Huffman Code

Greedy Algorithms

Outline

- Coding and Decoding
- The optimal source coding problem
- Human coding: A greedy algorithm

Encoding

Example

Suppose we want to store a given a $100,000$ character data file. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

- A binary code encodes each character as a binary string or codeword over some given alphabet Σ
  - a code is a set of codewords.
  - e.g., $\{000, 001, 010, 011, 100, 101\}$ and $\{0, 101, 100, 111, 1101, 1100\}$

  are codes over the binary alphabet $\Sigma = \{0, 1\}$.

Encoding

Given a code C over some alphabet it is easy to encode the message using C. Just scan through the message, replacing the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$

If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded as 01 00 11

If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded as 110 0 111

Fixed Length vs Variable Length Encoding

Total character = 100,000
Fixed Length = 3* 100,000
= 300,000

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

Note: since there are 6 characters, a fixed-length code must use at least 3 bits per codeword).

- The fixed-length code requires 300,000 bits to store the file.
- The variable-length code requires only $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$bits, saving a lot of space!

- Goal is to save space!

Decoding

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}.$$ non-ambiguous

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}.$$ non-ambiguous

$$C_3 = \{a = 1,\ b = 110,\ c = 10,\ d = 111\}$$ ambiguous

Not prefix free

Given an encoded message, decoding is the process of turning it back into the original message.

Message is uniquely decodable if it can be decoded in only one way.

Decode: 010011 according to C_1 = bad,

Decode: 1100111 according to C_2 = bad, acda,

Decode: 1101111 according to C_3 = bad, acda,

Decoding

$$
\begin{aligned}
C_1 &= \{a = 00,\ b = 01,\ c = 10,\ d = 11\}. \\
C_2 &= \{a = 0,\ b = 110,\ c = 10,\ d = 111\}. \\
C_3 &= \{a = 1,\ b = 110,\ c = 10,\ d = 111\}
\end{aligned}
$$

Given an encoded message, decoding is the process of turning it back into the original message.

Message is uniquely decodable if it can be decoded in only one way.

Example

Relative to C_1, 010011 is uniquely decodable to bad.

Relative to C_2, 1100111 is uniquely decodable to bad.

Relative to C_3, 1101111 is not uniquely decipherable

it could have encoded either bad or acad.

Unique Decipherability

In fact, *any* message encoded using C_1 or C_2 is uniquely decipherable. Unique decipherability property is needed in order for a code to be useful.

Character	a	b	c	d	e	f
Codes	0	01	10	101	011	110
Frequency	45	13	12	16	9	5

- Number of bits required for encoding 100,000 characters
- = 45*1+ 13*2+12*2+16*3+9*3+5*3 = 185,000 bits

- Decode 100110

Character	a	b	c	d	e	f
Codes	0	01	10	101	011	110
Frequency	45	13	12	16	9	5

- Number of bits required for encoding 100,000 characters
- = 45*1+ 13*2+12*2+16*3+9*3+5*3 = 185,000 bits

- Decode 100110

- c b c 10 , 01 , 10
- c a f 10, 0 , 110
- Which decoding is correct?
- **Ambiguous encoding scheme**, ambiguity lies in boundary of a code, where does a code end?

Prefix Codes

Fixed-length codes are always uniquely decipherable. WHY?

We saw before that fixed-length codes do not always give the best compression though, so we prefer to use variable length codes.

Definition

A code is called a prefix (free) code if no codeword is a prefix of another one.

Example

$\{a = 0,\ b = 110,\ c = 01,\ d = 111\}$ is *not* a prefix code.
$\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$ is a prefix code.

Prefix Codes

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Because no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 = 01101100 = abba$$

We are therefore interested in finding *good* (best compression) prefix-free codes.

Outline

- Coding and Decoding
- The optimal source coding problem
- Human coding: A greedy algorithm

The Optimal Source Coding Problem

Huffman Coding Problem

Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$, find a binary prefix code C for A that minimizes the number of bits

$$B(C) = \sum_{i=1}^{n} f(a_i) L(c_i)$$

needed to encode a message of $\sum_{i=1}^{n} f(a_i)$ characters, where

- c_i is the codeword for encoding a_i, and
- $L(c_i)$ is the length of the codeword c_i.

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store these 100 characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150 \text{ bits}$$

a 25% saving.

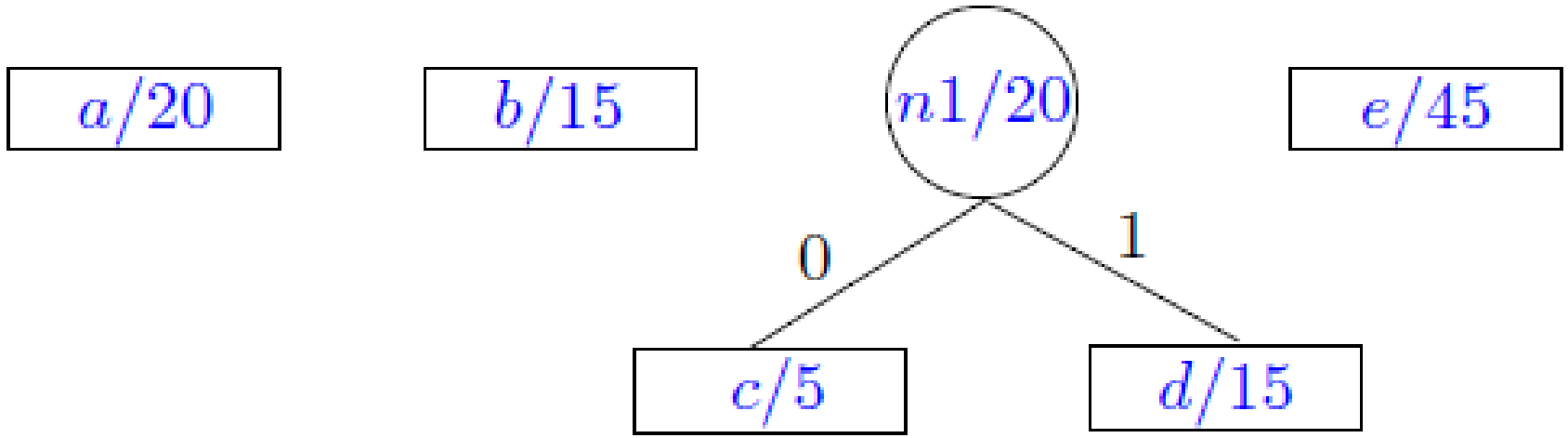
Outline

- Coding and Decoding
- The optimal source coding problem
- Huffman coding: A greedy algorithm

Example of Human Coding

Let $S = \{a/20, b/15, c/5, d/15, e/45\}$ be the original character set S alphabet and its corresponding frequency distribution.

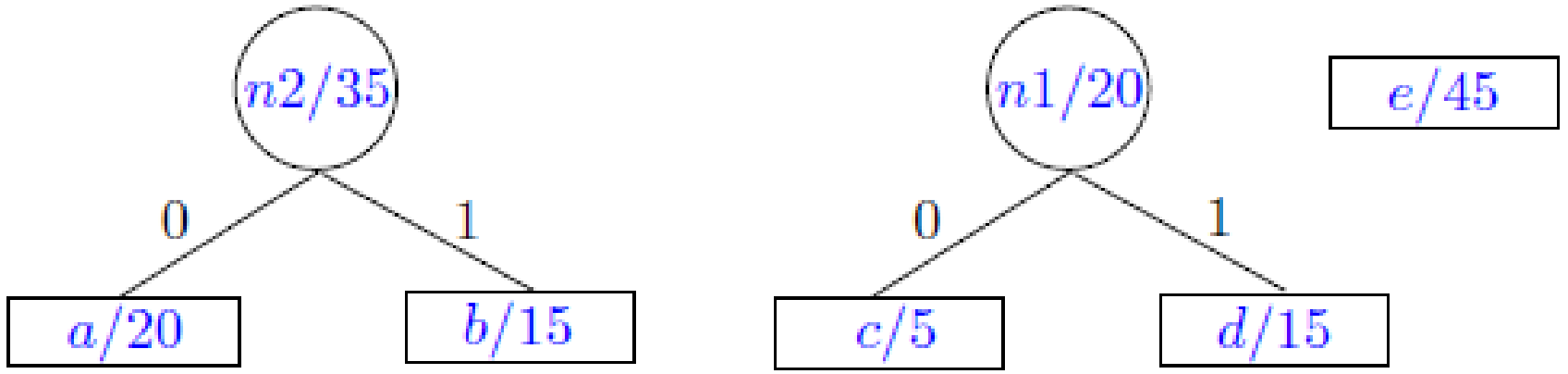
1. The first Huffman coding step merges c **and** d. (could also have merged c and b).



Now have $S = \{a/20, b/15, \mathbf{n1/20}, e/45\}$.

Example of Human Coding

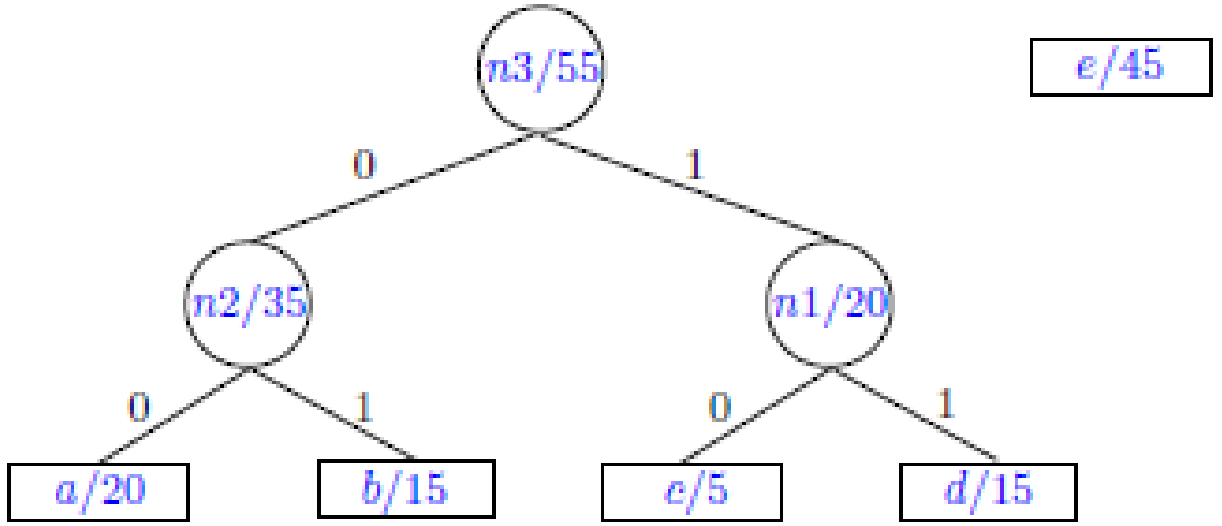
2. Algorithm merges a **and** b (could also have merged $n1$ and b)



Now have $S = \{n2/35, n1/20, e/45\}$.

Example of Human Coding

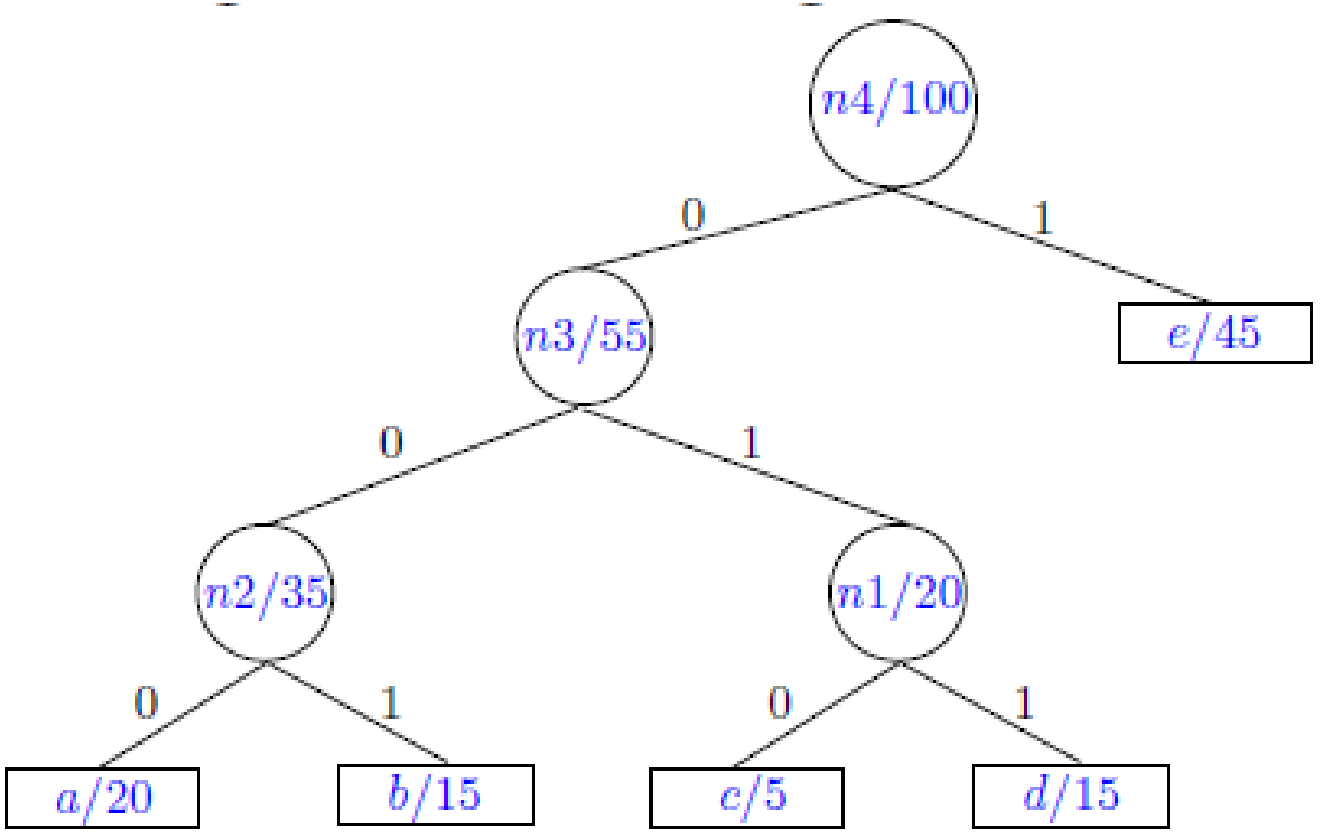
Algorithm merges $n1$ **and** $n2$.



Now have $S_3 = \{\mathbf{n3/55}, e/45\}$.

Example of Human Coding

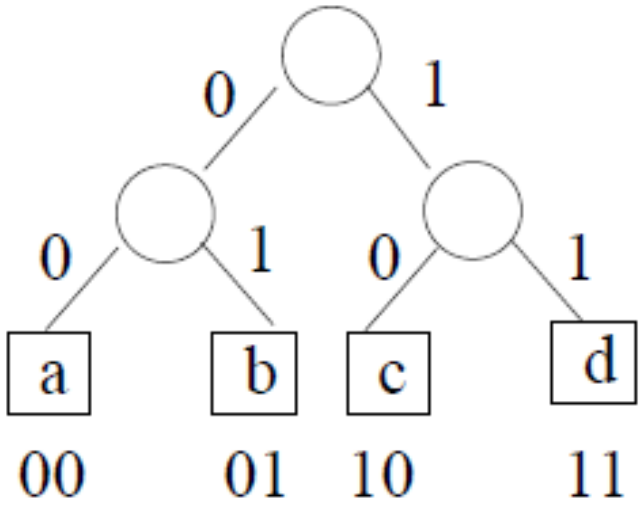
Algorithm next merges e **and** $n3$ and finishes.

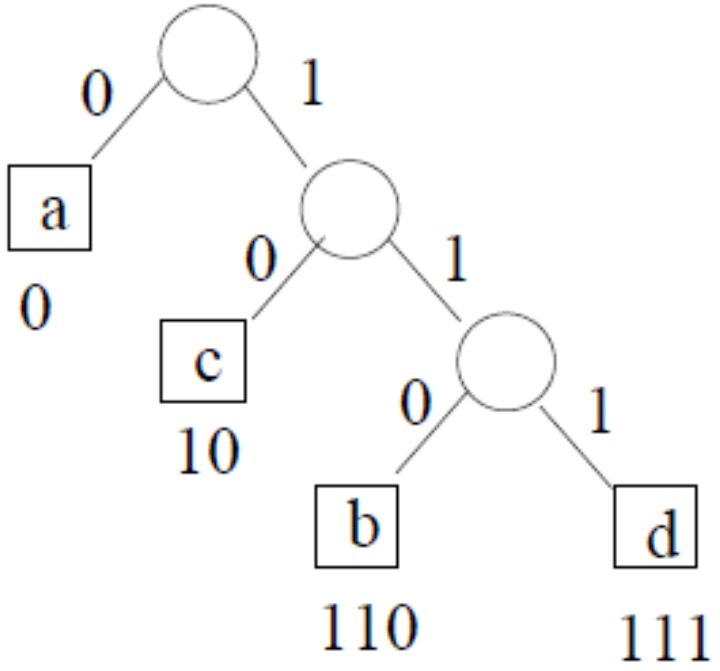


The Huffman code is:
$a = 000$, $b = 001$,
$c = 010$, $d = 011$,
$e = 1$.

Correspondence between Binary Trees and Prefix Codes

1-1 correspondence between leaves and characters.





- Left edge is labeled 0; right edge is labeled 1
- The binary string on a path from the root to a leaf is the codeword associated with the character at the leaf.

Huffman Coding

Set S be the original set of message characters.

1. (Greedy idea)
   - Pick two smallest frequency characters x, y from S.
   - Create a subtree that has these two characters as leaves.
   - Label the root of this subtree as z.
2.
   - Set frequency $f(z) = f(x) + f(y)$.
   - Remove x, y from S and add z to S
     - $S = S \cup \{z\} - \{x, y\}$.
     - Note that $|S|$ has just decreased by one.

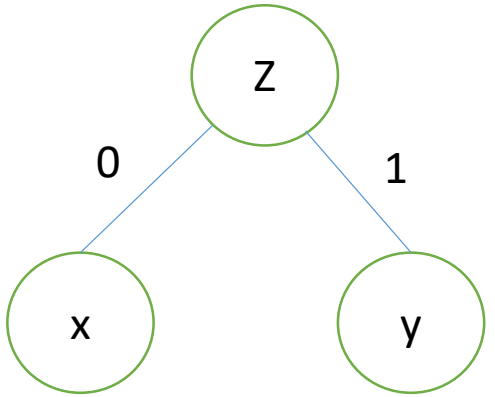
Repeat this procedure, called *merge*, with the new alphabet S, until S has only one character left in it.

The resulting tree is the Huffman code tree.

- It encodes the optimum (minimum-cost) prefix code for the given frequency distribution.

Huffman Coding Algorithm

Given character set S with frequency distribution $\{f(a) : a \in S\}$:
The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.



Huffman(S)

$n = |S|$;
$Q = S$; // the future leaves
for $i = 1$ **to** $n - 1$ **do**
    // Why $n - 1$?
    z = new node;
    x = $left[z]$ = Extract-Min(Q);
    y = $right[z]$ = Extract-Min(Q);
    $f[z] = f[left[z]] + f[right[z]]$;
    Insert(Q, z);
end
return Extract-Min(Q); // root of the tree

Running time is $O(n \log n)$, as each priority queue operation takes time $O(\log n)$.

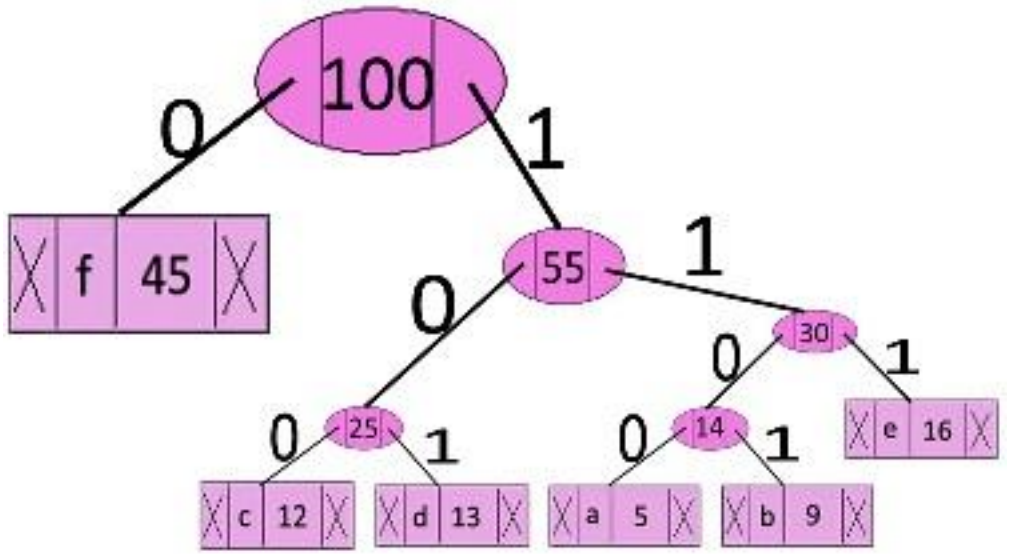
Huffman Code Practice Problem

Character	a	b	c	d	e	f
Frequency	5	9	12	13	16	45

Create Huffman code tree for above example. Please try it your self before looking at solution on next slide.

Huffman Code Practice Problem

- Solution



Notice that these codes are prefix codes

Character	a	b	c	d	e	f
Codes	1100	1101	100	101	111	0