Huffman Codes

- Huffman's greedy algorithm presents an optimal way to compress a sequence of characters as a binary string
- The algorithm uses a table that shows the frequency of each character in the given sequence

Data Compression Applications:

- Disk storage
- Networking: Lower network bandwidth when transmitting data

Example

- you have a 100,000-character data file that you wish to store compactly
- There are only 6 distinct characters used in the data sequence
- Table 1 below includes the frequency of each character in thousands

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- We consider representing the sequence as a binary character code
- Each character is represented by a unique binary string (codeword)
- If we use a fixed-length codes for n distinct characters, we need $\lceil lg\, n \rceil$ bits to represent each character

For instance:

- Two characters need 1-bit representation, that are {0,1}
- Three characters need a 2-bits representation, that are {00,01,10}
- Four characters need *also* a 2-bits representation, that are {00,01,10,11} and so on ..

- With fixed-length codes we represent our 100,000 characters sequence with 300,000 bits

- To obtain a shorter representation, we can use a variable-length code

- We give frequent characters short codewords and infrequent characters long codewords

- The last row in Table 1 shows the variable-length codes in our example

- The number of bits needed with the new representation is

$$(45 \cdot 1 \ + \ 13 \cdot 3 \ + \ 12 \cdot 3 \ + \ 16 \cdot 3 \ + \ 9 \cdot 4 \ + \ 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

- Obviously, an algorithm is needed to create the variable-length codes

- Also, such representation needs a method to extract the original sequence

- Example: Show how 110001001101 is a representation of the sequence **face** using the variable-length codes from Table 1
- In fact the codewords in Table 1 are optimal for data compression
- Such codewords are called *prefix-free codes* (no codeword is also a prefix of some other codeword)
- Another example, Show how 100011001101 is a representation of the sequence **cafe** using the variable-length codes from Table 1

Constructing a Huffman code

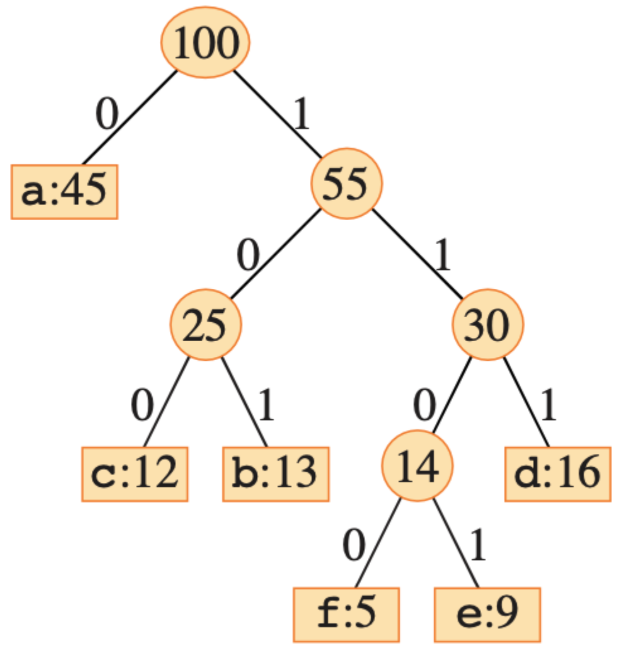
Huffman invented a greedy algorithm that constructs an optimal prefix-free code

Notations:

- C is the sequence alphabet
- Each C's alphabet character $c \in C$, and has an attribute $c.freq$ giving its frequency
- Q is a min-priority queue (Can be implemented using a MIN-HEAP)
- EXTRACT-MIN gives the item with the minimum value in the queue

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    x = EXTRACT-MIN(Q)
    y = EXTRACT-MIN(Q)
    z.left = x
    z.right = y
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // the root of the tree is the only node left
```

According to the character frequencies given in Table 1, the Huffman's algorithm returns the following tree



How to show that Huffman's algorithm is a greedy one?

1 - Solving the problem locally by selecting the two characters with the lowest frequencies is the greedy choice

2 - Optimal prefix-free codes have the optimal-substructure property (The optimal problem solution includes optimal solutions to its subproblems)

x and y are characters with the minimum frequencies in C

Let C' be C with characters x and y removed, and a new character z is added

C'

(The proofs of these properties are beyond the scope of this topic)

Time complexity analysis

- Build-Min-Heap initializes Q in $O(n)$
- Each heap operation takes $O(lg\, n)$
- Since we have a loop that iterates for $n - 1$, the running time is $O(n\, lg\, n)$

Summary

- A greedy algorithm picks a local optimal solution that leads to a globally optimal solution
- Greedy algorithms solve problems with optimal substructure