Huffman codes

- Data compression, typically saving 20%–90%
- Basic idea:
  represent often encountered characters by shorter (binary) codes

Huffman codes

Example

- Suppose we have the following data file with total 100K characters:

Char.	a	b	c	d	e	f
Freq.	45K	13K	12K	16K	9K	5K
3-bit fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

- Total number of bits required to encode the file:
  - Fixed-length code:
  $$100\text{K} \times 3 = 300\text{K}$$
  - Variable-length code:
  $$1{\cdot}45\text{K} + 3{\cdot}13\text{K} + 3{\cdot}12\text{K} + 3{\cdot}16\text{K} + 4{\cdot}9\text{K} + 4{\cdot}5\text{K} = 225\text{K}$$
- Variable-length code saves 25%.

Huffman codes

Prefix codes

- Prefix codes: no codeword is also a prefix of some other code.
- Encoding and decoding with a prefix code.
  Example:

Char.	a	b	c	d	e	f
Code	0	101	100	111	1101	1100

Encode:

- beef $\longrightarrow$ 101110111011100
- face $\longrightarrow$ 110001001101

Decode:

- 101110111011100 $\longrightarrow$ beef
- 110001001101 $\longrightarrow$ face

Huffman codes

Representation of prefix code:

- full binary tree (every nonleaf node has two children)
- All legal codes are at the leaves, since no prefix is shared

Huffman codes

Cost and optimality

- Given a code = a binary tree T, for each $c \in C$, define

$$\begin{aligned} f(c) &= \text{frequency of } c \text{ in the file} \\ d_T(c) &= \text{depth of } c\text{' leave in the tree } T \\ &= \text{length of the code for } c \\ &= \text{number of bits} \end{aligned}$$

  Then the number of bits ("cost of the tree/code T") required to encode the file

$$B(T) = \sum_{c \in C} f(c) d_T(c),$$

- A code T is optimal if $B(T)$ is minimal.

Huffman codes

Review: priority queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated key.
- A min-priority queue supports the following operations:
  - Insert(S,x): inserts the element x into the set S, i.e., $S = S \cup \{x\}$.
  - Minimum(S): returns the element of S with the smallest "key".
  - ExtractMin(S): removes and returns the element of S with the smallest "key".
  - DecreaseKey(S,x,k): decreases the value of element x's key to the new value k, which is assumed to be at least as small as x's current key value.
- A max-priority queue supports the operations: Insert(S, x), Maximum(S), ExtractMax(S), IncreaseKey(S, x, k).

Note: use a heap to implement a prority queue is described in section 6.5 of [CLRS 3rd ed.]

Huffman codes

Basic idea of Huffman codes to produce a prefix code for alphabet C
Let $C =$ alphabet (set of characters)

1. Builds a full binary tree T in a *bottom-up* manner
2. Begins with $|C|$ leaves, performs a sequence of $|C| - 1$ "merging" operations to create T
3. "Merging" operation is *greedy:* the two with lowest frequencies are merged.

Huffman codes

Pseudocode

```
Huffmancode(C)
n = |C|
Q = C    // min-priority queue, keyed by freq attribute
for i = 1 to n-1
    allocate a new node z
    z_left = x = ExtractMin(Q)
    z_right = y = ExtractMin(Q)
    freq[z] = freq[x] + freq[y]
    Insert(Q,z)
endfor
return ExtractMin(Q)  // the root of the tree
```

Huffman codes

Optimality: To prove the greedy algorithm Huffmancode producing an optimal prefix code, we show that it exhibits the following two ingradients:

1. **The greedy-choice property**

   If $x, y \in C$ and $f(x) = f(y)$, then there exists an optimal code T such that

   - $d_T(x) = d_T(y)$
   - the codes for x and y differ only in the last bit

2. **The optimal substructure property**

   If $x, y \in C$ having the lowest frequencies, and let z be their parent. Then the tree

   $$T' = T - \{x, y\}$$

   represents an optimal prefix code for the alphabet

   $$C' = (C - \{x, y\}) \cup \{z\}.$$

The proofs are on pages 433-435 of [CLRS 3rd ed.]

Huffman codes

By the above two properties, after each greedy choice is made, we are left with an optimization problem of the same form as the original. By induction, we have

Theorem. Huffman code is an optimal prefix code.