NPTEL MOOC,JAN-FEB 2015
Week 6, Module 5

# DESIGN AND ANALYSIS OF ALGORITHMS

## Greedy algorithms: Huffman codes

**MADHAVAN MUKUND, CHENNAI MATHEMATICAL INSTITUTE**
**http://www.cmi.ac.in/~madhavan**

# Communication and compression

- Messages in English/Hindi/Tamil/… are transmitted between computers in binary
- Encode letters {a,b,…,z} as strings over {0,1}
    - 26 letters, $2^5 = 32$, use strings of length 5?
- Can we optimize the amount of data to transfer?
    - Use shorter strings for more frequent letters?

# Variable length encoding

## Morse code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01
- Decode 0101 — etet, aa, eta, aet?
- Use pauses between letters to distinguish
    - Like an extra symbol in encoding

# Variable length encoding

## Prefix code

- Encoding E( ), E(x) is not a prefix of E(y) for any x,y
  - In Morse code E(e) = 0 is a prefix of E(a) = 01
- Example: {a,b,c,d,e}

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |

- Decode 0010000011101

# Variable length encoding

Prefix code

- Encoding E( ), E(x) is not a prefix of E(y) for any x,y
  - In Morse code E(e) = 0 is a prefix of E(a) = 01
- Example: {a,b,c,d,e}

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |

- Decode 001|0000011101

c

# Variable length encoding

## Prefix code

- Encoding E( ), E(x) is not a prefix of E(y) for any x,y
  - In Morse code E(e) = 0 is a prefix of E(a) = 01
- Example: {a,b,c,d,e}

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |

- Decode 001|000|0011101

  c e

# Variable length encoding

## Prefix code

- Encoding E( ), E(x) is not a prefix of E(y) for any x,y
  - In Morse code E(e) = 0 is a prefix of E(a) = 01
- Example: {a,b,c,d,e}

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |

- Decode 001|000|001|1101

  c e c

# Variable length encoding

## Prefix code

- Encoding E( ), E(x) is not a prefix of E(y) for any x,y
  - In Morse code E(e) = 0 is a prefix of E(a) = 01
- Example: {a,b,c,d,e}

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |

- Decode 001|000|001|11|01

  c e c a

# Variable length encoding

## Prefix code

- Encoding E( ), E(x) is not a prefix of E(y) for any x,y
  - In Morse code E(e) = 0 is a prefix of E(a) = 01
- Example: {a,b,c,d,e}

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |

- Decode 001|000|001|11|01|

  c e c a b

# Optimal prefix codes

- Measure frequency f(x) of each letter x
  - Fraction of occurrences of x over large body of text
  - $A = \{x_1, x_2, \ldots, x_n\}$, $f(x_1) + f(x_2) + \ldots + f(x_n) = 1$
  - f(x) is the "probability" that next letter is x

# Optimal prefix codes …

- Message M consists of n symbols
  - For each letter x, $n \cdot f(x)$ occurrences of x in M
- Each x is encoded by $E(x)$ with length $|E(x)|$
- Total length of encoded message:
  - Sum over all x, $n \cdot f(x) \cdot |E(x)|$
- Average number of bits per letter
  - Sum over all x, $f(x) \cdot |E(x)|$

# Optimal prefix codes ..

- Suppose we have these frequencies for our example

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

- Average number of bits per letter is
  - $0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.18 \cdot 2 + 0.05 \cdot 3$
  - 2.25
- Fixed length encoding uses 3 bits per letter
  - 25% saving using variable length code

# Optimal prefix codes ..

- A better encoding

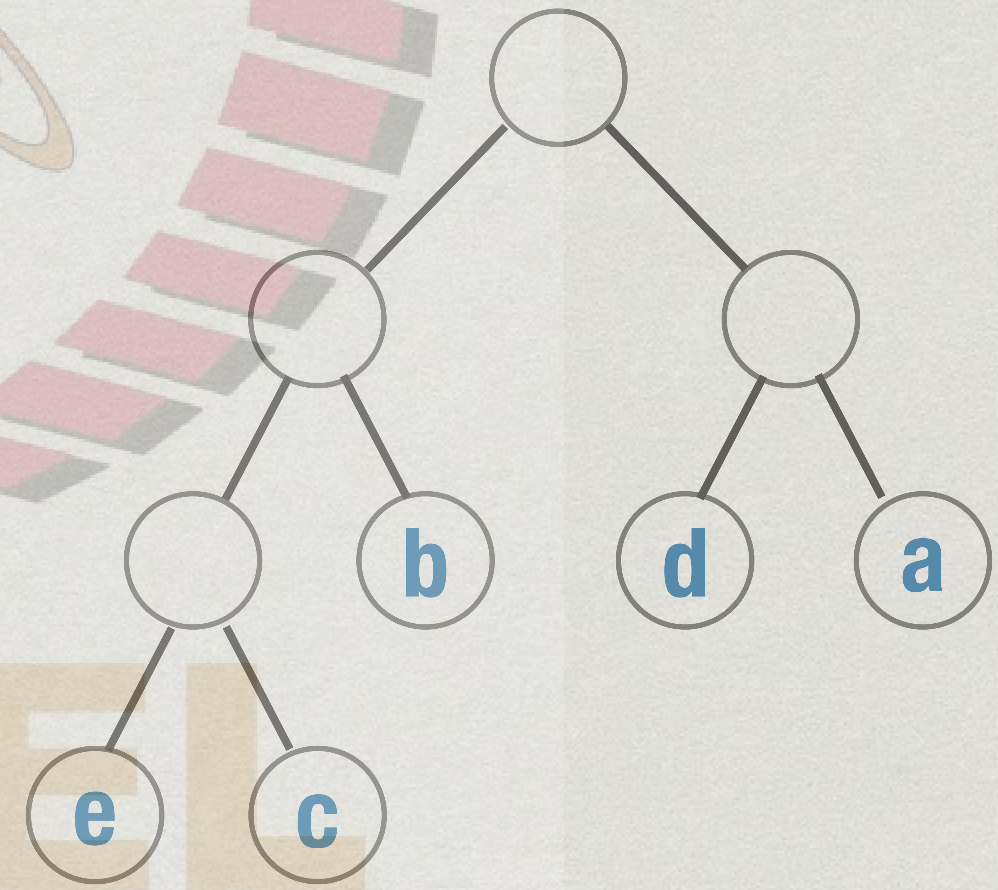
| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 10 | 01 | 001 | 000 |
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

- Average number of bits per letter is
  - $0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 2 + 0.18 \cdot 3 + 0.05 \cdot 3$
  - 2.23
- Given a set of letters A with frequencies, produce a prefix code that is as efficient as possible
  - Minimize ABL(A) — Average Bits per Letter

# Codes as trees

- Encoding can be viewed as a binary tree
- Path to a node is a binary string—left is 0, right is 1
- Label each node by the letter it encodes
- Prefix code: only leaves encode letters

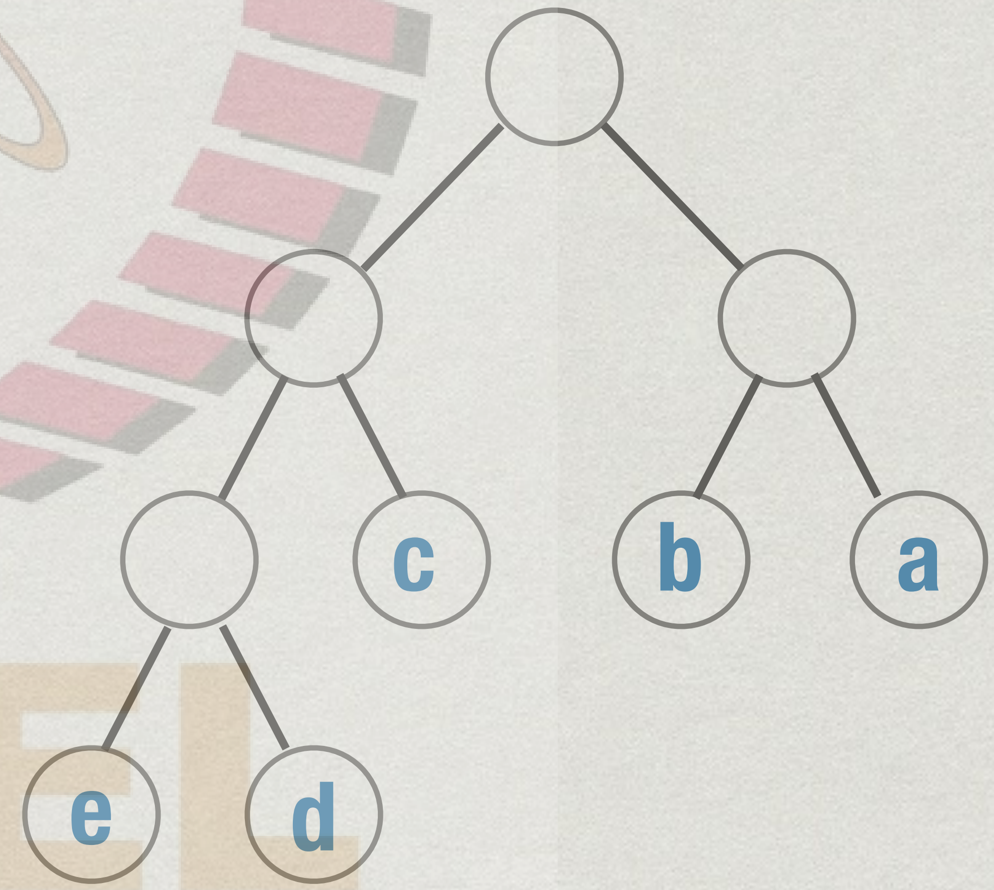
| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 01 | 001 | 10 | 000 |



# Codes as trees

- Encoding can be viewed as a binary tree
- Path to a node is a binary string—left is 0, right is 1
- Label each node by the letter it encodes
- Prefix code: only leaves encode letters

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| E(x) | 11 | 10 | 01 | 001 | 000 |



# Codes as trees …

- Full tree: Every node has 0 or 2 children

Claim 1: Any optimal prefix code generates a full tree

- If any node has only one child, we can promote its child and create a shorter tree

# Codes as trees …

Claim 2: In an optimal tree, if a leaf labelled x is at a smaller depth than a leaf labelled y, then $f(y) \leq f(x)$

- If $f(y) > f(x)$, exchange labels to get a better tree

# Codes as trees …

Claim 3: In an optimal tree, if a leaf at maximum depth is labelled x then its sibling is also a leaf.

- If not, the sibling of this leaf has children
- There is a leaf at a lower depth
- But depth of the leaf labelled x was at maximum depth

# A recursive solution

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick letters x and y such that f(x) and f(y) are lowest
- We will assign these to a pair of leaves at maximum depth (left/right does not matter)

# A recursive solution …

- “Combine” x and y into a new letter “xy” with f(xy) = f(x) + f(y)
- New alphabet A’ is original A - {x,y} + {xy}
- Recursively find an optimal encoding of A’
  - Base case, |A’| = 2, assign the two letters codes 0, 1
- Replace the leaf labelled “xy” by a node with two children labelled x and y
- Huffman’s algorithm — Huffman coding

# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

# Huffman's algorithm

| x | a | b | c | d | e |
| --- | --- | --- | --- | --- | --- |
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Combine d, e as "de"

| x | a | b | c | de |
| --- | --- | --- | --- | --- |
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Combine
d, e as "de"

| x | a | b | c | de |
|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

Combine
c, de as "cde"

| x | a | b | cde |
|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.43 |

# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Combine d, e as "de"

| x | a | b | c | de |
|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

Combine c, de as "cde"

| x | a | b | cde |
|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.43 |

Combine a, b as "ab"

| x | ab | cde |
|---|---|---|
| f(x) | 0.57 | 0.43 |

# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Combine
d, e as "de"

| x | a | b | c | de |
|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

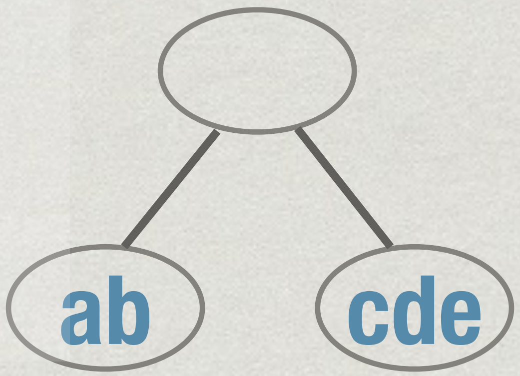
Combine
c, de as "cde"

| x | a | b | cde |
|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.43 |

Combine
a, b as "ab"

| x | ab | cde |
|---|---|---|
| f(x) | 0.57 | 0.43 |

Two letters,
base case



# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Combine
d, e as "de"

| x | a | b | c | de |
|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

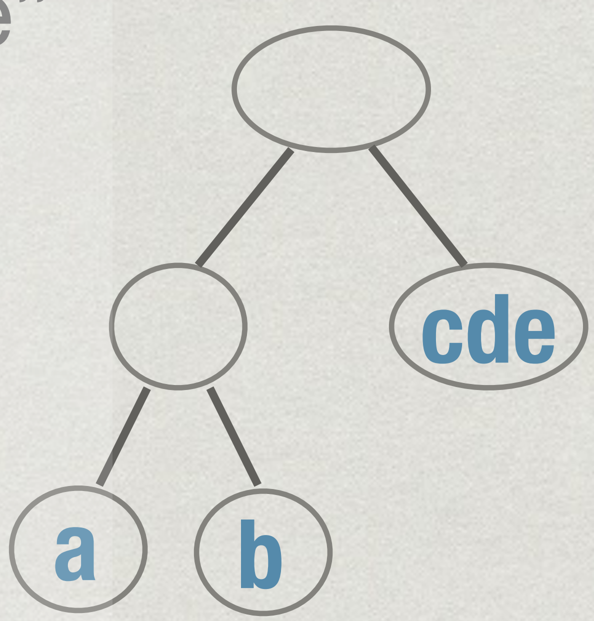
Combine
c, de as "cde"

| x | a | b | cde |
|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.43 |

Split "ab"
as a, b

| x | ab | cde |
|---|---|---|
| f(x) | 0.57 | 0.43 |

Two letters,
base case



# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Combine d, e as "de"

| x | a | b | c | de |
|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

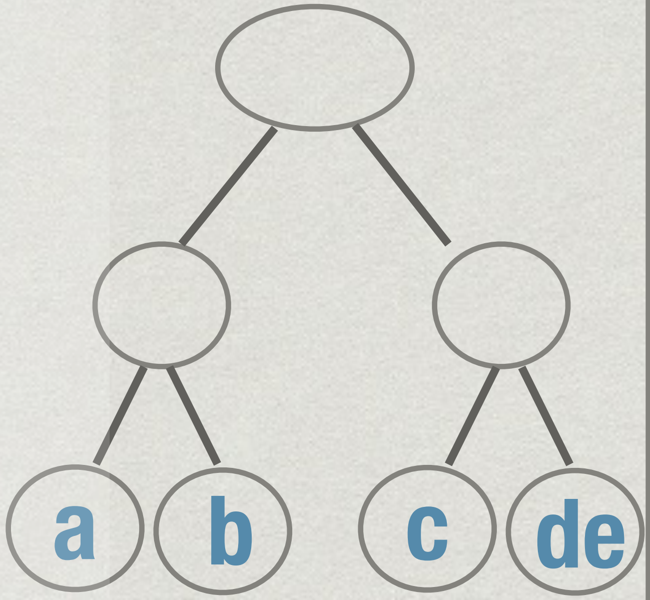
Split "cde" as c, de

| x | a | b | cde |
|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.43 |

Split "ab" as a, b

| x | ab | cde |
|---|---|---|
| f(x) | 0.57 | 0.43 |

Two letters, base case



# Huffman's algorithm

| x | a | b | c | d | e |
|---|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.18 | 0.05 |

Split "de" as d, e

| x | a | b | c | de |
|---|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.20 | 0.23 |

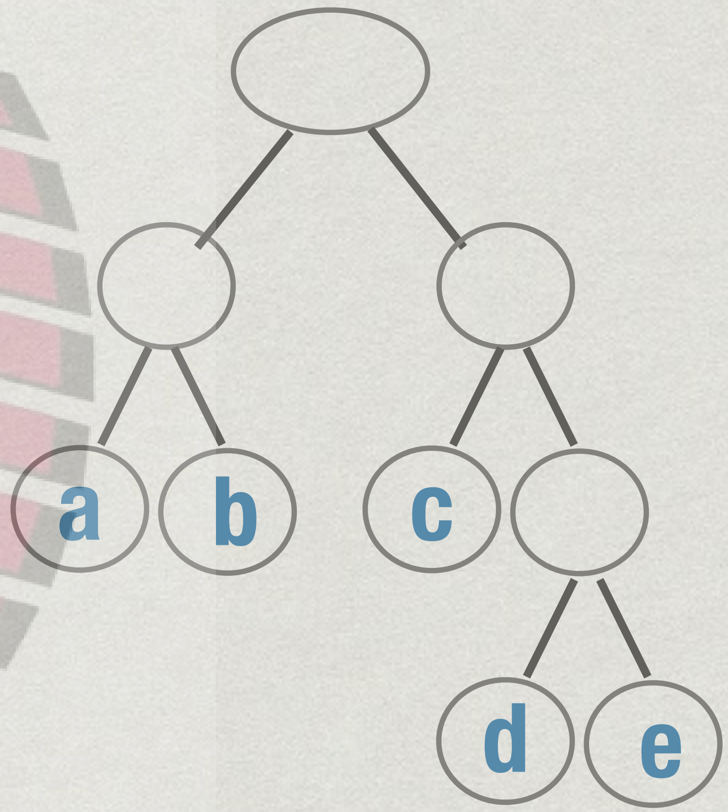
Split "cde" as c, de

| x | a | b | cde |
|---|---|---|---|
| f(x) | 0.32 | 0.25 | 0.43 |

Split "ab" as a, b

| x | ab | cde |
|---|---|---|
| f(x) | 0.57 | 0.43 |

Two letters, base case



# Optimality

- By induction on the size of the alphabet A
- For |A| = 2, base case, clearly the code that uses {0,1} for the two letters is optimal
- Assuming our algorithm is optimal for |A| = k-1, we have to show it is also optimal for |A| = k

# Optimality

- Combine lowest frequency x, y into xy
- Construct a tree T' for this alphabet
- ABL(T') optimal by induction
- Expand xy into x,y to get T from T'

Claim: ABL(T) - ABL(T') = f(xy)

# Optimality

Claim: ABL(T) - ABL(T') = f(xy)

* From T' to T, only xy, x, y change contribution to ABL
* Subtract depth(xy)f(xy), add (1+depth(xy))(f(x) + f(y))
* f(xy) = f(x)+f(y), so
  depth(xy)f(xy) = depth(xy)(f(x) + f(y))
* Hence ABL(T) is bigger than ABL(T') by
  f(x)+f(y) = f(xy)

# Optimality

- Suppose there is another tree S with ABL(S) < ABL(T)
- Can shuffle labels of max depth leaves in S, so that lowest frequency pair x and y label siblings
- Merge, x and y into xy and contract S to S’
- S’ is over same alphabet as T’, T’ is optimal by induction, so ABL(T’) ≤ ABL(S’)
- ABL(S) - ABL(S’) = ABL(T) - ABL(T’) = f(xy),
  so ABL(T) ≤ ABL(S) as well, contradiction!

# Implementation, complexity

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
  - Linear scan to find minimum values
  - $|A| = k$, number of recursive calls is $k - 1$
  - Complexity is $O(k^2)$

# Implementation, complexity

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Instead, maintain frequencies in a heap
  - O(log k) to find minimum values and insert new combined letter
  - Complexity drops to O(k log k)

# Why is Huffman coding greedy?

* We recursively combine letters with two lowest frequencies
* This is a locally optimal choice
* We never go back and consider other ways of pairing up letters

# Historical note

- Shannon and Fano tried a divide and conquer approach
  - Partition A as $A_1$, $A_2$
  - Sum of frequencies in $A_1$, $A_2$ roughly equal
  - Solve each partition recursively
  - Shannon-Fano codes are not optimal
- Huffman heard about this problem in a class by Fano and later found an optimal solution