



# ***Design and analysis of algorithms***

## ***Lecture 22& 23***

Edyta Szymańska

edyta@cc.gatech.edu

# *Huffman codes - another greedy algorithm*

Store a map of a chromosome, i.e. a string of 130 million characters : A, C, G, T.

# *Huffman codes - another greedy algorithm*

Store a map of a chromosome, i.e. a string of 130 million characters : A, C, G, T.

How to do this?

# Huffman codes - another greedy algorithm

Store a map of a chromosome, i.e. a string of 130 million characters : A, C, G, T.

How to do this?

- ⑥ Default: 1 byte per character ? NO !

# Huffman codes - another greedy algorithm

Store a map of a chromosome, i.e. a string of 130 million characters : **A, C, G, T**.

How to do this?

- ⑥ Default: 1 byte per character ? NO !
- ⑥ 2 bits per character suffice: **A : 00, C : 01, G : 10, T : 11**, total= 260 Megabits used

# Huffman codes - another greedy algorithm

Store a map of a chromosome, i.e. a string of 130 million characters : **A, C, G, T**.

How to do this?

- ⑥ Default: 1 byte per character ? NO !
- ⑥ 2 bits per character suffice: **A : 00, C : 01, G : 10, T : 11**, total= 260 Megabits used

Extra information: the characters appear in the string with different frequencies, namely

$$f[A] = 70 \cdot 10^6, f[C] = 3 \cdot 10^6, f[G] = 20 \cdot 10^6, f[T] = 37 \cdot 10^6$$

Thus, it should be worth assigning a shorter bit string to *A* than to *C*.

# Huffman codes - another greedy algorithm

Store a map of a chromosome, i.e. a string of 130 million characters : **A, C, G, T**.

How to do this?

- ⑥ Default: 1 byte per character ? NO !
- ⑥ 2 bits per character suffice: **A : 00, C : 01, G : 10, T : 11**, total= 260 Megabits used

Extra information: the characters appear in the string with different frequencies, namely

$$f[A] = 70 \cdot 10^6, f[C] = 3 \cdot 10^6, f[G] = 20 \cdot 10^6, f[T] = 37 \cdot 10^6$$

Thus, it should be worth assigning a shorter bit string to *A* than to *C*.

# *Prefix codes*

prefix codes: no codeword can be a *prefix* of a different codeword (to avoid ambiguity in decoding).

# *Prefix codes*

prefix codes: no codeword can be a *prefix* of a different codeword (to avoid ambiguity in decoding).

The following is a proper prefix code:

$A : 0, C : 110, G : 111, T : 10$

# Prefix codes

prefix codes: no codeword can be a *prefix* of a different codeword (to avoid ambiguity in decoding).

The following is a proper prefix code:

$A : 0, C : 110, G : 111, T : 10$

Total number of bits used:  $213 \cdot 10^6$ , improvement = 17%

# Prefix codes

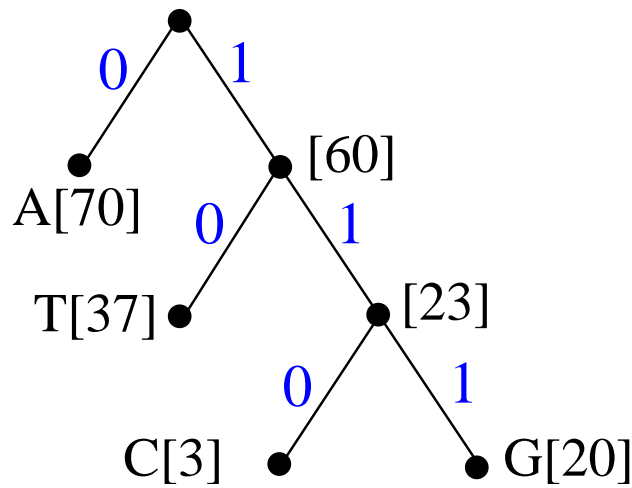
prefix codes: no codeword can be a *prefix* of a different codeword (to avoid ambiguity in decoding).

The following is a proper prefix code:

$A : 0, C : 110, G : 111, T : 10$

Total number of bits used:  $213 \cdot 10^6$ , improvement = 17%

Representation: binary tree



# *Huffman encoding algorithm*

The tree representation provides also a decoding scheme.

# Huffman encoding algorithm

The tree representation provides also a decoding scheme.

INPUT: a set of  $n$  characters from alphabet  $C$

with frequencies  $f_1, f_2, \dots, f_n$

OUTPUT: build a tree  $T$ , in which every leaf corresponds to a character and its depth (distance from the root  $d_T(c)$ ) is the length of its codeword, such that it minimizes the cost:

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

# Huffman encoding algorithm

The tree representation provides also a decoding scheme.

INPUT: a set of  $n$  characters from alphabet  $C$

with frequencies  $f_1, f_2, \dots, f_n$

OUTPUT: build a tree  $T$ , in which every leaf corresponds to a character and its depth (distance from the root  $d_T(c)$ ) is the length of its codeword, such that it minimizes the cost:

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

Also, the cost  $B(T)$  can be interpreted as the sum of all leaves and internal vertices, except the root, where the frequency assigned to an internal vertex is the sum of the frequencies of its descendant leaves.

# *Huffman encoding algorithm*

Properties of the optimal solution:

# *Huffman encoding algorithm*

Properties of the optimal solution:

- ⑥ The optimal solution is represented by a full binary tree.

# *Huffman encoding algorithm*

Properties of the optimal solution:

- ⑥ The optimal solution is represented by a full binary tree.
- ⑥ The two characters with smallest frequencies must be together at the bottom of the tree, as children of the lowest internal node of the tree.

# *Huffman encoding algorithm*

Properties of the optimal solution:

- ⑥ The optimal solution is represented by a full binary tree.
- ⑥ The two characters with smallest frequencies must be together at the bottom of the tree, as children of the lowest internal node of the tree.

# *Huffman encoding algorithm*



# *Huffman encoding algorithm*

Greedy algorithm:

# Huffman encoding algorithm

Greedy algorithm:

HUFFMAN( $C$ )

$Q := C$

for  $i := 1$  to  $n - 1$  do

    allocate a new node  $z$

$left[z] := x := \text{EXTRACT\_MIN}(Q)$

$right[z] := y := \text{EXTRACT\_MIN}(Q)$

$f[z] := f[x] + f[y]$

    INSERT( $Q, z$ )

return EXTRACT\_MIN(Q)

# Huffman encoding algorithm

Greedy algorithm:

HUFFMAN( $C$ )

$Q := C$

for  $i := 1$  to  $n - 1$  do

    allocate a new node  $z$

$left[z] := x := \text{EXTRACT\_MIN}(Q)$

$right[z] := y := \text{EXTRACT\_MIN}(Q)$

$f[z] := f[x] + f[y]$

    INSERT( $Q, z$ )

return EXTRACT\_MIN(Q)

Example:

# Huffman encoding algorithm

Greedy algorithm:

HUFFMAN( $C$ )

$Q := C$

for  $i := 1$  to  $n - 1$  do

    allocate a new node  $z$

$left[z] := x := \text{EXTRACT\_MIN}(Q)$

$right[z] := y := \text{EXTRACT\_MIN}(Q)$

$f[z] := f[x] + f[y]$

    INSERT( $Q, z$ )

return EXTRACT\_MIN(Q)

Example:

$A : 35, B : 13, C : 12, D : 16, E : 9, F : 5$

# Huffman encoding algorithm

Greedy algorithm:

HUFFMAN( $C$ )

$Q := C$

for  $i := 1$  to  $n - 1$  do

    allocate a new node  $z$

$left[z] := x := \text{EXTRACT\_MIN}(Q)$

$right[z] := y := \text{EXTRACT\_MIN}(Q)$

$f[z] := f[x] + f[y]$

    INSERT( $Q, z$ )

return EXTRACT\_MIN(Q)

Example:

$A : 35, B : 13, C : 12, D : 16, E : 9, F : 5$

Running time:  $O(n \log n)$