Lecture 15: Huffman Coding

CLRS- 16.3

Outline of this Lecture

- Codes and Compression.
- Huffman coding.
- Correctness of the Huffman coding algorithm.

Suppose that we have a $100,000$ character data file that we wish to store . 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*. We would like to find a binary code that encodes the file using as few bits as possible, ie., *compresses it* as much as possible.

In a *fixed-length code* each codeword has the same length. In a *variable-length code* codewords may have different lengths. Here are examples of fixed and variable legth codes for our problem (note that a fixed-length code must have at least 3 bits per codeword).

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

The fixed length-code requires $300,000$ bits to store the file. The variable-length code uses 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! Can we do better?

Note: In what follows a *code* will be a set of codewords, e.g., $\{000, 001, 010, 011, 100, 101\}$
and $\{0, 101, 100, 111, 1101, 1100\}$

Encoding

Given a code (corresponding to some alphabet Γ) and a message it is easy to *encode* the message. Just replace the characters by the codewords.

Example: $\Gamma = \{a, b, c, d\}$
If the code is

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

then bad is encoded into 010011

If the code is

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

then bad is encoded into 1100111

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.
A message is *uniquely decodable* (vis-a-vis a particular code) if it can only be decoded in one way.

For example relative to C_1, 010011 is uniquely decodable to bad.
Relative to C_2 1100111 is uniquely decodable to bad.
But, relative to C_3, 1101111 is not uniquely decipherable since it could have encoded either bad or acad.

In fact, one can show that every message encoded using C_1 and C_2 are uniquely decipherable. The unique decipherability property is needed in order for a code to be useful.

Prefix-Codes

Fixed-length codes are always uniquely decipherable (why).
We saw before that these do not always give the best compression so we prefer to use variable length codes.

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

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

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

$$01101100 = 01101100 = abba$$

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

Fixed-Length versus Variable-Length Codes

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?

Fixed-Length versus Variable-Length Prefix Codes

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

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

a 25% saving.

Remark: We will see later that this is the *optimum* (lowest cost) prefix code.

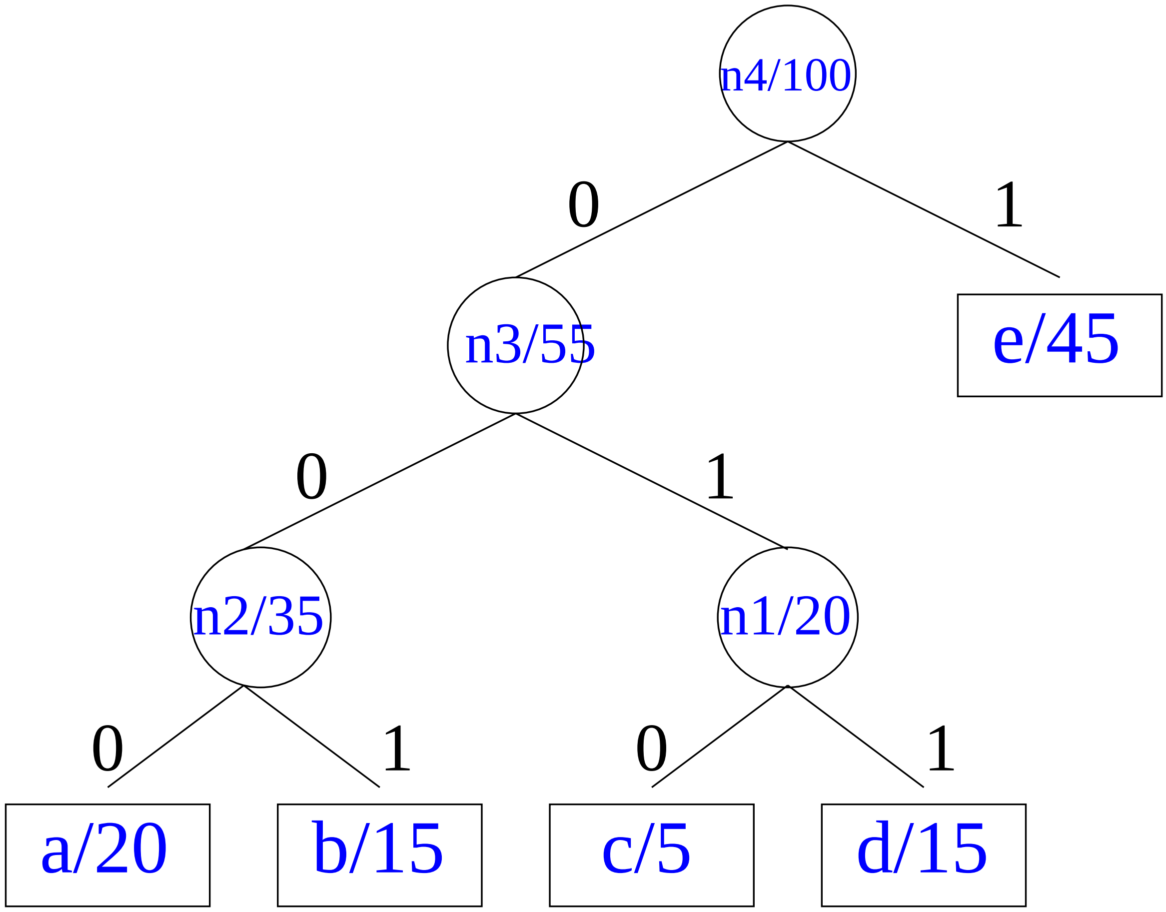
Optimum Source Coding Problem

The 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

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

needed to encode a message of $\sum_{a=1}^{n} f(a)$ characters, where $c(a_i)$ is the codeword for encoding a_i, and $L(c(a_i))$ is the length of the codeword $c(a_i)$.

Remark: Huffman developed a nice greedy algorithm for solving this problem and producing a minimum-cost (optimum) prefix code. The code that it produces is called a *Huffman code* .



$$\{a = 000,\ b = 001,\ c = 010,\ d = 011,\ e = 1\}$$

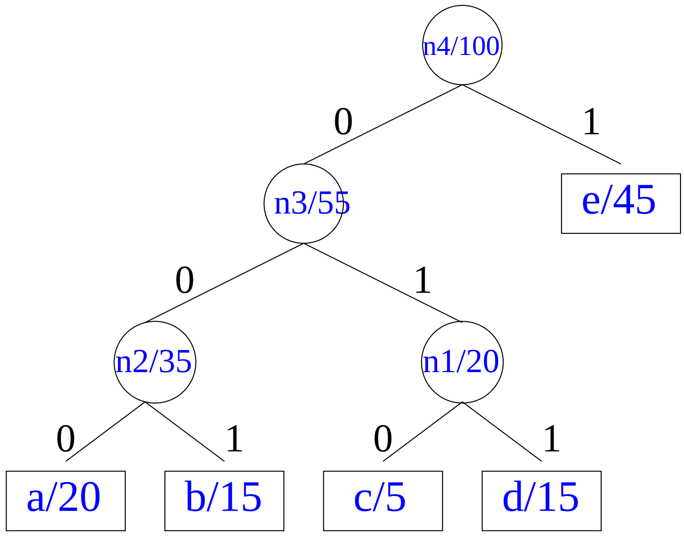
Correspondence between Binary Trees and prefix codes.

1-1 correspondence between leaves and characters.

Label of leaf is frequency of character.

Left edge is labeled 0; right edge is labeled 1

Path from root to leaf is codeword associated with character.



$$\{a = 000,\ b = 001,\ c = 010,\ d = 011,\ e = 1\}$$

Note that $d(a_i)$, the depth of leaf a_i in tree T is equal to the length, $L(c(a_i))$ of the codeword in code C associated with that leaf. So,

$$\sum_{a=1}^{n} f(a_i)L(c(a_i)) = \sum_{a=1}^{n} f(a_i)d(a_i).$$

The sum $\sum_{a=1}^{n} f(a_i)d(a_i)$ is the *weighted external pathlength* of tree T .

The Huffman encoding problem is equivalent to the minimum-weight external pathlength problem: given weights $f(a_1), \ldots, f(a_n)$, find tree T with n leaves labeled $a_1, \ldots, a_n$ that has minimum weighted external path length.

Huffman Coding

Step 1: Pick two letters x, y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea)
Label the root of this subtree as z.

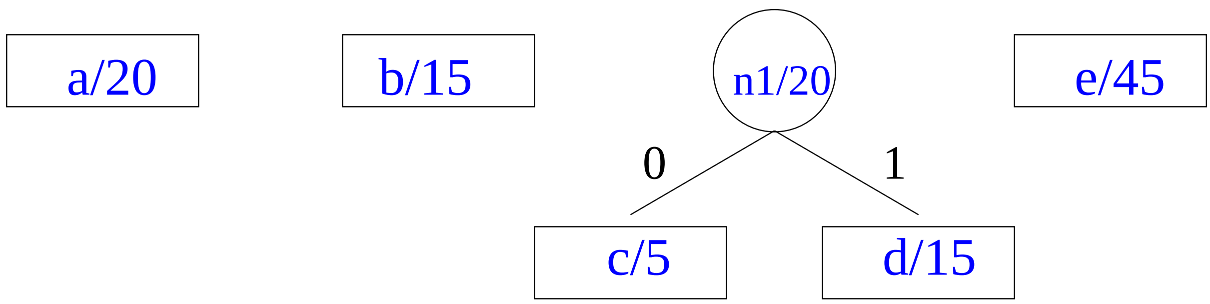
Step 2: Set frequency $f(z) = f(x) + f(y)$.
Remove x, y and add z creating new alphabet
$A' = A \cup \{z\} - \{x, y\}$.
Note that $|A'| = |A| - 1$.

Repeat this procedure, called *merge*, with new alphabet A' until an alphabet with only one symbol is left.

The resulting tree is the Huffman code.

Example of Huffman Coding

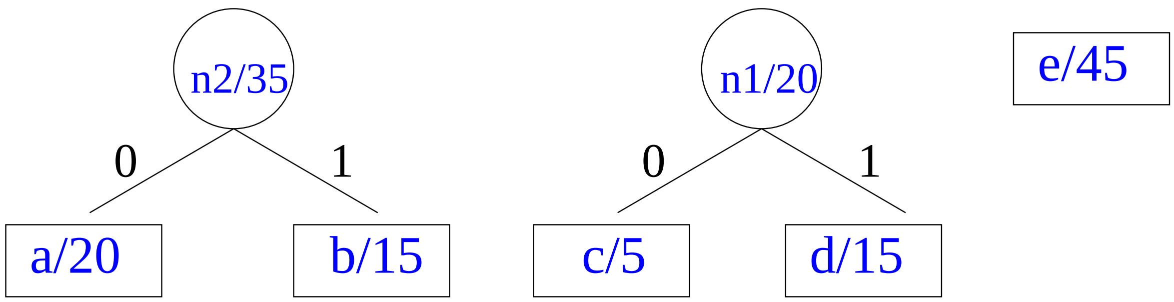
Let $A = \{a/20, b/15, c/5, d/15, e/45\}$ be the alphabet and its frequency distribution.
In the first step Huffman coding merges c and d.



Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.

Example of Huffman Coding – Continued

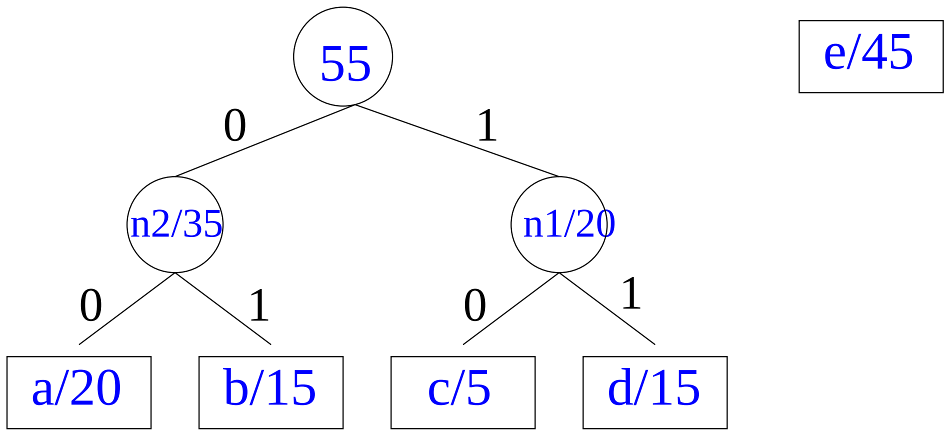
Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.
Algorithm merges a and b
(could also have merged $n1$ and b).



New alphabet is $A_2 = \{n2/35, n1/20, e/45\}$.

Example of Huffman Coding – Continued

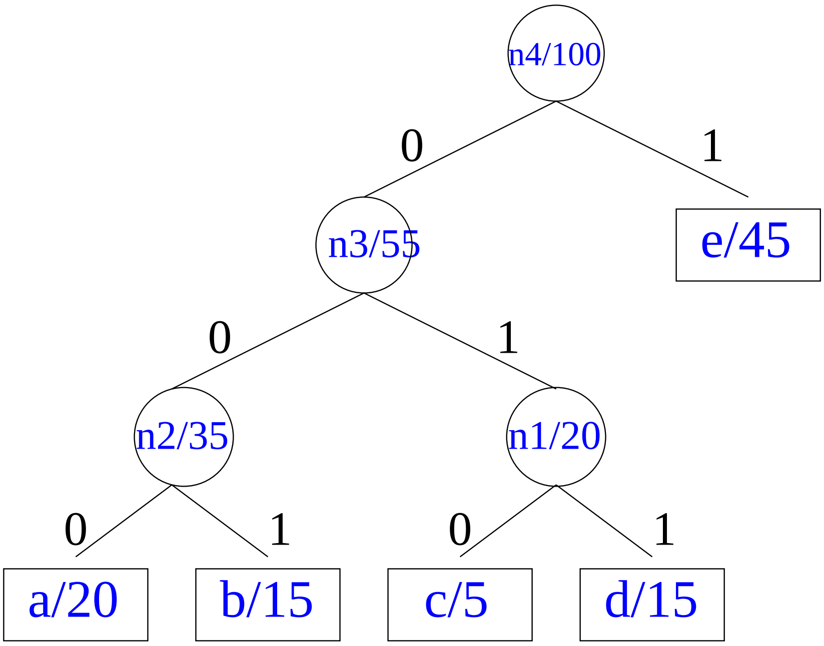
Alphabet is $A_2 = \{n2/35, n1/20, e/45\}$.
Algorithm merges $n1$ and $n2$.



New alphabet is $A_3 = \{n3/55, e/45\}$.

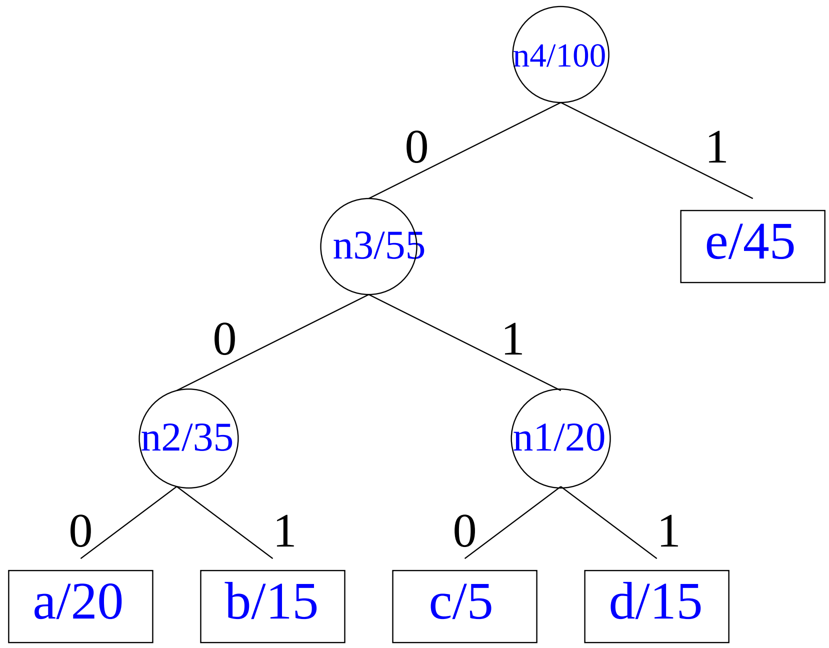
Example of Huffman Coding – Continued

Current alphabet is $A_3 = \{n3/55, e/45\}$.
Algorithm merges e and $n3$ and finishes.



Example of Huffman Coding – Continued

Huffman code is obtained from the Huffman tree.



Huffman code is
$a = 000$, $b = 001$, $c = 010$, $d = 011$, $e = 1$.
This is the optimum (minimum-cost) prefix code for this distribution.

Huffman Coding Algorithm

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

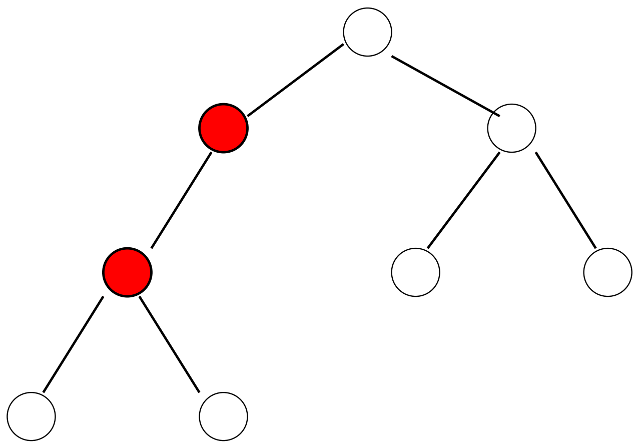
Huffman(A)
{ $n = |A|$;
  $Q = A$; the future leaves
  for $i = 1$ to $n - 1$ Why $n - 1$?
  { $z =$ new node;
    $left[z]$ =Extract-Min(Q);
    $right[z]$ =Extract-Min(Q);
    $f[z] = f[left[z]] + f[right[z]]$;
    Insert(Q, z);
  }
  return Extract-Min(Q) root of the tree
}

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

An Observation: Full Trees

Lemma: The tree for any optimal prefix code must be "full", meaning that every internal node has exactly two children.

Proof: If some internal node had only one child then we could simply get rid of this node and replace it with its unique child. This would decrease the total cost of the encoding.



Huffman Codes are Optimal

Lemma: Consider the two letters, x and y with the smallest frequencies. There is an optimal code tree in which these two letters are sibling leaves in the tree in the lowest level.

Proof: Let T be an optimum prefix code tree, and let b and c be two siblings at the maximum depth of the tree (must exist because T is full). Assume without loss of generality that $f(b) \le f(c)$ and $f(x) \le f(y)$ (if this is not true, then rename these characters). Since x and y have the two smallest frequencies it follows that $f(x) \le f(b)$ (they may be equal) and $f(y) \le f(c)$ (may be equal). Because b and c are at the deepest level of the tree we know that $d(b) \ge d(x)$ and $d(c) \ge d(y)$.

Now switch the positions of x and b in the tree resulting in a different tree T' and see how the cost changes. Since T is optimum,

$$
\begin{aligned}
B(T) &\le B(T') \\
&= B(T) - f(x)d(x) - f(b)d(b) + f(x)d(b) + f(b)d(x) \\
&= B(T) + (f(x) - f(b))(d(b) - d(x)) \\
&\le B(T).
\end{aligned}
$$

Therefore, $B(T') = B(T)$, that is, T' is an optimum tree. By switching y with c we get a new tree T'' which by a similar argument is optimum. The final tree T'' satisfies the statement of the claim.

Huffman Codes are Optimal

We have just shown there is *an* optimum tree $T^{''}$ agrees with our *first* greedy choice, i.e., x and y are siblings, and are in the lowest level.

Similarly to the proof we seen early for the fractional knapsack problem, we still need to show the *optimal substructure* property of Huffman coding problem.

Lemma: Let T be a full binary tree representing an optimal prefix code over an alphabet C, where frequency $f[c]$ is defined for each character $c \in C$. Consider any two characters x and y that appear as sibling leaves in T, and let z be their parent. Then, considering z as a character with frequency $f[z] = f[x] + f[y]$, the tree $T^{'} = T - \{x, y\}$ represents an optimal prefix code for the alphabet $C^{'} = C - \{x, y\} \bigcup \{z\}$.

Huffman Codes are Optimal

Proof: An exercise. (Hint : First write down the cost relation between $T^{'}$, and T. We then show $T^{'}$ is an optimal prefix code tree for $C^{'}$ by *contradiction* (by making use of the assumption that T is an optimal tree for C.))

By combining the results of the lemma, it follows that the Huffman codes are optimal.