# Compression, Information and Entropy – Huffman's coding

Lecture 22 November 11, 2014

# Part I

# Huffman coding

- $\bullet$   $\Sigma$ : alphabet.
- binary code: assigns a string of 0s and 1s to each character in the alphabet.
- $\odot$  each symbol in input = a codeword over some other alphabet.
- ullet Useful for transmitting messages over a wire: only 0/1
- receiver gets a binary stream of bits...
- o ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- Ontinuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- Open ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- **binary code**: assigns a string of 0s and 1s to each character in the alphabet.
- $\odot$  each symbol in input = a codeword over some other alphabet.
- ① Useful for transmitting messages over a wire: only 0/1
- 5 receiver gets a binary stream of bits...
- o ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- Ontinuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- **binary code**: assigns a string of 0s and 1s to each character in the alphabet.
- $oldsymbol{0}$  each symbol in input = a codeword over some other alphabet.
- $\odot$  Useful for transmitting messages over a wire: only 0/1
- 5 receiver gets a binary stream of bits...
- o ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- Ontinuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- **binary code**: assigns a string of 0s and 1s to each character in the alphabet.
- $oldsymbol{0}$  each symbol in input = a codeword over some other alphabet.
- Useful for transmitting messages over a wire: only 0/1.
- receiver gets a binary stream of bits...
- ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- ... continuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- **binary code**: assigns a string of 0s and 1s to each character in the alphabet.
- $oldsymbol{0}$  each symbol in input = a codeword over some other alphabet.
- **①** Useful for transmitting messages over a wire: only 0/1.
- receiver gets a binary stream of bits...
- o ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- Ontinuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- binary code: assigns a string of 0s and 1s to each character in the alphabet.
- each symbol in input = a codeword over some other alphabet.
- Useful for transmitting messages over a wire: only 0/1.
- o receiver gets a binary stream of bits...
- ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- Ontinuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- Open ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- binary code: assigns a string of 0s and 1s to each character in the alphabet.
- each symbol in input = a codeword over some other alphabet.
- Useful for transmitting messages over a wire: only 0/1.
- o receiver gets a binary stream of bits...
- ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- Ontinuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bullet$   $\Sigma$ : alphabet.
- binary code: assigns a string of 0s and 1s to each character in the alphabet.
- each symbol in input = a codeword over some other alphabet.
- Useful for transmitting messages over a wire: only 0/1.
- o receiver gets a binary stream of bits...
- ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- ... continuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

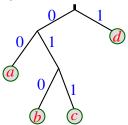
- $\bullet$   $\Sigma$ : alphabet.
- binary code: assigns a string of 0s and 1s to each character in the alphabet.
- each symbol in input = a codeword over some other alphabet.
- **①** Useful for transmitting messages over a wire: only 0/1.
- receiver gets a binary stream of bits...
- ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- ... continuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- $\bigcirc$   $\Sigma$ : alphabet.
- binary code: assigns a string of 0s and 1s to each character in the alphabet.
- $oldsymbol{0}$  each symbol in input = a codeword over some other alphabet.
- ullet Useful for transmitting messages over a wire: only 0/1.
- o receiver gets a binary stream of bits...
- ... decode the message sent.
- prefix code: reading a prefix of the input binary string uniquely match it to a code word.
- ... continuing to decipher the rest of the stream.
- binary/prefix code is prefix-free if no code is a prefix of any other.
- ASCII and Unicode's UTF-8 are both prefix-free binary codes.

- Morse code is binary+prefix code but not prefix-free.
- $\bigcirc$  ... code for S  $(\cdots)$  includes the code for E  $(\cdot)$  as a prefix.
- Prefix codes are binary trees...
- ...characters in leafs, code word is path from root
- oprefix treestree!prefix tree or code trees.
- Open Decoding / Encoding is easy.

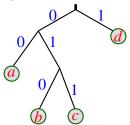
- Morse code is binary+prefix code but not prefix-free.
- Prefix codes are binary trees...
- ...characters in leafs, code word is path from root
- prefix treestree!prefix tree or code trees
- Operation of the property of the second of the property of

- Morse code is binary+prefix code but not prefix-free.
- Prefix codes are binary trees...



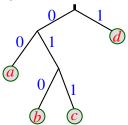
- ...characters in leafs, code word is path from root.
- prefix treestree!prefix tree or code trees
- Open Decoding / Property is easy.

- Morse code is binary+prefix code but **not** prefix-free.
- Prefix codes are binary trees...



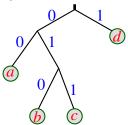
- ...characters in leafs, code word is path from root.
- prefix treestree!prefix tree or code trees
- Open Decoding / Person of the property of t

- Morse code is binary+prefix code but **not** prefix-free.
- Prefix codes are binary trees...



- ...characters in leafs, code word is path from root.
- prefix treestree!prefix tree or code trees.
- Open Decoding / Person of the property of t

- Morse code is binary+prefix code but not prefix-free.
- Prefix codes are binary trees...



- ...characters in leafs, code word is path from root.
- prefix treestree!prefix tree or *code trees*.
- Decoding/encoding is easy.

- Encoding: given frequency table:  $f[1 \dots n]$ .
- code(i): binary string for ith character. len(s): length (in bits) of binary string s.
- Compute tree T that minimizes

$$cost(\mathsf{T}) = \sum_{i=1}^{n} f[i] * len(code(i)), \qquad (1)$$

- Encoding: given frequency table:  $f[1 \dots n]$ .
- ode(i): binary string for ith character. len(s): length (in bits) of binary string s.
- Compute tree T that minimizes

$$cost(\mathbf{T}) = \sum_{i=1}^{n} f[i] * len(code(i)), \qquad (1)$$

- Encoding: given frequency table:  $f[1 \dots n]$ .
- 3 code(i): binary string for ith character. len(s): length (in bits) of binary string s.
- Compute tree T that minimizes

$$cost(\mathsf{T}) = \sum_{i=1}^{n} f[i] * len(code(i)), \qquad (1)$$

- Encoding: given frequency table:  $f[1 \dots n]$ .
- len(s): length (in bits) of binary string s.
- Compute tree T that minimizes

$$cost(\mathsf{T}) = \sum_{i=1}^{n} f[i] * len(code(i)), \qquad (1)$$

CS573 Fall 2014

# Frequency table for...

"A tale of two cities" by Dickens

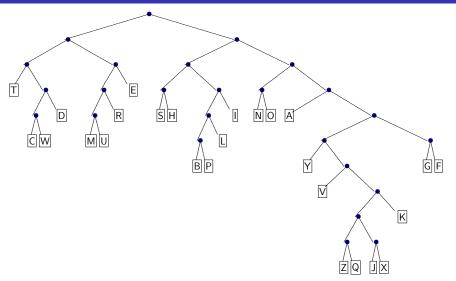
\ n	16,492	'1'	61	'C'	13,896	'Q'	667	ì
, ,	130,376	'2'	10	'D'	28,041	'R'	37,187	ì
'!'	955	'3'	12	'Ε'	74,809	'S'	37,575	ì
4111	5,681	'4'	10	'F'	13,559	'T'	54,024	ì
'\$'	2	'5'	14	'G'	12,530	'U'	16,726	ì
'%'	1	'6'	11	'H'	38,961	'V'	5,199	ì
411	1,174	'7'	13	'1'	41,005	'W'	14,113	ì
'('	151	'8'	13	'J'	710	'X'	724	ì
')'	151	'9'	14	'K'	4,782	'Y'	12,177	ì
' <b>*</b> '	70	':'	267	'L'	22,030	'Z'	215	ì
٠,	13,276	٠.,	1,108	'M'	15,298	-	182	ì
' <u>_</u> '	2,430	'?'	913	'N'	42,380	,,,	93	ì
.,	6,769	'A'	48,165	'O'	46,499	'@'	2	ì
'0'	20	'B'	8,414	'P'	9,957	'/'	26	ì

# Computed prefix codes...

char	frequency	code	char	freq	code
'A'	48165	1110	'N'	42380	1100
'B'	8414	101000	'O'	46499	1101
'C'	13896	00100	'P'	9957	101001
'D'	28041	0011	'Q'	667	1111011001
'E'	74809	011	'R'	37187	0101
'F'	13559	111111	'S'	37575	1000
'G'	12530	111110	'T'	54024	000
'H'	38961	1001	'U'	16726	01001
'1'	41005	1011	'V'	5199	1111010
'J'	710	1111011010	'W'	14113	00101
'K'	4782	11110111	'X'	724	1111011011
L'	22030	10101	'Y'	12177	111100
'M'	15298	01000	ʻZ'	215	1111011000

# The Huffman tree generating the code

Build only on A-Z for clarity.



- two trees for some disjoint parts of the alphabet...
- Merge into larger tree by creating a new node and hanging the trees from this common node.



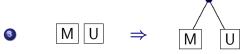


- two trees for some disjoint parts of the alphabet...
- Merge into larger tree by creating a new node and hanging the trees from this common node.



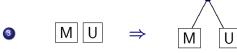


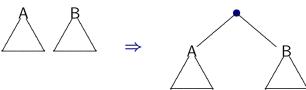
- two trees for some disjoint parts of the alphabet...
- Merge into larger tree by creating a new node and hanging the trees from this common node.





- two trees for some disjoint parts of the alphabet...
- Merge into larger tree by creating a new node and hanging the trees from this common node.





- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- 3 ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

# Building optimal prefix code trees

- take two least frequent characters in frequency table...
- ... merge them into a tree, and put the root of merged tree back into table.
- ...instead of the two old trees.
- Algorithm stops when there is a single tree.
- Intuition: infrequent characters participate in a large number of merges. Long code words.
- Algorithm is due to David Huffman (1952).
- Resulting code is best one can do.
- Huffman coding: building block used by numerous other compression algorithms.

#### Lemma

- **1** T: optimal code tree (prefix free!).
- 2 Then T is a full binary tree
- $oxed{0}$  ... every node of  $oxed{T}$  has either  $oxed{0}$  or  $oxed{2}$  children
- lacksquare If height of lacksquare is  $oldsymbol{d}$ , then there are leafs nodes of height  $oldsymbol{d}$  that are sibling.

#### Lemma

- **1** T: optimal code tree (prefix free!).
- 2 Then T is a full binary tree.
- ${ t 0} \, \dots \,$  every node of  ${ t T} \,$  has either  ${ t 0} \,$  or  ${ t 2} \,$  children
- lacksquare If height of lacksquare is  $oldsymbol{d}$ , then there are leafs nodes of height  $oldsymbol{d}$  that are sibling.

#### Lemma

- **1** T: optimal code tree (prefix free!).
- 2 Then T is a full binary tree.
- ... every node of T has either 0 or 2 children.
- **9** If height of **T** is d, then there are leafs nodes of height d that are sibling.

#### Lemma

- **1** T: optimal code tree (prefix free!).
- 2 Then T is a full binary tree.
- ... every node of T has either 0 or 2 children.
- If height of T is d, then there are leafs nodes of height d that are sibling.

- lacktriangled If  $\exists$  internal node  $v \in \mathbf{V}(\mathbf{T})$  with single child...
- ② New code tree is better compressor:  $cost(T) = \sum_{i=1}^{n} f[i] * len(code(i)).$
- $oldsymbol{u}$ : leaf u with maximum depth d in  $oldsymbol{\mathsf{T}}$ . Consider parent  $v=\overline{\mathbf{p}}(u)$ .
- $\bigcirc$   $\Longrightarrow$  v: has two children, both leafs

- If  $\exists$  internal node  $v \in V(T)$  with single child... ...remove it.
- 2 New code tree is better compressor:  $cost(T) = \sum_{i=1}^{n} f[i] * len(code(i)).$
- $oldsymbol{u}$ : leaf u with maximum depth d in  $oldsymbol{\mathsf{T}}$ . Consider parent  $v=\overline{\mathbf{p}}(u)$ .
- $\bigcirc$   $\Longrightarrow$  v: has two children, both leafs

- **1** If  $\exists$  internal node  $v \in V(T)$  with single child... ...remove it.
- ② New code tree is better compressor:  $cost(T) = \sum_{i=1}^{n} f[i] * len(code(i)).$
- $oldsymbol{u}$ : leaf  $oldsymbol{u}$  with maximum depth  $oldsymbol{d}$  in  $oldsymbol{\mathsf{T}}$ . Consider parent  $oldsymbol{v} = \overline{\mathbf{p}}(oldsymbol{u}).$
- $\longrightarrow v$ : has two children, both leafs

- If  $\exists$  internal node  $v \in V(T)$  with single child... ...remove it.
- ② New code tree is better compressor:  $cost(T) = \sum_{i=1}^{n} f[i] * len(code(i)).$
- $oldsymbol{u}$ : leaf u with maximum depth d in  $oldsymbol{\mathsf{T}}$ . Consider parent  $v=\overline{\mathbf{p}}(u)$ .
- $\longrightarrow v$ : has two children, both leafs

- If  $\exists$  internal node  $v \in V(T)$  with single child... ...remove it.
- New code tree is better compressor:  $cost(\mathsf{T}) = \sum_{i=1}^{n} f[i] * len(code(i)).$
- $\bullet$  u: leaf u with maximum depth d in T. Consider parent  $v=\overline{\mathbf{p}}(u)$ .
- $\bullet \longrightarrow v$ : has two children, both leafs

- If  $\exists$  internal node  $v \in V(T)$  with single child... ...remove it.
- New code tree is better compressor:  $cost(T) = \sum_{i=1}^{n} f[i] * len(code(i)).$
- 3 u: leaf u with maximum depth d in T. Consider parent  $v = \overline{p}(u)$ .
- $\bullet$   $\Longrightarrow$  v: has two children, both leafs

# Infrequent characters are stuck together...

 $\exists$  optimal code tree in which x and y are siblings.

#### Lemma

x, y: two least frequent characters (breaking ties arbitrarily).

- **①** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- ② T: optimal code tree with depth d.
- ullet By lemma...  ${\sf T}$  has two leafs at depth d that are siblings,
- ① If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- $\circ$   $\mathfrak{I}'$ : swap x and  $\alpha$ .
- $\mathbf{0} \; \operatorname{cost}(\mathfrak{T}') = \operatorname{cost}(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- x: one of the two least frequent characters...but  $\alpha$  is not.
- @ Swapping x and  $\alpha$  does not increase cost.
- $\blacksquare$  T: optimal code tree, swapping x and  $\alpha$  does not decrease cost
- $\mathfrak{D}'$  is also an optimal code tree
- $\blacksquare$  Must be that  $f[\alpha] = f[x]$ .

- **1** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- $\bullet$  T: optimal code tree with depth d.
- $\odot$  By lemma... **T** has two leafs at depth d that are siblings,
- 1 If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- $lacksquare{3}$   $\mathfrak{I}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- lacktriangledown lacktriangledown depth inc by lacktriangledown, and depth of lacktriangledown decreases by lacktriangledown.
- $\bigcirc \ \, \mathrm{cost}(\Im') = \mathrm{cost}(\mathsf{T}) \Big( f[\alpha] f[x] \Big) \, \Delta.$
- x: one of the two least frequent characters.
- @ Swapping x and  $\alpha$  does not increase cost.
- $\blacksquare$  T: optimal code tree, swapping x and  $\alpha$  does not decrease cost
- Must be that  $f[\alpha] = f[x]$ .

- **1** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- 1 If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- $\circ$   $\mathfrak{I}'$ : swap x and  $\alpha$ .
- ullet x depth inc by  $\Delta$ , and depth of lpha decreases by  $\Delta$ .
- $\mathbf{0} \; \operatorname{cost}(\mathfrak{T}') = \operatorname{cost}(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- x: one of the two least frequent characters.but \( \alpha \) is not
- @ Swapping x and  $\alpha$  does not increase cost.

- ${}^{\small{\textcircled{\tiny 0}}}$  Must be that f[lpha]=f[x].

- **1** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- **1** If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- $\circ$   $\mathfrak{I}'$ : swap x and  $\alpha$ .
- lacktriangledown lacktriangledown depth inc by lacktriangledown, and depth of lacktriangledown decreases by lacktriangledown.
- $\bigcirc \ \, \mathrm{cost}(\Im') = \mathrm{cost}(\mathsf{T}) \Big( f[\alpha] f[x] \Big) \, \Delta.$
- x: one of the two least frequent characters. ...but  $\alpha$  is not.
- @ Swapping x and  $\alpha$  does not increase cost.

- ${}^{\small{\textcircled{\tiny 0}}}$  Must be that f[lpha]=f[x].

- **①** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- **②** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- **1** If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacktriangledown  $\mathfrak{T}'$ : swap  $\boldsymbol{x}$  and  $\boldsymbol{\alpha}$ .
- lacktriangledown lacktriangledown depth inc by lacktriangledown, and depth of lacktriangledown decreases by lacktriangledown.
- $\mathbf{O} \, \cot(\mathfrak{I}') = \cot(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$

- @ Swapping x and  $\alpha$  does not increase cost.
- $\mathfrak{D}'$  is also an optimal code tree
- ${}^{ ext{ iny M}}$  Must be that f[lpha]=f[x].

- **①** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- **1** If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- $lackbox{0} \ \mathfrak{I}'$ : swap  $m{x}$  and  $m{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \cos (\mathfrak{T}') = \cos (\mathsf{T}) \left(f[lpha] f[x]\right) \Delta.$

- @ Swapping x and  $\alpha$  does not increase cost.
- $\mathfrak{D}'$  is also an optimal code tree
- ${}^{ ext{ iny III}}$  Must be that f[lpha]=f[x].

- **1** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- **2** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- **1** If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacksquare  $\mathfrak{T}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \, \cos(\mathfrak{T}') = \cos(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- x: one of the two least frequent characters.
- @ Swapping x and  $\alpha$  does not increase cost.
- $exttt{@}$   $\mathsf{T}$ : optimal code tree, swapping x and lpha does not decrease cost

- Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- **②** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacksquare  $\mathfrak{T}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $oldsymbol{\circ} \cos \mathrm{t}(\mathfrak{T}') = \mathrm{cost}(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- $oldsymbol{x}$ : one of the two least frequent characters. ...but  $oldsymbol{lpha}$  is not.
- @ Swapping x and  $\alpha$  does not increase cost.
- $\blacksquare$  T: optimal code tree, swapping x and  $\alpha$  does not decrease cost
- ${}^{ ext{ iny M}}$  Must be that f[lpha]=f[x].

- **①** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- $\bullet$  T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacksquare  $\mathfrak{T}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \cos (\mathfrak{T}') = \cos (\mathsf{T}) \Big(f[lpha] f[x]\Big) \Delta.$
- $oldsymbol{x}$ : one of the two least frequent characters. ...but  $oldsymbol{lpha}$  is not.
- @ Swapping x and lpha does not increase cost.
- $\blacksquare$  T: optimal code tree, swapping x and  $\alpha$  does not decrease cost
- $\mathfrak{D}'$  is also an optimal code tree
- Must be that  $f[\alpha] = f[x]$ .

- **1** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- **2** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacksquare  $\mathfrak{T}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \, \cos(\mathfrak{T}') = \cos(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- $oldsymbol{x}$ : one of the two least frequent characters. ...but  $oldsymbol{lpha}$  is not.
- **1** Swapping x and  $\alpha$  does not increase cost.

- ${}_{lacktriangledown}$  Must be that f[lpha]=f[x].

- **①** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- **2** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacksquare  $\mathfrak{T}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \, \cos(\mathfrak{T}') = \cos(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- x: one of the two least frequent characters.
   ...but α is not.
- lacktriangle Swapping  $oldsymbol{x}$  and  $oldsymbol{lpha}$  does not increase cost.
- **1** T: optimal code tree, swapping x and  $\alpha$  does not decrease cost.
- $\mathfrak{D}'$  is also an optimal code tree
- ${}^{ ext{ iny M}}$  Must be that f[lpha]=f[x].

- **1** Claim:  $\exists$  optimal code s.t. x and y are siblings + deepest.
- **2** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- $lackbox{1}{\mathfrak{I}}$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \, \cos(\mathfrak{T}') = \cos(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- x: one of the two least frequent characters.
   ...but α is not.
- lacktriangle Swapping  $oldsymbol{x}$  and  $oldsymbol{lpha}$  does not increase cost.
- f 0 f T: optimal code tree, swapping m x and m lpha does not decrease cost.
- $extbf{@} extbf{$\mathfrak{I}'$}$  is also an optimal code tree

- lacktriangle Claim: eta optimal code s.t.  $m{x}$  and  $m{y}$  are siblings + deepest.
- **2** T: optimal code tree with depth d.
- lacktriangle By lemma... lacktriangle has two leafs at depth d that are siblings,
- If not x and y, but some other characters  $\alpha$  and  $\beta$ .
- lacksquare  $\mathfrak{T}'$ : swap  $oldsymbol{x}$  and  $oldsymbol{lpha}$ .
- **1** x depth inc by  $\Delta$ , and depth of  $\alpha$  decreases by  $\Delta$ .
- $\mathbf{O} \, \cos(\mathfrak{T}') = \cos(\mathsf{T}) \Big(f[lpha] f[x]\Big) \, \Delta.$
- $oldsymbol{x}$ : one of the two least frequent characters. ...but  $oldsymbol{lpha}$  is not.
- lacktriangle Swapping  $oldsymbol{x}$  and  $oldsymbol{lpha}$  does not increase cost.
- **1** T: optimal code tree, swapping x and  $\alpha$  does not decrease cost.
- $oldsymbol{\mathfrak{D}}'$  is also an optimal code tree
- $\blacksquare$  Must be that  $f[\alpha] = f[x]$ .

- **1 y**: second least frequent character.
- ②  $\beta$ : lowest leaf in tree. Sibling to x.
- ① Swapping y and  $\beta$  must give yet another optimal code tree.
- ullet Final opt code tree,  $oldsymbol{x}, oldsymbol{y}$  are max-depth siblings.

- **1 y**: second least frequent character.
- $\mathfrak{D}$ : lowest leaf in tree. Sibling to x.
- ① Swapping y and  $\beta$  must give yet another optimal code tree.
- ullet Final opt code tree, x, y are max-depth siblings.

- **1 y**: second least frequent character.
- $\bigcirc$   $\beta$ : lowest leaf in tree. Sibling to x.
- **3** Swapping y and  $\beta$  must give yet another optimal code tree.
- ullet Final opt code tree, x,y are max-depth siblings.

- **1 y**: second least frequent character.
- $\bigcirc$   $\beta$ : lowest leaf in tree. Sibling to x.
- **3** Swapping y and  $\beta$  must give yet another optimal code tree.
- ullet Final opt code tree,  $oldsymbol{x}, oldsymbol{y}$  are max-depth siblings.

# Huffman's codes are optimal

#### Theorem

Huffman codes are optimal prefix-free binary codes.

- lacktriangle If message has lacktriangle or lacktriangle diff characters, then theorem easy.
- $oldsymbol{0}$  Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathfrak{T}_{\mathrm{opt}}$  for f[1..n]
- extstyle ext
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

- If message has 1 or 2 diff characters, then theorem easy.
- ullet Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathcal{T}_{ ext{opt}}$  for f[1..n]
- $\hbox{ $ @ $ Remove 1 and 2 from $ $ $ $ T_{opt}. $ }$
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

- lacktriangle If message has lacktriangle or lacktriangle diff characters, then theorem easy.
- **3** Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\Longrightarrow$   $\exists$  opt. code tree  $\mathfrak{T}_{\mathrm{opt}}$  for f[1..n]
- ${\color{red} \textbf{0}} \ \ \mathsf{Remove} \ \mathbf{1} \ \mathsf{and} \ \mathbf{2} \ \mathsf{from} \ \mathbf{\mathfrak{T}_{opt}}.$
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

- lacktriangle If message has lacktriangle or lacktriangle diff characters, then theorem easy.
- **3** Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathcal{T}_{\mathrm{opt}}$  for f[1..n]
- lacksquare Remove 1 and 2 from lacksquare\_opt.
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

- lacktriangle If message has lacktriangle or lacktriangle diff characters, then theorem easy.
- **3** Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathfrak{T}_{\mathrm{opt}}$  for f[1..n]
- $\bigcirc$  Remove 1 and 2 from  $\Im_{\mathrm{opt}}$ .
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

- lacktriangle If message has lacktriangle or lacktriangle diff characters, then theorem easy.
- **3** Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathfrak{T}_{\mathrm{opt}}$  for f[1..n]
- $oldsymbol{\mathfrak{I}}_{\mathrm{opt}}$  has  $oldsymbol{1}$  and  $oldsymbol{2}$  as siblings.
- $\bigcirc$  Remove 1 and 2 from  $\bigcirc$ Opt.
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

#### Proof...

- lacktriangle If message has lacktriangle or lacktriangle diff characters, then theorem easy.
- **3** Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathfrak{T}_{\mathrm{opt}}$  for f[1..n]
- $oldsymbol{\mathfrak{I}}_{\mathrm{opt}}$  has  $oldsymbol{1}$  and  $oldsymbol{2}$  as siblings.
- **9** Remove 1 and 2 from  $\mathfrak{T}_{opt}$ .
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3, \ldots, n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

#### Proof...

- $lue{0}$  If message has 1 or 2 diff characters, then theorem easy.
- **3** Assume f[1] and f[2] are the two smallest.
- Let f[n+1] = f[1] + f[2].
- lacksquare lemma  $\implies \exists$  opt. code tree  $\mathfrak{T}_{\mathrm{opt}}$  for f[1..n]
- $oldsymbol{\mathfrak{I}}_{\mathrm{opt}}$  has  $oldsymbol{1}$  and  $oldsymbol{2}$  as siblings.
- lacksquare Remove 1 and 2 from  $\mathfrak{T}_{\mathrm{opt}}.$
- $\mathfrak{I}'_{\mathrm{opt}}$ : Remaining tree has  $3,\ldots,n$  as leafs and "special" character n+1 (i.e., parent 1,2 in  $\mathfrak{I}_{\mathrm{opt}}$ )

• character n+1: has frequency f[n+1]. Now, f[n+1] = f[1] + f[2], we have  $\mathrm{cost}(\mathfrak{T}_{\mathrm{opt}}) = \sum^n f[i] \mathrm{depth}_{\mathfrak{T}_{\mathrm{opt}}}(i)$  $= \cos \left( {{ extstyle 5}'_{ ext{opt}}} 
ight) + \left( {f[1] + f[2]} 
ight) \operatorname{depth} ({{ extstyle 5}_{ ext{opt}}})$  $-ig(f[1]+f[2]ig)(\operatorname{depth}(\mathfrak{T}_{\mathrm{opt}})-1)$  $= \cos\left(\mathfrak{I}_{\mathrm{opt}}'\right) + f[1] + f[2].$ 

• character n+1: has frequency f[n+1]. Now, f[n+1] = f[1] + f[2], we have  $\mathrm{cost}(\mathcal{T}_{\mathrm{opt}}) = \sum_{i=1}^{n} f[i] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i)$  $= \quad \sum f[i] \mathrm{depth}_{\mathcal{I}_{\mathrm{opt}}}(i) + f[1] \mathrm{depth}_{\mathcal{I}_{\mathrm{opt}}}(1)$  $+f[2] \operatorname{depth}_{\mathcal{T}_{out}}(2) - f[n+1] \operatorname{depth}_{\mathcal{T}_{out}}(n+1)$  $= \cos \left( \mathfrak{T}_{\mathrm{opt}}' \right) + \left( f[1] + f[2] \right) \operatorname{depth}(\mathfrak{T}_{\mathrm{opt}})$  $-\Big(f[1]+f[2]\Big)(\operatorname{depth}(\mathfrak{T}_{\mathrm{opt}})-1)$  $= \cos \left( \mathfrak{I}_{\mathrm{opt}}' \right) + f[1] + f[2].$ 

• character n+1: has frequency f[n+1]. Now, f[n+1] = f[1] + f[2], we have  $\mathrm{cost}(\mathcal{T}_{\mathrm{opt}}) = \sum_{i=1}^{n} f[i] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i)$  $=\sum_{i=1}^{n-1}f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i)+f[1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1)$  $+f[2] \operatorname{depth}_{\mathcal{T}_{\mathrm{out}}}(2) - f[n+1] \operatorname{depth}_{\mathcal{T}_{\mathrm{out}}}(n+1)$  $= \cos \left( {rac{{{ extstyle extstyle }}}{{{ extstyle opt}}}} 
ight) + \left( {f[1] + f[2]} 
ight) \operatorname{depth} ({rac{{{ extstyle opt}}}{{{ extstyle opt}}}})$  $-ig(f[1]+f[2]ig)(\operatorname{depth}(\mathfrak{T}_{\mathrm{opt}})-1)$  $= \cos\left(\mathfrak{I}_{\mathrm{opt}}'\right) + f[1] + f[2].$ 

• character n+1: has frequency f[n+1]. Now, f[n+1] = f[1] + f[2], we have  $\mathrm{cost}(\mathfrak{T}_{\mathrm{opt}}) = \sum_{i=1}^{n} f[i] \mathrm{depth}_{\mathfrak{T}_{\mathrm{opt}}}(i)$  $=\sum_{i=1}^{n-1} f[i] ext{depth}_{\mathcal{T}_{ ext{opt}}}(i) + f[1] ext{depth}_{\mathcal{T}_{ ext{opt}}}(1)$  $+f[2] \operatorname{depth}_{\mathcal{T}_{out}}(2) - f[n+1] \operatorname{depth}_{\mathcal{T}_{out}}(n+1)$  $= \cos \left( {rac{{{ extstyle extstyle }}}{{{ extstyle opt}}}} 
ight) + \left( {f[1] + f[2]} 
ight) \operatorname{depth} ({rac{{{ extstyle opt}}}{{{ extstyle opt}}}})$  $-\Big(f[1]+f[2]\Big)(\operatorname{depth}(\mathfrak{T}_{\mathrm{opt}})-1)$  $= \cos \left( \Im_{\mathrm{opt}}' \right) + f[1] + f[2].$ 

- lacksquare implies  $\min$  cost of  $\mathcal{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathcal{T}'_{\mathrm{opt}}$ .
- ②  $\mathfrak{I}'_{\mathrm{opt}}$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H'$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- ① By construction:  $\mathfrak{I}_H'$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- Sy induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} oldsymbol{\odot} &\Longrightarrow & ext{cost}(\mathsf{T}_H) = ext{cost}ig(\mathfrak{I}_H'ig) + f[1] + f[2] = \ & ext{cost}ig(\mathfrak{I}_{ ext{opt}}'ig) + f[1] + f[2] = ext{cost}(\mathfrak{I}_{ ext{opt}}) \ , \end{aligned}$
- Huffman tree has the same cost as the optimal tree.

- lacksquare implies  $\min$  cost of  $\mathcal{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathcal{T}'_{\mathrm{opt}}$ .
- $\mathfrak{D}_{\mathrm{opt}}'$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H'$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- ① By construction:  $\mathfrak{I}_H'$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- By induction:Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} oldsymbol{\odot} &\Longrightarrow & ext{cost}(\mathsf{T}_H) = & ext{cost}ig(\mathfrak{T}_H'ig) + f[1] + f[2] = \ & ext{cost}ig(\mathfrak{T}_{ ext{opt}}'ig) + f[1] + f[2] = & ext{cost}(\mathfrak{T}_{ ext{opt}}) \ , \end{aligned}$
- Huffman tree has the same cost as the optimal tree.

- lacksquare implies  $\min$  cost of  $\mathcal{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathcal{T}'_{\mathrm{opt}}$ .
- $\mathfrak{D}_{\mathrm{opt}}'$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H^{\prime}$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- lacktriangledown By construction:  $\mathfrak{I}'_H$  formed by removing leafs f 1 and f 2 from  $f T_H$ .
- Sy induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} oldsymbol{\odot} &\Longrightarrow ext{cost}(\mathsf{T}_H) = ext{cost}ig( \mathfrak{I}_H' ig) + f[1] + f[2] = \ ext{cost}ig( \mathfrak{I}_{ ext{opt}}' ig) + f[1] + f[2] = ext{cost}( \mathfrak{I}_{ ext{opt}}) \,, \end{aligned}$
- Huffman tree has the same cost as the optimal tree.

- lacksquare implies  $\min$  cost of  $\mathcal{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathcal{T}'_{\mathrm{opt}}$ .
- $\mathfrak{D}_{\mathrm{opt}}'$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H^{\prime}$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- **9** By construction:  $\mathfrak{I}'_H$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- ullet By induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} oldsymbol{\odot} &\Longrightarrow & ext{cost}(\mathsf{T}_H) = & ext{cost}ig(\mathfrak{T}_H'ig) + f[1] + f[2] = \ & ext{cost}ig(\mathfrak{T}_{ ext{opt}}'ig) + f[1] + f[2] = & ext{cost}(\mathfrak{T}_{ ext{opt}}) \ , \end{aligned}$
- Huffman tree has the same cost as the optimal tree.

- lacksquare implies  $\min$  cost of  $\mathfrak{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathfrak{T}'_{\mathrm{opt}}.$
- ②  $\mathfrak{I}'_{\mathrm{opt}}$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H'$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- **9** By construction:  $\mathfrak{I}'_H$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- ullet By induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} egin{aligned} igotimes & \cot(\mathsf{T}_H) = \cot\left(\mathfrak{T}_H'
  ight) + f[1] + f[2] = \ \cot\left(\mathfrak{T}_{\mathrm{opt}}'
  ight) + f[1] + f[2] = \cot(\mathfrak{T}_{\mathrm{opt}}) \,, \end{aligned}$
- Huffman tree has the same cost as the optimal tree.

- lacksquare implies  $\min$  cost of  $\mathfrak{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathfrak{T}'_{\mathrm{opt}}.$
- $\mathfrak{D}'_{\mathrm{opt}}$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H'$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- **9** By construction:  $\mathfrak{I}'_H$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- ullet By induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} egin{aligned} igotimes & \cot(\mathsf{T}_H) = \cot\left(\mathfrak{T}_H'
  ight) + f[1] + f[2] = \ \cot\left(\mathfrak{T}_{\mathrm{opt}}'
  ight) + f[1] + f[2] = \cot(\mathfrak{T}_{\mathrm{opt}}) \,, \end{aligned}$
- Huffman tree has the same cost as the optimal tree.

- lacksquare implies  $\min$  cost of  $\mathfrak{T}_{\mathrm{opt}} \equiv \min$  cost  $\mathfrak{T}'_{\mathrm{opt}}.$
- $oldsymbol{\mathfrak{I}}_{\mathrm{opt}}^{\prime}$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H'$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- **9** By construction:  $\mathfrak{I}'_H$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- ullet By induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} egin{aligned} &\Longrightarrow & \cos (\mathsf{T}_H) = \cos \left( \mathcal{T}_H' 
  ight) + f[1] + f[2] = \ & \cos \left( \mathcal{T}_{\mathrm{opt}}' 
  ight) + f[1] + f[2] = & \cos \left( \mathcal{T}_{\mathrm{opt}} 
  ight), \end{aligned}$
- Muffman tree has the same cost as the optimal tree.

- lacktriangledown implies  $\min$  cost of  $\mathfrak{T}_{\mathrm{opt}}' \equiv \min$  cost  $\mathfrak{T}_{\mathrm{opt}}'$ .
- $\mathfrak{D}'_{\mathrm{opt}}$ : must be optimal coding tree for  $f[3\ldots n+1]$ .
- $\mathfrak{I}_H^{\prime}$ : Huffman tree for  $f[3,\ldots,n+1]$   $\mathsf{T}_H$ : overall Huffman tree constructed for  $f[1,\ldots,n]$ .
- **9** By construction:  $\mathfrak{I}'_H$  formed by removing leafs 1 and 2 from  $\mathsf{T}_H$ .
- ullet By induction: Huffman tree generated for  $f[3,\ldots,n+1]$  is optimal.
- $egin{aligned} egin{aligned} &\Longrightarrow & \cos t(\mathsf{T}_H) = \cos tig(\mathfrak{T}_H'ig) + f[1] + f[2] = \ & \cos tig(\mathfrak{T}_{\mathrm{opt}}'ig) + f[1] + f[2] = & \cos t(\mathfrak{T}_{\mathrm{opt}}) \,, \end{aligned}$
- $\bigcirc$   $\Longrightarrow$  Huffman tree has the same cost as the optimal tree.

- A tale of two cities: 779,940 bytes.
- using above Huffman compression results in a compression to a file of size 439,688 bytes.
- Ignoring space to store tree.
- gzip: 301,295 bytes bzip2: 220,156 bytes!
- Huffman encoder can be easily written in a few hours of work!
- 6 All later compressors use it as a black box...

- A tale of two cities: 779,940 bytes.
- using above Huffman compression results in a compression to a file of size 439,688 bytes.
- Ignoring space to store tree.
- gzip: 301,295 bytes bzip2: 220,156 bytes!
- Huffman encoder can be easily written in a few hours of work!
- 6 All later compressors use it as a black box...

- A tale of two cities: 779,940 bytes.
- using above Huffman compression results in a compression to a file of size 439,688 bytes.
- Ignoring space to store tree.
- gzip: 301,295 bytes bzip2: 220,156 bytes!
- Muffman encoder can be easily written in a few hours of work!
- 6 All later compressors use it as a black box...

- A tale of two cities: 779,940 bytes.
- using above Huffman compression results in a compression to a file of size 439,688 bytes.
- Ignoring space to store tree.
- gzip: 301,295 bytes bzip2: 220,156 bytes!
- Huffman encoder can be easily written in a few hours of work!
- All later compressors use it as a black box...

- A tale of two cities: 779,940 bytes.
- using above Huffman compression results in a compression to a file of size 439,688 bytes.
- Ignoring space to store tree.
- gzip: 301,295 bytes bzip2: 220,156 bytes!
- Huffman encoder can be easily written in a few hours of work!
- All later compressors use it as a black box...

- A tale of two cities: 779,940 bytes.
- using above Huffman compression results in a compression to a file of size 439,688 bytes.
- Ignoring space to store tree.
- gzip: 301,295 bytes bzip2: 220,156 bytes!
- Huffman encoder can be easily written in a few hours of work!
- All later compressors use it as a black box...

- $oldsymbol{0}$  input is made out of n characters.
- $p_i$ : fraction of input that is ith char (probability).
- use probabilities to build Huffman tree
- Q: What is the length of the codewords assigned to characters as function of probabilities?
- special case...

- $oldsymbol{0}$  input is made out of  $oldsymbol{n}$  characters.
- $p_i$ : fraction of input that is ith char (probability).
- use probabilities to build Huffman tree
- Q: What is the length of the codewords assigned to characters as function of probabilities?
- special case...

- $oldsymbol{0}$  input is made out of  $oldsymbol{n}$  characters.
- $p_i$ : fraction of input that is ith char (probability).
- use probabilities to build Huffman tree.
- Q: What is the length of the codewords assigned to characters as function of probabilities?
- special case...

- $oldsymbol{0}$  input is made out of  $oldsymbol{n}$  characters.
- $p_i$ : fraction of input that is ith char (probability).
- use probabilities to build Huffman tree.
- Q: What is the length of the codewords assigned to characters as function of probabilities?
- special case...

- input is made out of n characters.
- $p_i$ : fraction of input that is ith char (probability).
- use probabilities to build Huffman tree.
- Q: What is the length of the codewords assigned to characters as function of probabilities?
- special case...

# Average length of codewords...

Special case

#### Lemma

```
1, \ldots, n: symbols.
```

Assume, for  $i = 1, \ldots, n$ :

- $p_i = 1/2^{l_i}$ : probability for the *i*th symbol
- $\mathbf{0} \ \mathbf{l}_i \geq \mathbf{0}$ : integer.

Then, in Huffman coding for this input, the code for i is of length  $l_i$ .

- induction of the Huffman algorithm.
- n=2: claim holds since there are only two characters with probability 1/2.
- ${ top}$  Let i and j be the two characters with lowest probability.
- lacksquare Must be  $p_i=p_j$  (otherwise,  $\sum_k p_k 
  eq 1$ ).
- ullet Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i-1$ , by induction (on remaining n-1 symbols).
- $m{\circ}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- induction of the Huffman algorithm.
- ② n=2: claim holds since there are only two characters with probability 1/2.
- ${ top}$  Let i and j be the two characters with lowest probability.
- lacksquare Must be  $p_i=p_j$  (otherwise,  $\sum_k p_k 
  eq 1$ ).
- ullet Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i-1$ , by induction (on remaining n-1 symbols).
- $m{0}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- induction of the Huffman algorithm.
- ② n=2: claim holds since there are only two characters with probability 1/2.
- **3** Let i and j be the two characters with lowest probability.
- ① Must be  $p_i = p_j$  (otherwise,  $\sum_k p_k \neq 1$ ).
- ullet Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i-1$ , by induction (on remaining n-1 symbols).
- $m{0}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- induction of the Huffman algorithm.
- ② n=2: claim holds since there are only two characters with probability 1/2.
- lacksquare Let i and j be the two characters with lowest probability.
- lacksquare Must be  $p_i=p_j$  (otherwise,  $\sum_k p_k 
  eq 1$ ).
- ullet Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i-1$ , by induction (on remaining n-1 symbols).
- $m{0}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- induction of the Huffman algorithm.
- ② n=2: claim holds since there are only two characters with probability 1/2.
- lacksquare Let i and j be the two characters with lowest probability.
- lacksquare Must be  $p_i=p_j$  (otherwise,  $\sum_k p_k 
  eq 1$ ).
- **1** Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i-1$ , by induction (on remaining n-1 symbols).
- $m{0}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- induction of the Huffman algorithm.
- ② n=2: claim holds since there are only two characters with probability 1/2.
- lacksquare Let i and j be the two characters with lowest probability.
- lacksquare Must be  $p_i=p_j$  (otherwise,  $\sum_k p_k 
  eq 1$ ).
- **1** Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i 1$ , by induction (on remaining n 1 symbols).
- $m{\circ}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- induction of the Huffman algorithm.
- ② n=2: claim holds since there are only two characters with probability 1/2.
- lacksquare Let i and j be the two characters with lowest probability.
- lacksquare Must be  $p_i=p_j$  (otherwise,  $\sum_k p_k 
  eq 1$ ).
- **1** Huffman's tree merges this two letters, into a single "character" that have probability  $2p_i$ .
- New "character" has encoding of length  $l_i 1$ , by induction (on remaining n 1 symbols).
- $m{Q}$  resulting tree encodes i and j by code words of length  $(l_i-1)+1=l_i.$

- $l_i = \lg 1/p_i$ .
- Output
  Average length of a code word is

$$\sum_i p_i \lg rac{1}{p_i}$$

 $oldsymbol{0}$  X is a random variable that takes a value i with probability  $p_i$ , then this formula is

$$\mathbb{H}(X) = \sum_{i} \Pr[X = i] \lg \frac{1}{\Pr[X = i]},$$

which is the entropy of X

- $p_i = 1/2^{l_i}$
- $l_i = \lg 1/p_i.$
- a Average length of a code word is

$$\sum_i p_i \lg rac{1}{p_i}$$

ullet X is a random variable that takes a value i with probability  $p_i$ , then this formula is

$$\mathbb{H}(X) = \sum_{i} \Pr[X = i] \lg \frac{1}{\Pr[X = i]},$$

which is the entropy of X.

- $p_i = 1/2^{l_i}$
- $l_i = \lg 1/p_i.$
- Average length of a code word is

$$\sum_i p_i \lg rac{1}{p_i}.$$

lacktriangledown X is a random variable that takes a value i with probability  $p_i$ , then this formula is

$$\mathbb{H}(X) = \sum_{i} \Pr[X = i] \lg \frac{1}{\Pr[X = i]},$$

which is the entropy of X.

- $l_i = \lg 1/p_i.$
- Average length of a code word is

$$\sum_i p_i \lg rac{1}{p_i}.$$

 $oldsymbol{0}$  X is a random variable that takes a value i with probability  $p_i$ , then this formula is

$$\mathbb{H}(X) = \sum_i \Pr[X=i] \lg rac{1}{\Pr[X=i]},$$

which is the entropy of X.