Huffman Code

Abhiram Ranade

February 15, 2016

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

Encoding: Converting the string from the source alphabet to its representation using the target alphabet

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

Encoding: Converting the string from the source alphabet to its representation using the target alphabet

Decoding: Given a representation in the target alphabet, recovering the string from which it was generated.

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

Encoding: Converting the string from the source alphabet to its representation using the target alphabet

Decoding: Given a representation in the target alphabet, recovering the string from which it was generated.

Goal: Result of encoding should be short.

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

Encoding: Converting the string from the source alphabet to its representation using the target alphabet

Decoding: Given a representation in the target alphabet, recovering the string from which it was generated.

Goal: Result of encoding should be short.

- Takes smaller memory to store.

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

Encoding: Converting the string from the source alphabet to its representation using the target alphabet

Decoding: Given a representation in the target alphabet, recovering the string from which it was generated.

Goal: Result of encoding should be short.

- Takes smaller memory to store.
- Takes less time to transmit.

The coding problem

We are often called upon to represent a string in one alphabet using another alphabet.

Example: We represent strings written in the English language (Roman alphabet) using binary numbers using the ASCII code.

Encoding: Converting the string from the source alphabet to its representation using the target alphabet

Decoding: Given a representation in the target alphabet, recovering the string from which it was generated.

Goal: Result of encoding should be short.

- Takes smaller memory to store.
- Takes less time to transmit.

There are other goals to, not considered here.

Fixed and variable length codes

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.
ASCII code: 8 bits.

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.
ASCII code: 8 bits.

Variable length codes: Codewords may have different lengths.

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.
ASCII code: 8 bits.

Variable length codes: Codewords may have different lengths.

- If a certain character occurs more frequently, assign it a shorter codeword. Thus your message is encoded more compactly.

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.
ASCII code: 8 bits.

Variable length codes: Codewords may have different lengths.

- If a certain character occurs more frequently, assign it a shorter codeword. Thus your message is encoded more compactly.
- Designing variable length codes is trickier.

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.
ASCII code: 8 bits.

Variable length codes: Codewords may have different lengths.

- If a certain character occurs more frequently, assign it a shorter codeword. Thus your message is encoded more compactly.
- Designing variable length codes is trickier.
  Example: Using digits to encode letters A-Z:
  Say we use $\gamma(A) = 1, \gamma(B) = 2, \ldots, \gamma(Z) = 26$.
  Will this work?

Fixed and variable length codes

Represent each character c in the source alphabet by a string ("codeword") $\gamma(c)$ of characters of the target alphabet.

Fixed length codes: All codewords have same length.
ASCII code: 8 bits.

Variable length codes: Codewords may have different lengths.

- If a certain character occurs more frequently, assign it a shorter codeword. Thus your message is encoded more compactly.
- Designing variable length codes is trickier.
  Example: Using digits to encode letters A-Z:
  Say we use $\gamma(A) = 1, \gamma(B) = 2, \ldots, \gamma(Z) = 26$.
  Will this work?

  26 could mean BF or Z. Not acceptable.
  The code needs to be uniquely decodable.

Encoding efficiency

Encoding efficiency

All characters do not appear with the same frequency.

Encoding efficiency

All characters do not appear with the same frequency.

For example, in the English language, the letter 'e' is much more common than the letter 'q'.

Encoding efficiency

All characters do not appear with the same frequency.

For example, in the English language, the letter 'e' is much more common than the letter 'q'.

Suppose f_c denotes the probability with which the character c appears in strings to be encoded.

Encoding efficiency

All characters do not appear with the same frequency.

For example, in the English language, the letter 'e' is much more common than the letter 'q'.

Suppose f_c denotes the probability with which the character c appears in strings to be encoded.

Estimated by sampling

Encoding efficiency

All characters do not appear with the same frequency.

For example, in the English language, the letter 'e' is much more common than the letter 'q'.

Suppose f_c denotes the probability with which the character c appears in strings to be encoded.

Estimated by sampling

Expected encoding length per character, $\text{EEL} = \sum_c |\gamma(c)| f_c$

Encoding efficiency

All characters do not appear with the same frequency.

For example, in the English language, the letter 'e' is much more common than the letter 'q'.

Suppose f_c denotes the probability with which the character c appears in strings to be encoded.

Estimated by sampling

Expected encoding length per character, $\mathrm{EEL} = \sum_c |\gamma(c)| f_c$

Desirable: Find a uniquely decodable code γ such that $\sum_c |\gamma(c)| f_c$ is minimized, for given f_c.

Prefix (free) codes

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Example: A=01, B=001, C=0001, D=00001, ..., Z=$0^{26}1$

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Example: A=01, B=001, C=0001, D=00001, ..., Z=$0^{26}1$

Non-Example: A=1, B=2, ..., Z=26.

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Example: A=01, B=001, C=0001, D=00001, . . ., Z=$0^{26}1$

Non-Example: A=1, B=2, . . ., Z=26.

Thm: Every message encoded by a prefix code is uniquely decodable.

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Example: A=01, B=001, C=0001, D=00001, ..., Z=$0^{26}1$

Non-Example: A=1, B=2, ..., Z=26.

Thm: Every message encoded by a prefix code is uniquely decodable.
Proof:

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Example: A=01, B=001, C=0001, D=00001, ..., Z=$0^{26}1$

Non-Example: A=1, B=2, ..., Z=26.

Thm: Every message encoded by a prefix code is uniquely decodable.
Proof: Decode greedily!

Prefix (free) codes

A code is said to be a prefix (free) code if no codeword is a prefix of another.

Example: A=01, B=001, C=0001, D=00001, ..., Z=$0^{26}1$

Non-Example: A=1, B=2, ..., Z=26.

Thm: Every message encoded by a prefix code is uniquely decodable.
Proof: Decode greedily!

Non-prefix uniquely decodable codes exist, but for any such code there is a prefix code with no larger EEL.

A representation for Prefix codes

A representation for Prefix codes

Digital Search tree/Trie:

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.
- Certain vertices are marked as special.

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.
- Certain vertices are marked as special.
- If $x =$ concatenation of labels along the path from the root to a special node, then we consider x to be stored in the trie.

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.
- Certain vertices are marked as special.
- If $x =$ concatenation of labels along the path from the root to a special node, then we consider x to be stored in the trie.

Representation of Prefix code γ: Trie T holding codewords $\gamma(c)$.

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.
- Certain vertices are marked as special.
- If x = concatenation of labels along the path from the root to a special node, then we consider x to be stored in the trie.

Representation of Prefix code γ: Trie T holding codewords $\gamma(c)$.
Code is prefix free $\Rightarrow$ Codewords end at leaves.

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.
- Certain vertices are marked as special.
- If x = concatenation of labels along the path from the root to a special node, then we consider x to be stored in the trie.

Representation of Prefix code γ: Trie T holding codewords $\gamma(c)$.
Code is prefix free $\Rightarrow$ Codewords end at leaves.

Vertex in which $\gamma(c)$ ends is labelled c.

A representation for Prefix codes

Digital Search tree/Trie:

- A rooted tree. Edges labelled with a character of the (target) alphabet.
- Certain vertices are marked as special.
- If x = concatenation of labels along the path from the root to a special node, then we consider x to be stored in the trie.

Representation of Prefix code γ: Trie T holding codewords $\gamma(c)$.
Code is prefix free $\Rightarrow$ Codewords end at leaves.

Vertex in which $\gamma(c)$ ends is labelled c.

Example:
$\gamma(a) = 11, \gamma(b) = 10, \gamma(c) = 01, \gamma(d) = 001, \gamma(e) = 000$

Decoding Algorithm

Decoding Algorithm

1. Let encoded string be $x_0 x_1 \ldots x_{n-1}$.

Decoding Algorithm

1. Let encoded string be $x_0 x_1 \ldots x_{n-1}$.
2. $i = 0$

Decoding Algorithm

1. Let encoded string be $x_0 x_1 \ldots x_{n-1}$.
2. $i = 0$
3. Start at the root of T. Use characters $x_i x_{i+1} \ldots$ to trace a path to some leaf with a label c. Output c as the decoded character.

Decoding Algorithm

1. Let encoded string be $x_0 x_1 \ldots x_{n-1}$.
2. $i = 0$
3. Start at the root of T. Use characters $x_i x_{i+1} \ldots$ to trace a path to some leaf with a label c. Output c as the decoded character.
4. Suppose the path used in the above step had length j. Then set $i = i + j$ and repeat the above step, unless end of the end of the encoded string is reached.

Encoding efficiency

Encoding efficiency

Expected encoding length per character, EEL $= \sum_c |\gamma(c)| f_c$

Encoding efficiency

Expected encoding length per character, EEL $= \sum_c |\gamma(c)| f_c$

Observation: $|\gamma(c)| = L(c)$, the length of the path from the root to the leaf labelled c in the trie.

Encoding efficiency

Expected encoding length per character, $\text{EEL} = \sum_c |\gamma(c)| f_c$

Observation: $|\gamma(c)| = L(c)$, the length of the path from the root to the leaf labelled c in the trie.

Design the trie to minimize $\sum_c L(c) f_c$

Huffman's algorithm

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL =$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32 + 2 \times 0.25$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32 + 2 \times 0.25 + 2 \times 0.20$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32 + 2 \times 0.25 + 2 \times 0.20 + 3 \times 0.18$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32 + 2 \times 0.25 + 2 \times 0.20 + 3 \times 0.18 + 3 \times 0.05$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32 + 2 \times 0.25 + 2 \times 0.20 + 3 \times 0.18 + 3 \times 0.05$
$= 1.55 + 0.69 = 2.21$

Huffman's algorithm

Designs an optimal prefix code, for the case of the binary target alphabet $\{0, 1\}$

Example result:
Source alphabet $S = \{a, b, c, d, e\}$,
probabilities: $\{0.32, 0.25, 0.20, 0.18, 0.05\}$

Huffman code: $a = 11, b = 10, c = 01, d = 001, e = 000$

$EEL = 2 \times 0.32 + 2 \times 0.25 + 2 \times 0.20 + 3 \times 0.18 + 3 \times 0.05$
$= 1.55 + 0.69 = 2.21$

Fixed length code would have 3 bits each; $EEL = 3$.

Properties of the optimal trie

Properties of the optimal trie

Lemma 1: Every node has either two children or none.

Properties of the optimal trie

Lemma 1: Every node has either two children or none.

Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\qquad L(c_1)f(c_1) + L(c_2)f(c_2)$

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\qquad L(c_1)f(c_1) + L(c_2)f(c_2)$
Suppose we exchange the labels c_1, c_2.

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\quad L(c_1)f(c_1) + L(c_2)f(c_2)$
Suppose we exchange the labels c_1, c_2.
New contribution of c_1, c_2 to EEL: $\quad L(c_1)f(c_2) + L(c_2)f(c_1)$

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\quad L(c_1)f(c_1) + L(c_2)f(c_2)$
Suppose we exchange the labels c_1, c_2.
New contribution of c_1, c_2 to EEL: $\quad L(c_1)f(c_2) + L(c_2)f(c_1)$

Initial $-$ final $= L(c_1)(f(c_1) - f(c_2)) + L(c_2)(f(c_2) - f(c_1))$

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\qquad L(c_1)f(c_1) + L(c_2)f(c_2)$
Suppose we exchange the labels c_1, c_2.
New contribution of c_1, c_2 to EEL: $\qquad L(c_1)f(c_2) + L(c_2)f(c_1)$

Initial $-$ final $= L(c_1)(f(c_1) - f(c_2)) + L(c_2)(f(c_2) - f(c_1))$
$= [L(c_1) - L(c_2)][f(c_1) - f(c_2)]$

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\quad L(c_1)f(c_1) + L(c_2)f(c_2)$
Suppose we exchange the labels c_1, c_2.
New contribution of c_1, c_2 to EEL: $\quad L(c_1)f(c_2) + L(c_2)f(c_1)$

Initial $-$ final $= L(c_1)(f(c_1) - f(c_2)) + L(c_2)(f(c_2) - f(c_1))$
$= [L(c_1) - L(c_2)][f(c_1) - f(c_2)] \leq 0$ $\quad$ Initial is optimal.

Properties of the optimal trie

Lemma 1: Every node has either two children or none.
Proof: If some node has just one child, `short-cut` that node.
EEL must decrease.

Aside: how much is the decrease?
if c is lone child, decrease $= f(c)$

Lemma 2: $L(c_1) > L(c_2) \Rightarrow f(c_1) \leq f(c_2)$.
Proof: Consider an optimal trie.
Initial contribution of c_1, c_2 to EEL $\quad L(c_1)f(c_1) + L(c_2)f(c_2)$
Suppose we exchange the labels c_1, c_2.
New contribution of c_1, c_2 to EEL: $\quad L(c_1)f(c_2) + L(c_2)f(c_1)$

Initial $-$ final $= L(c_1)(f(c_1) - f(c_2)) + L(c_2)(f(c_2) - f(c_1))$
$= [L(c_1) - L(c_2)][f(c_1) - f(c_2)] \leq 0$ Initial is optimal.

$\square$

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \le f_d \le f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \le f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \ne c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.
Suppose character f appears at the other sibling.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.
Suppose character f appears at the other sibling.
$L(f) > L(d) \Rightarrow f(f) \leq f(d)$ (Lemma 2)

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.
Suppose character f appears at the other sibling.
$L(f) > L(d) \Rightarrow f(f) \leq f(d)$ (Lemma 2)

$f(f) < f(d)$ would imply $f(c), f(d)$ do not have least frequencies.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.
Suppose character f appears at the other sibling.
$L(f) > L(d) \Rightarrow f(f) \leq f(d)$ (Lemma 2)

$f(f) < f(d)$ would imply $f(c), f(d)$ do not have least frequencies.
Thus $f(f) = f(d) \Rightarrow$ either $f = d$ or we can exchange f, d without changing EEL.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.
Suppose character f appears at the other sibling.
$L(f) > L(d) \Rightarrow f(f) \leq f(d)$ (Lemma 2)

$f(f) < f(d)$ would imply $f(c), f(d)$ do not have least frequencies.
Thus $f(f) = f(d) \Rightarrow$ either $f = d$ or we can exchange f, d without changing EEL. $\square$.

The key lemma

There exists an optimal trie in which any two characters c, d with least frequencies (i.e. s.t. $f_c \leq f_d \leq f_e$ for any e) are siblings.

Proof: Let e appear furthest from the root in an optimal trie.
$L(e) > L(c) \Rightarrow f(e) \leq f(c)$ (Lemma 2)
$\Rightarrow f(e) = f(c)$ $f(c)$ is smallest
$e \neq c \Rightarrow$ we can exchange e, c without changing EEL.
WLOG optimal trie has c farthest from root.

The parent of c must have 2 children (Lemma 1)
Since c is at a farthest leaf, the other sibling must also be a leaf.
Suppose character f appears at the other sibling.
$L(f) > L(d) \Rightarrow f(f) \leq f(d)$ (Lemma 2)

$f(f) < f(d)$ would imply $f(c), f(d)$ do not have least frequencies.
Thus $f(f) = f(d) \Rightarrow$ either $f = d$ or we can exchange f, d without changing EEL. $\square$.

Greedy choice!

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.
Proof:

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.
Proof:
$T^* =$ optimal trie for S.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.
Proof:
$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.
Proof:
$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Proof:

$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.
Proof:
$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

$EEL(T'')$

Optimal Substructure

c_1, c_2 = two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
T' = an optimal trie for S'.
T = trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Proof:

T^* = optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
T'' = tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

$$EEL(T'') = EEL(T^*) - f(c_1) - f(c_2)$$

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Proof:
$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

$EEL(T'') = EEL(T^*) - f(c_1) - f(c_2)$
$\geq EEL(T')$ Optimality of T'

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Proof:

$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

$EEL(T'') = EEL(T^*) - f(c_1) - f(c_2)$
$\geq EEL(T')$ Optimality of T'

But $EEL(T) = EEL(T') + f(c_1) + f(c_2)$

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Proof:
$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

$EEL(T'') = EEL(T^*) - f(c_1) - f(c_2)$
$\geq EEL(T')$ Optimality of T'

But $EEL(T) = EEL(T') + f(c_1) + f(c_2) \leq EEL(T^*)$

Optimal Substructure

$c_1, c_2 =$ two least frequent characters in an alphabet S.
$S' = S - \{c_1, c_2\} \cup \{c\}$, with $f(c) = f(c_1) + f(c_2)$.
$T' =$ an optimal trie for S'.
$T =$ trie obtained from T' by attaching leaves c_1, c_2 to leaf c, and removing the label c.
Then T is optimal for S.

Proof:

$T^* =$ optimal trie for S. Wlog T^* has c_1, c_2 occurring as siblings.
$T'' =$ tree obtained from T^* by removing leaves c_1, c_2, and assigning their parent the label c.
T'' is also a trie for S'.

$EEL(T'') = EEL(T^*) - f(c_1) - f(c_2)$
$\geq EEL(T')$ Optimality of T'

But $EEL(T) = EEL(T') + f(c_1) + f(c_2) \leq EEL(T^*)$

Thus T is optimal for S.

Some implementation details

Some implementation details

Assume that the letters in S are numbered $1, \ldots, n$.

Some implementation details

Assume that the letters in S are numbered $1, \ldots, n$.

As the algorithm progresses, we will create new letters. These will be numbered $n+1, \ldots, 2n-1$.

Some implementation details

Assume that the letters in S are numbered $1, \ldots, n$.

As the algorithm progresses, we will create new letters. These will be numbered $n+1, \ldots, 2n-1$.

The trie will be generated as an array T.

Some implementation details

Assume that the letters in S are numbered $1, \ldots, n$.

As the algorithm progresses, we will create new letters. These will be numbered $n+1, \ldots, 2n-1$.

The trie will be generated as an array T.

If v is root, $T(v) = v$.

Some implementation details

Assume that the letters in S are numbered $1, \ldots, n$.

As the algorithm progresses, we will create new letters. These will be numbered $n+1, \ldots, 2n-1$.

The trie will be generated as an array T.

If v is root, $T(v) = v$.

Otherwise $T(v) =$ parent of v in the trie.

The algorithm

Huffman(S){

  If $S = \{c\}$ then $T(c) = c$. Root

  Let $c_1, c_2 =$ two least frequent characters from S.

  $S' = S - \{c_1, c_2\} \cup \{c\}$

  Huffman(S')

  $T(c_1) = T(c_2) = c$ Leaves c_1, c_2 below c.

}

The algorithm

Huffman(S){
  If $S = \{c\}$ then $T(c) = c$. Root
  Let $c_1, c_2 =$ two least frequent characters from S.
  $S' = S - \{c_1, c_2\} \cup \{c\}$
  Huffman(S')
  $T(c_1) = T(c_2) = c$ Leaves c_1, c_2 below c.
}

Extracting least frequency characters:

The algorithm

Huffman(S){
  If $S = \{c\}$ then $T(c) = c$. Root
  Let $c_1, c_2 =$ two least frequent characters from S.
  $S' = S - \{c_1, c_2\} \cup \{c\}$
  Huffman(S')
  $T(c_1) = T(c_2) = c$ Leaves c_1, c_2 below c.
}

Extracting least frequency characters:
Store $(c, f(c))$ pairs in a min heap, with key $= f(c)$.

The algorithm

Huffman(S){
  If $S = \{c\}$ then $T(c) = c$. Root
  Let $c_1, c_2 =$ two least frequent characters from S.
  $S' = S - \{c_1, c_2\} \cup \{c\}$
  Huffman(S')
  $T(c_1) = T(c_2) = c$ Leaves c_1, c_2 below c.
}

Extracting least frequency characters:
Store $(c, f(c))$ pairs in a min heap, with key $= f(c)$.
$2n - 1$ insert operations. $2n - 1$ delete-min operations.

The algorithm

Huffman(S){
  If $S = \{c\}$ then $T(c) = c$. Root
  Let $c_1, c_2 =$ two least frequent characters from S.
  $S' = S - \{c_1, c_2\} \cup \{c\}$
  Huffman(S')
  $T(c_1) = T(c_2) = c$ Leaves c_1, c_2 below c.
}

Extracting least frequency characters:
Store $(c, f(c))$ pairs in a min heap, with key $= f(c)$.
$2n - 1$ insert operations. $2n - 1$ delete-min operations.

Running time: $O(n \log n)$. (dominated by priority queue)

Example

Source alphabet $S = \{a, b, c, d, e\}$,
frequencies respectively $\{0, 32, 0.25, 0.20, 0.18, 0.05\}$

Remarks

Remarks

- How would you think up Huffman's algorithm? What is routine, what is insightful?

Remarks

- How would you think up Huffman's algorithm? What is routine, what is insightful?
- Insight 1: Representing codes using tries, and formulating the problem as finding a trie.

Remarks

- How would you think up Huffman's algorithm? What is routine, what is insightful?
- Insight 1: Representing codes using tries, and formulating the problem as finding a trie.
- Routine: Figuring out the properties of the optimal trie.

Remarks

- How would you think up Huffman's algorithm? What is routine, what is insightful?
- Insight 1: Representing codes using tries, and formulating the problem as finding a trie.
- Routine: Figuring out the properties of the optimal trie.
- Insight 2: Combining characters together.