CMPSC 465
Data Structures and Algorithms
Spring 2022

Instructor: Chunhao Wang

Greedy algorithms

Greedy algorithms

Huffman Encoding (Textbook Section 5.2)

Huffman Encoding

An encoding scheme used in, e.g., MP3 encoding

Huffman Encoding

An encoding scheme used in, e.g., MP3 encoding

Data: a string S of symbols over an alphabet Γ

Huffman Encoding

An encoding scheme used in, e.g., MP3 encoding

Data: a string S of symbols over an alphabet Γ
Goal: find a binary encoding e of Γ resulting in minimum encoded length of S

Huffman Encoding

An encoding scheme used in, e.g., MP3 encoding

Data: a string S of symbols over an alphabet Γ
Goal: find a binary encoding e of Γ resulting in minimum encoded length of S

Denote the encoded string by S_e

Huffman Encoding

An encoding scheme used in, e.g., MP3 encoding

Data: a string S of symbols over an alphabet Γ
Goal: find a binary encoding e of Γ resulting in minimum encoded length of S

Denote the encoded string by S_e

Example: ASCII encoding $\begin{array}{cc} a & 01100001 \\ b & 01100010 \\ \vdots & \vdots \end{array}$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1 : \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array}$$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1: \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1 : \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1: \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

$$e_2: \begin{array}{l} a \to 0 \\ b \to 10 \\ c \to 11 \end{array}$$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1 : \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

$$e_2 : \begin{array}{l} a \to 0 \\ b \to 10 \\ c \to 11 \end{array} \qquad |S_{e_2}| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1 : \begin{array}{l} a \rightarrow 00 \\ b \rightarrow 01 \\ c \rightarrow 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

$$e_2 : \begin{array}{l} a \rightarrow 0 \\ b \rightarrow 10 \\ c \rightarrow 11 \end{array} \qquad |S_{e_2}| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

- Be careful!

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1 : \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

$$e_2 : \begin{array}{l} a \to 0 \\ b \to 10 \\ c \to 11 \end{array} \qquad |S_{e_2}| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

- Be careful! $e_2 : \begin{array}{l} a \to 0 \\ b \to 1 \\ c \to 01 \end{array}$

Different encodings

Consider $\Gamma = \{a, b, c\}$

Stats on S: a appears 45 times, b 16 times, and c twice

- Fixed-length encoding

$$e_1: \begin{array}{l} a \to 00 \\ b \to 01 \\ c \to 10 \end{array} \qquad |S_{e_1}| = 45 \times 2 + 16 \times 2 + 2 \times 2 = 126$$

- Variable-length encoding

$$e_2: \begin{array}{l} a \to 0 \\ b \to 10 \\ c \to 11 \end{array} \qquad |S_{e_2}| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

- Be careful! $e_2: \begin{array}{l} a \to 0 \\ b \to 1 \\ c \to 01 \end{array}$ Decoding will lead to ambiguity

Prefix-free encoding

Consider the bad encoding $e_2:\quad \begin{array}{l} a \to 0 \\ b \to 1 \\ c \to 01 \end{array}$

Prefix-free encoding

Consider the bad encoding $e_2: \begin{array}{l} a \to 0 \\ b \to 1 \\ c \to 01 \end{array}$ How to decode 010110?

Prefix-free encoding

Consider the bad encoding e_2 : $\begin{array}{l} a \to 0 \\ b \to 1 \\ c \to 01 \end{array}$ How to decode 010110?

ababba?, *ccba*?, *abcba*?, or . . . ?

Prefix-free encoding

Consider the bad encoding $e_2: \begin{array}{l} a \rightarrow 0 \\ b \rightarrow 1 \\ c \rightarrow 01 \end{array}$ How to decode 010110?

ababba?, *ccba*?, *abcba*?, or ...?

To avoid ambiguity, we need the encoding to be prefix-free

Prefix-free encoding

Consider the bad encoding e_2 :

$$\begin{aligned} a &\to 0 \\ b &\to 1 \\ c &\to 01 \end{aligned}$$

How to decode 010110?

ababba?, *ccba*?, *abcba*?, or . . . ?

To avoid ambiguity, we need the encoding to be prefix-free

Definition

An encoding is **prefix-free** if no codeword is a prefix of any other codewords

Tree representation of a prefix-free encoding (I)

Definition

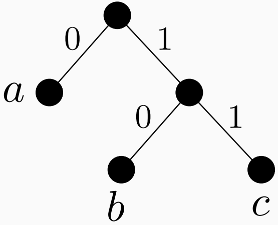
A **full binary tree** is a binary tree where each node is either a leaf or it has two children

Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding



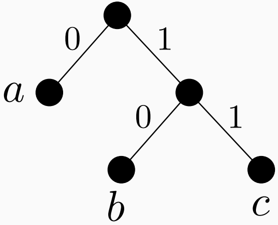
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ



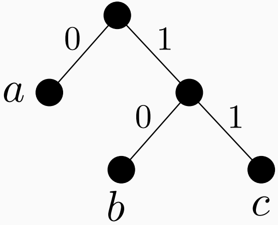
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0



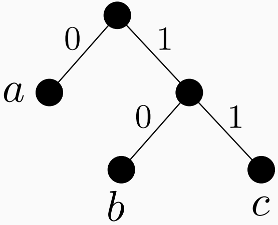
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0
- label edge to the right child with 1



Tree representation of a prefix-free encoding (I)

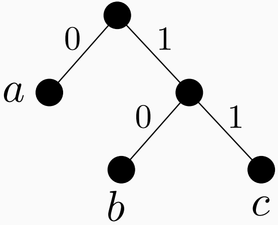
Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0
- label edge to the right child with 1

To obtain the encoding, read edge labels from root to a symbol



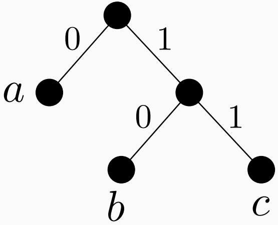
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0
- label edge to the right child with 1



To obtain the encoding, read edge labels from root to a symbol

$a \to 0$, $b \to 10$, $c \to 11$,

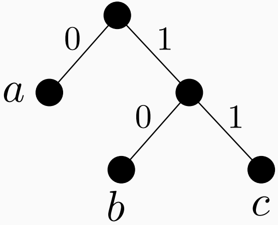
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0
- label edge to the right child with 1



To obtain the encoding, read edge labels from root to a symbol

$a \to 0$, $b \to 10$, $c \to 11$,

Depth of a leaf $\equiv$ length of its codeword

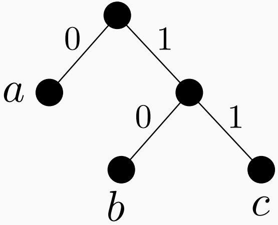
Tree representation of a prefix-free encoding (I)

Definition

A **full binary tree** is a binary tree where each node is either a leaf or it has two children

We use a full binary tree to represent a prefix-free encoding

- leaves are corresponding to symbols in Γ
- label edge to the left child with 0
- label edge to the right child with 1



To obtain the encoding, read edge labels from root to a symbol

$a \to 0$, $b \to 10$, $c \to 11$,

Depth of a leaf $\equiv$ length of its codeword

It guarantees to be prefix-free

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

Tree representation of a prefix-free encoding (II)

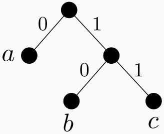
Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

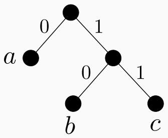


Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$

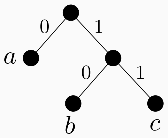


Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$



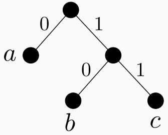
$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$



$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

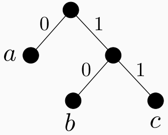
A useful re-write: label internal nodes with counts of descendants

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$



$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

A useful re-write: label internal nodes with counts of descendants

For all non-root node v, define

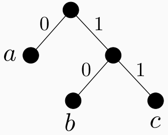
$\mathrm{cost}(v) :=$ sum of leaf node counts descending from v

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$

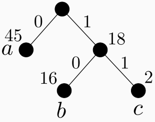


$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

A useful re-write: label internal nodes with counts of descendants

For all non-root node v, define

$\mathrm{cost}(v) :=$ sum of leaf node counts descending from v

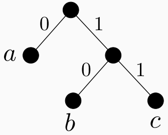


Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$

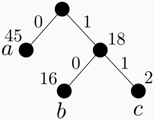


$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

A useful re-write: label internal nodes with counts of descendants

For all non-root node v, define

$\mathrm{cost}(v) :=$ sum of leaf node counts descending from v



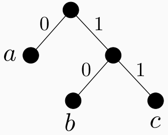
$$|S_e| = \sum_{v \in T - \{\mathrm{root}\}} \mathrm{cost}(v)$$

Tree representation of a prefix-free encoding (II)

Let e be an encoding represented by a tree

For string S, let f_v be the symbol count in S for each $v \in \Gamma$

$$|S_e| = \sum_{v \in \Gamma} f_v \cdot \mathrm{depth}(v)$$



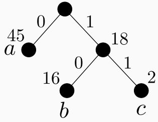
$$|S_e| = 45 \times 1 + 16 \times 2 + 2 \times 2 = 81$$

A useful re-write: label internal nodes with counts of descendants

For all non-root node v, define

$\mathrm{cost}(v) :=$ sum of leaf node counts descending from v

$$|S_e| = \sum_{v \in T - \{\mathrm{root}\}} \mathrm{cost}(v)$$



$$|S_e| = 45 + 16 + 2 + 18 = 81$$

Constructing the prefix-free encoding tree

Idea: put more frequent symbols at smaller depth

Constructing the prefix-free encoding tree

Idea: put more frequent symbols at smaller depth

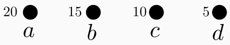
Greedy approach: continually merge least frequent symbols/nodes until you have a full binary tree encoding all symbols

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$

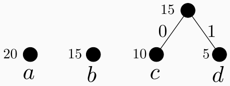
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



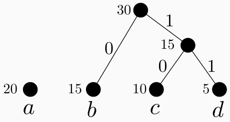
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



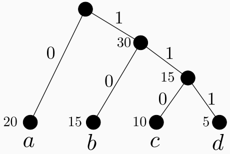
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



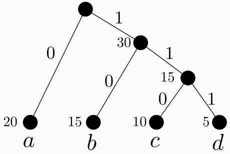
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



$a \to 0$

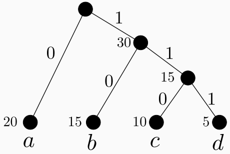
$b \to 10$

$c \to 110$

$d \to 111$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



$a \to 0$

$b \to 10$

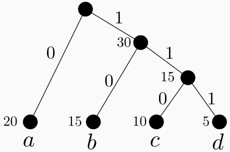
$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



$a \to 0$

$b \to 10$

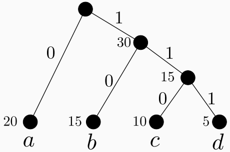
$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



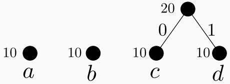
$a \to 0$

$b \to 10$

$c \to 110$

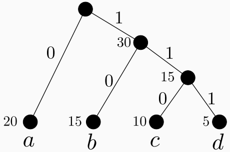
$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



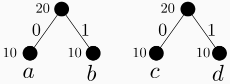
$a \to 0$

$b \to 10$

$c \to 110$

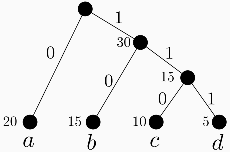
$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



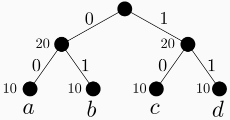
$a \to 0$

$b \to 10$

$c \to 110$

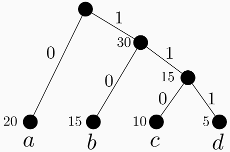
$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



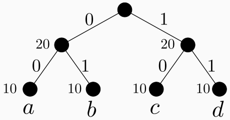
$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



$a \to 00$

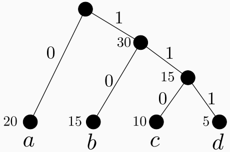
$b \to 01$

$c \to 10$

$d \to 11$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



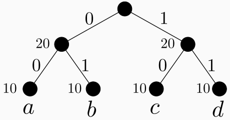
$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



$a \to 00$

$b \to 01$

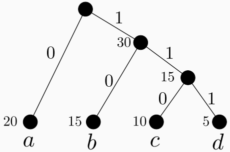
$c \to 10$

$d \to 11$

- $a : 10, b : 10, c : 5, d : 5$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



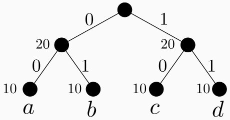
$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



$a \to 00$

$b \to 01$

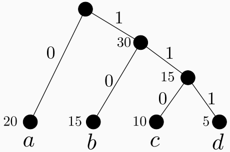
$c \to 10$

$d \to 11$

- $a : 10, b : 10, c : 5, d : 5$

Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



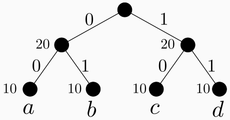
$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



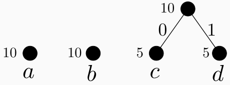
$a \to 00$

$b \to 01$

$c \to 10$

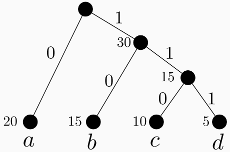
$d \to 11$

- $a : 10, b : 10, c : 5, d : 5$



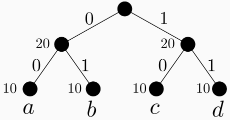
Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



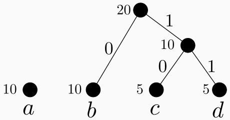
$a \to 0$
$b \to 10$
$c \to 110$
$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$

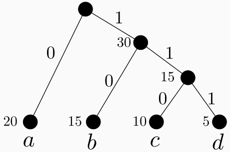


$a \to 00$
$b \to 01$
$c \to 10$
$d \to 11$

- $a : 10, b : 10, c : 5, d : 5$



- $a : 20, b : 15, c : 10, d : 5$



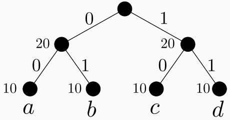
$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



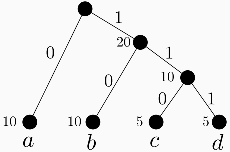
$a \to 00$

$b \to 01$

$c \to 10$

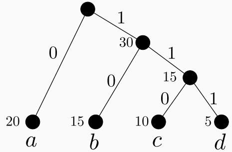
$d \to 11$

- $a : 10, b : 10, c : 5, d : 5$



Constructing the prefix-free encoding tree – examples

- $a : 20, b : 15, c : 10, d : 5$



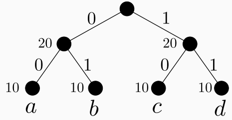
$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

- $a : 10, b : 10, c : 10, d : 10$



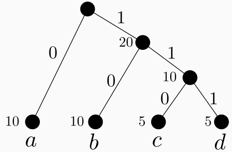
$a \to 00$

$b \to 01$

$c \to 10$

$d \to 11$

- $a : 10, b : 10, c : 5, d : 5$



$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

Proof of optimality (I)

Proof sketch

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Suppose we have a tree T with two lowest frequent symbols not as deep as possible.

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Suppose we have a tree T with two lowest frequent symbols not as deep as possible. Then at least one has a smaller depth.

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Suppose we have a tree T with two lowest frequent symbols not as deep as possible. Then at least one has a smaller depth. Switch it with one of the deepest nodes that is more frequent.

Proof of optimality (I)

Proof sketch

- Greedy choice property:

Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Suppose we have a tree T with two lowest frequent symbols not as deep as possible. Then at least one has a smaller depth. Switch it with one of the deepest nodes that is more frequent.

This improves the encoding length. Thus T is not optimal □

Proof of optimality (I)

Proof sketch

- Greedy choice property:

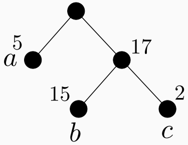
Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Suppose we have a tree T with two lowest frequent symbols not as deep as possible. Then at least one has a smaller depth. Switch it with one of the deepest nodes that is more frequent.

This improves the encoding length. Thus T is not optimal □



Proof of optimality (I)

Proof sketch

- Greedy choice property:

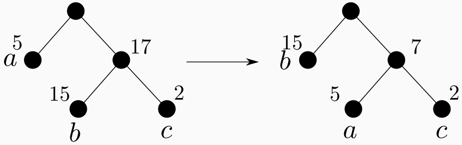
Claim

Every optimal solution has two lowest frequent symbols as leaves connected to an internal node of greatest depth

Proof. (exchange argument).

Suppose we have a tree T with two lowest frequent symbols not as deep as possible. Then at least one has a smaller depth. Switch it with one of the deepest nodes that is more frequent.

This improves the encoding length. Thus T is not optimal $\square$



Proof of optimality (II)

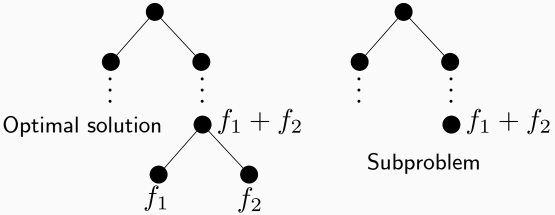
- Optimal substructure
  Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.

Proof of optimality (II)

- Optimal substructure
  Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$

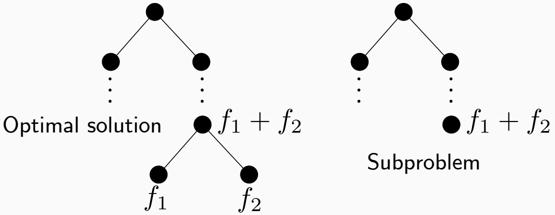
Proof of optimality (II)

- Optimal substructure
  Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$



Proof of optimality (II)

- Optimal substructure
  Consider a global optimal solution for $(f_1, f_2, \ldots, f_n)$ with f_1, f_2 the least frequencies.
  Our subproblem is: find an optimal solution for $(f_1 + f_2, f_3, \ldots, f_n)$



Thus, the greedy solution will lead to the global optimal solution

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
```

Pseudocode

def HUFFMAN(f): // $f: f[1], \ldots, f[n]$; Γ has n symbols

  T: empty tree;

  H: priority queue ordered by f;

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
        i := extract_min(H);
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
        i := extract_min(H);
        j := extract_min(H);
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
        i := extract_min(H);
        j := extract_min(H);
        Creat a node k in T with children i and j;
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
        i := extract_min(H);
        j := extract_min(H);
        Creat a node k in T with children i and j;
        f[k] := f[i] + f[j];
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
        i := extract_min(H);
        j := extract_min(H);
        Creat a node k in T with children i and j;
        f[k] := f[i] + f[j];
        insert(H, k);
```

Pseudocode

```
def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
    T: empty tree;
    H: priority queue ordered by f;
    for i := 1 in n:
        insert(H, i);
    for k := n + 1 in 2n − 1:
        i := extract_min(H);
        j := extract_min(H);
        Creat a node k in T with children i and j;
        f[k] := f[i] + f[j];
        insert(H, k);
```

Binary heap: insert $O(\log n)$, extract_min: $O(\log n)$

Pseudocode

```
1  def HUFFMAN(f): // f : f[1], ..., f[n]; Γ has n symbols
2      T: empty tree;
3      H: priority queue ordered by f;
4      for i := 1 in n:
5          insert(H, i);
6      for k := n + 1 in 2n − 1:
7          i := extract_min(H);
8          j := extract_min(H);
9          Creat a node k in T with children i and j;
10         f[k] := f[i] + f[j];
11         insert(H, k);
```

Binary heap: insert $O(\log n)$, extract_min: $O(\log n)$

Lines 4-5: $O(n \log n)$;

Pseudocode

1 **def** HUFFMAN(f): // $f : f[1], \ldots, f[n]$; Γ has n symbols
2   T: empty tree;
3   H: priority queue ordered by f;
4   **for** $i := 1$ **in** n**:**
5         $\mathrm{insert}(H, i)$;
6   **for** $k := n + 1$ **in** $2n - 1$**:**
7         $i := \mathrm{extract\_min}(H)$;
8         $j := \mathrm{extract\_min}(H)$;
9   Creat a node k in T with children i and j;
10         $f[k] := f[i] + f[j]$;
11         $\mathrm{insert}(H, k)$;

Binary heap: insert $O(\log n)$, extract_min: $O(\log n)$

Lines 4-5: $O(n \log n)$; Lines 6-11: $O(n \log n)$

Pseudocode

1 **def** HUFFMAN(f): // $f : f[1], \ldots, f[n]$; Γ has n symbols
2   T: empty tree;
3   H: priority queue ordered by f;
4   **for** $i := 1$ **in** n:
5         $\text{insert}(H, i)$;
6   **for** $k := n + 1$ **in** $2n - 1$:
7         $i := \text{extract\_min}(H)$;
8         $j := \text{extract\_min}(H)$;
9   Creat a node k in T with children i and j;
10         $f[k] := f[i] + f[j]$;
11         $\text{insert}(H, k)$;

Binary heap: insert $O(\log n)$, extract_min: $O(\log n)$

Lines 4-5: $O(n \log n)$; Lines 6-11: $O(n \log n)$

Total cost: $O(n \log n)$

More about the pseudocode

Question: why $2n-1$ in line 6?

More about the pseudocode

Question: why $2n - 1$ in line 6?

Answer: if a full binary tree has n leaves, then it has $2n - 1$ total nodes

More examples

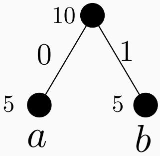
$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$

More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$

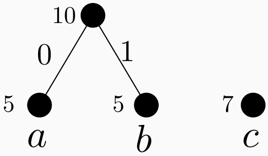
More examples

$a:5, b:5, c:7, d:10, e:15, f:17$



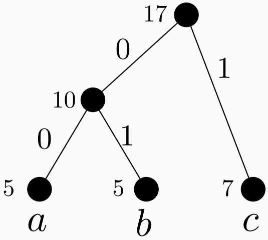
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



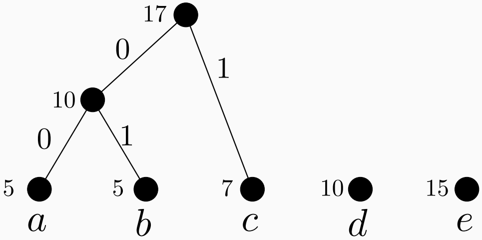
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



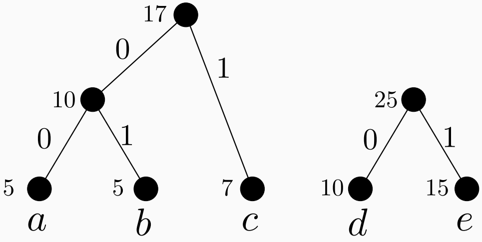
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



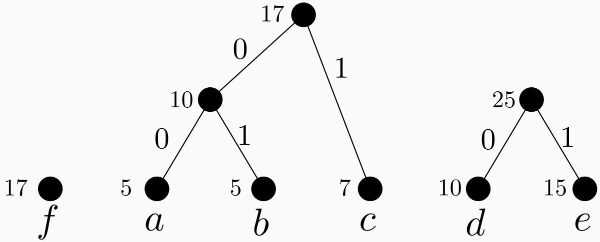
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



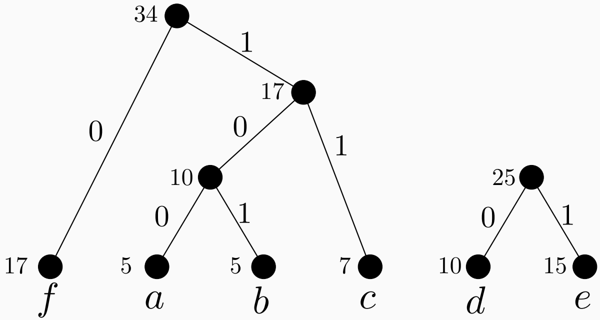
More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$



More examples

$a : 5, b : 5, c : 7, d : 10, e : 15, f : 17$