each one is 2 T(n/8) + *c*(*n*/4)



And so on…

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?



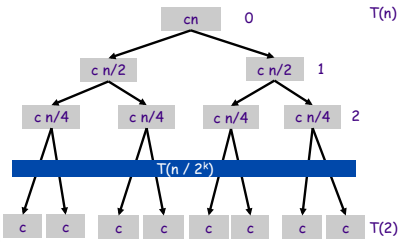
Unrolling Recurrence

- 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.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Pf. (by induction on n)
  - Base case: n = 1
  - Inductive hypothesis: T(n) = n $\log_2$ n
  - Goal: show that T(2n) = 2n $\log_2$ (2n)

Why doubling n?

Proof by Induction

- Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Pf. (by induction on n)
  - Inductive hypothesis: T(n) = n $\log_2$ n

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\log_2 n + 2n \\ &= 2n(\log_2(2n) - 1) + 2n \\ &= 2n\log_2(2n) \end{aligned}$$

Another Example

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

Another Example

- 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) ≤ *q* T(n/2) + *cn* when n > 2

    T(2) ≤ *c*

Intuition about running time?

Assignments

- Continue reading Chapter 4
- Start reading Chapter 5
- Wiki
  - Read 4.6-4.8, 5.1
- PS5 due Friday