Compression, Information and Entropy – Huffman's coding

Lecture 22

November 11, 2014

Part I

Huffman coding

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

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

Codes...

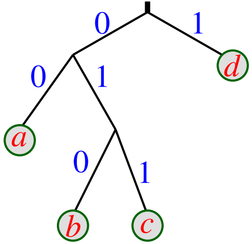
1. Morse code is binary+prefix code but **not** prefix-free.
2. ... code for S ($\cdots$) includes the code for E ($\cdot$) as a prefix.
3. Prefix codes are binary trees...
4. ...characters in leafs, code word is path from root.
5. prefix treestree!prefix tree or ***code trees***.
6. Decoding/encoding is easy.

Codes...

1. Morse code is binary+prefix code but **not** prefix-free.
2. ... code for S ($\cdots$) includes the code for E ($\cdot$) as a prefix.
3. Prefix codes are binary trees...
4. ...characters in leafs, code word is path from root.
5. prefix treestree!prefix tree or ***code trees***.
6. Decoding/encoding is easy.

Codes...

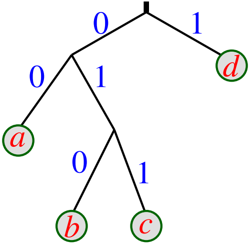
1. Morse code is binary+prefix code but **not** prefix-free.
2. ... code for `S` ($\cdots$) includes the code for `E` ($\cdot$) as a prefix.
3. Prefix codes are binary trees...



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

Codes...

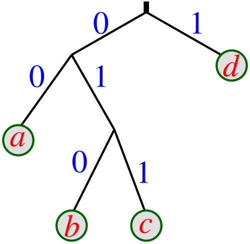
1. Morse code is binary+prefix code but **not** prefix-free.
2. ... code for `S` ($\cdot\cdot\cdot$) includes the code for `E` ($\cdot$) as a prefix.
3. Prefix codes are binary trees...



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

Codes...

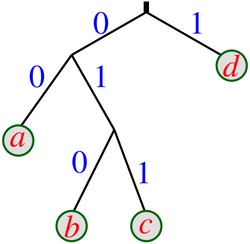
1. Morse code is binary+prefix code but **not** prefix-free.
2. ... code for `S` $(\cdots)$ includes the code for `E` $(\cdot)$ as a prefix.
3. Prefix codes are binary trees...



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

Codes...

1. Morse code is binary+prefix code but **not** prefix-free.
2. ... code for S ($\cdots$) includes the code for E ($\cdot$) as a prefix.
3. Prefix codes are binary trees...



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

Codes...

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

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

Codes...

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

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

Codes...

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

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

Codes...

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

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

Frequency table for...

"A tale of two cities" by Dickens

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

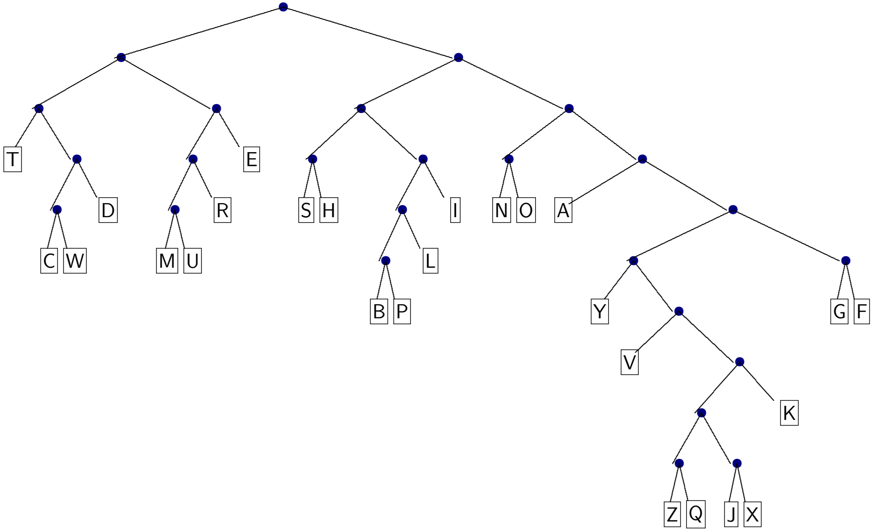
Computed prefix codes...

char	frequency	code
'A'	48165	1110
'B'	8414	101000
'C'	13896	00100
'D'	28041	0011
'E'	74809	011
'F'	13559	111111
'G'	12530	111110
'H'	38961	1001
'I'	41005	1011
'J'	710	1111011010
'K'	4782	11110111
'L'	22030	10101
'M'	15298	01000

char	freq	code
'N'	42380	1100
'O'	46499	1101
'P'	9957	101001
'Q'	667	1111011001
'R'	37187	0101
'S'	37575	1000
'T'	54024	000
'U'	16726	01001
'V'	5199	1111010
'W'	14113	00101
'X'	724	1111011011
'Y'	12177	111100
'Z'	215	1111011000

The Huffman tree generating the code

Build only on A-Z for clarity.



Mergeablity of code trees

1. two trees for some disjoint parts of the alphabet...
2. Merge into larger tree by creating a new node and hanging the trees from this common node.
3.

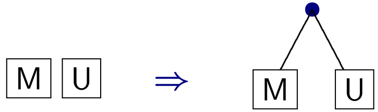
4. ...put together two subtrees.

Mergeablity of code trees

1. two trees for some disjoint parts of the alphabet...
2. Merge into larger tree by creating a new node and hanging the trees from this common node.
3.

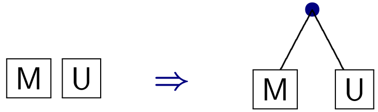
4. ...put together two subtrees.

Mergeablity of code trees

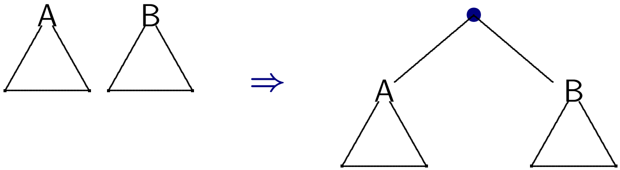
1. two trees for some disjoint parts of the alphabet...
2. Merge into larger tree by creating a new node and hanging the trees from this common node.
3. 

4. ...put together two subtrees.

Mergeablity of code trees

1. two trees for some disjoint parts of the alphabet...
2. Merge into larger tree by creating a new node and hanging the trees from this common node.
3. 

4. ...put together two subtrees.



Building optimal prefix code trees

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

Building optimal prefix code trees

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

Building optimal prefix code trees

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

Building optimal prefix code trees

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

Building optimal prefix code trees

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

Building optimal prefix code trees

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

Building optimal prefix code trees

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

Building optimal prefix code trees

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

Lemma: lowest leafs are siblings...

Lemma

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

Lemma: lowest leafs are siblings...

Lemma

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

Lemma: lowest leafs are siblings...

Lemma

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

Lemma: lowest leafs are siblings...

Lemma

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

Proof...

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

Proof...

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

Proof...

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

Proof...

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

Proof...

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

Proof...

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

$\blacksquare$

Infrequent characters are stuck together...

Lemma

x, y: *two least frequent characters (breaking ties arbitrarily).*
$\exists$ *optimal code tree in which* x *and* y *are siblings.*

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta.$
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. **T**: optimal code tree with depth d.
3. By lemma... **T** has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. **T**: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. **T**: optimal code tree with depth d.
3. By lemma... **T** has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. **T**: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. $\mathbf{T}$: optimal code tree with depth d.
3. By lemma... $\mathbf{T}$ has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathbf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta.$
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. $\mathbf{T}$: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta.$
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. **T**: optimal code tree with depth d.
3. By lemma... **T** has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathbf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. **T**: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. T: optimal code tree with depth d.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\,\Delta.$
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof...

1. Claim: $\exists$ optimal code s.t. x and y are siblings + deepest.
2. **T**: optimal code tree with depth d.
3. By lemma... **T** has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters. ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. **T**: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. 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.
3. By lemma... T has two leafs at depth d that are siblings,
4. If not x and y, but some other characters α and β.
5. $\mathcal{T}'$: swap x and α.
6. x depth inc by Δ, and depth of α decreases by Δ.
7. $\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathsf{T}) - \Big(f[\alpha] - f[x]\Big)\Delta$.
8. x: one of the two least frequent characters.
   ...but α is not.
9. $\implies f[\alpha] \geq f[x]$.
10. Swapping x and α does not increase cost.
11. T: optimal code tree, swapping x and α does not decrease cost.
12. $\mathcal{T}'$ is also an optimal code tree
13. Must be that $f[\alpha] = f[x]$.

Proof continued...

1. y: second least frequent character.
2. β: lowest leaf in tree. Sibling to x.
3. Swapping y and β must give yet another optimal code tree.
4. Final opt code tree, x, y are max-depth siblings. ■

Proof continued...

1. y: second least frequent character.
2. β: lowest leaf in tree. Sibling to x.
3. Swapping y and β must give yet another optimal code tree.
4. Final opt code tree, x, y are max-depth siblings. ■

Proof continued...

1. y: second least frequent character.
2. β: lowest leaf in tree. Sibling to x.
3. Swapping y and β must give yet another optimal code tree.
4. Final opt code tree, x, y are max-depth siblings. ■

Proof continued...

1. y: second least frequent character.
2. β: lowest leaf in tree. Sibling to x.
3. Swapping y and β must give yet another optimal code tree.
4. Final opt code tree, x, y are max-depth siblings. ■

Huffman's codes are optimal

Theorem

Huffman codes are optimal prefix-free binary codes.

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\text{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\text{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\text{opt}}$.
8. $\mathcal{T}'_{\text{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\text{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\mathrm{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\mathrm{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\mathrm{opt}}$.
8. $\mathcal{T}'_{\mathrm{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\mathrm{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\text{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\text{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\text{opt}}$.
8. $\mathcal{T}'_{\text{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\text{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\mathrm{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\mathrm{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\mathrm{opt}}$.
8. $\mathcal{T}'_{\mathrm{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\mathrm{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\mathrm{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\mathrm{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\mathrm{opt}}$.
8. $\mathcal{T}'_{\mathrm{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\mathrm{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\text{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\text{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\text{opt}}$.
8. $\mathcal{T}'_{\text{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\text{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\mathrm{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\mathrm{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\mathrm{opt}}$.
8. $\mathcal{T}'_{\mathrm{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\mathrm{opt}}$)

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. $f[1 \ldots n]$ be original input frequencies.
3. Assume $f[1]$ and $f[2]$ are the two smallest.
4. Let $f[n+1] = f[1] + f[2]$.
5. lemma $\implies$ $\exists$ opt. code tree $\mathcal{T}_{\mathrm{opt}}$ for $f[1..n]$
6. $\mathcal{T}_{\mathrm{opt}}$ has 1 and 2 as siblings.
7. Remove 1 and 2 from $\mathcal{T}_{\mathrm{opt}}$.
8. $\mathcal{T}'_{\mathrm{opt}}$: Remaining tree has $3, \ldots, n$ as leafs and "special" character $n+1$ (i.e., parent $1, 2$ in $\mathcal{T}_{\mathrm{opt}}$)

La proof continued...

1. character $n+1$: has frequency $f[n+1]$. Now, $f[n+1] = f[1] + f[2]$, we have

$$\mathrm{cost}(\mathcal{T}_{\mathrm{opt}}) = \sum_{i=1}^{n} f[i] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i)$$

$$= \sum_{i=3}^{n+1} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) + f[1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1) + f[2]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(2) - f[n+1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(n+1)$$

$$= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + \left(f[1] + f[2]\right)\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) - \left(f[1] + f[2]\right)(\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) - 1)$$

$$= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2].$$

La proof continued...

1. character $n+1$: has frequency $f[n+1]$.
   Now, $f[n+1] = f[1] + f[2]$, we have

$$\begin{aligned}
\mathrm{cost}(\mathcal{T}_{\mathrm{opt}}) &= \sum_{i=1}^{n} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) \\
&= \sum_{i=3}^{n+1} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) + f[1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1) \\
&\quad + f[2]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(2) - f[n+1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(n+1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + \left(f[1] + f[2]\right)\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) \\
&\quad - \left(f[1] + f[2]\right)(\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) - 1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2].
\end{aligned}$$

La proof continued...

1. character $n+1$: has frequency $f[n+1]$.
   Now, $f[n+1] = f[1] + f[2]$, we have

$$
\begin{aligned}
\mathrm{cost}(\mathcal{T}_{\mathrm{opt}}) &= \sum_{i=1}^{n} f[i] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) \\
&= \sum_{i=3}^{n+1} f[i] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) + f[1] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1) \\
&\qquad + f[2] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(2) - f[n+1] \mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(n+1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + \left(f[1] + f[2]\right) \mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) \\
&\qquad - \left(f[1] + f[2]\right)(\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) - 1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2].
\end{aligned}
$$

La proof continued...

1. character $n+1$: has frequency $f[n+1]$.
   Now, $f[n+1] = f[1] + f[2]$, we have

$$
\begin{aligned}
\mathrm{cost}(\mathcal{T}_{\mathrm{opt}}) &= \sum_{i=1}^{n} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) \\
&= \sum_{i=3}^{n+1} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) + f[1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1) \\
&\quad + f[2]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(2) - f[n+1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(n+1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + \Big(f[1] + f[2]\Big)\,\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) \\
&\quad - \Big(f[1] + f[2]\Big)(\mathrm{depth}(\mathcal{T}_{\mathrm{opt}}) - 1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2].
\end{aligned}
$$

La proof continued...

1. implies $\mathbf{min}$ cost of $\mathcal{T}_{\mathrm{opt}} \equiv \mathbf{min}$ cost $\mathcal{T}'_{\mathrm{opt}}$.
2. $\mathcal{T}'_{\mathrm{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) = \mathrm{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \mathrm{cost}(\mathsf{T}_H) = \mathrm{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] = \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2] = \mathrm{cost}(\mathcal{T}_{\mathrm{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ∎

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\text{opt}} \equiv \min$ cost $\mathcal{T}'_{\text{opt}}$.
2. $\mathcal{T}'_{\text{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\text{cost}\left(\mathcal{T}'_{\text{opt}}\right) = \text{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \text{cost}(\mathsf{T}_H) = \text{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] =$
   $\text{cost}\left(\mathcal{T}'_{\text{opt}}\right) + f[1] + f[2] = \text{cost}(\mathcal{T}_{\text{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. $\blacksquare$

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\text{opt}} \equiv \min$ cost $\mathcal{T}'_{\text{opt}}$.
2. $\mathcal{T}'_{\text{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\text{cost}\left(\mathcal{T}'_{\text{opt}}\right) = \text{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \text{cost}(\mathsf{T}_H) = \text{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] =$
   $\text{cost}\left(\mathcal{T}'_{\text{opt}}\right) + f[1] + f[2] = \text{cost}(\mathcal{T}_{\text{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ■

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\mathrm{opt}} \equiv \min$ cost $\mathcal{T}'_{\mathrm{opt}}$.
2. $\mathcal{T}'_{\mathrm{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) = \mathrm{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \mathrm{cost}(\mathsf{T}_H) = \mathrm{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] = \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2] = \mathrm{cost}(\mathcal{T}_{\mathrm{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ■

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\mathrm{opt}} \equiv \min$ cost $\mathcal{T}'_{\mathrm{opt}}$.
2. $\mathcal{T}'_{\mathrm{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) = \mathrm{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \mathrm{cost}(\mathsf{T}_H) = \mathrm{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] =$
   $\mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2] = \mathrm{cost}(\mathcal{T}_{\mathrm{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ∎

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\mathrm{opt}} \equiv \min$ cost $\mathcal{T}'_{\mathrm{opt}}$.
2. $\mathcal{T}'_{\mathrm{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) = \mathrm{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \mathrm{cost}(\mathsf{T}_H) = \mathrm{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] = \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2] = \mathrm{cost}(\mathcal{T}_{\mathrm{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ■

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\text{opt}} \equiv \min$ cost $\mathcal{T}'_{\text{opt}}$.
2. $\mathcal{T}'_{\text{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\text{cost}\left(\mathcal{T}'_{\text{opt}}\right) = \text{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \text{cost}(\mathsf{T}_H) = \text{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] =$
   $\text{cost}\left(\mathcal{T}'_{\text{opt}}\right) + f[1] + f[2] = \text{cost}(\mathcal{T}_{\text{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ■

La proof continued...

1. implies $\min$ cost of $\mathcal{T}_{\text{opt}} \equiv \min$ cost $\mathcal{T}'_{\text{opt}}$.
2. $\mathcal{T}'_{\text{opt}}$: must be optimal coding tree for $f[3 \ldots n+1]$.
3. $\mathcal{T}'_H$: Huffman tree for $f[3, \ldots, n+1]$
   T_H: overall Huffman tree constructed for $f[1, \ldots, n]$.
4. By construction:
   $\mathcal{T}'_H$ formed by removing leafs 1 and 2 from T_H.
5. By induction:
   Huffman tree generated for $f[3, \ldots, n+1]$ is optimal.
6. $\mathrm{cost}\left(\mathcal{T}'_{\text{opt}}\right) = \mathrm{cost}\left(\mathcal{T}'_H\right)$.
7. $\implies \mathrm{cost}(\mathsf{T}_H) = \mathrm{cost}\left(\mathcal{T}'_H\right) + f[1] + f[2] = \mathrm{cost}\left(\mathcal{T}'_{\text{opt}}\right) + f[1] + f[2] = \mathrm{cost}(\mathcal{T}_{\text{opt}})$,
8. $\implies$ Huffman tree has the same cost as the optimal tree. ∎

What we get...

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

What we get...

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

What we get...

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

What we get...

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

What we get...

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

What we get...

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

Average size of code word

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

Average size of code word

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

Average size of code word

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

Average size of code word

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

Average size of code word

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

Average length of codewords...

Special case

Lemma

$1, \ldots, n$: *symbols.*
Assume, for $i = 1, \ldots, n$:

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

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

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$. ■

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$. ■

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$. ∎

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$. ■

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$.

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$. ■

Proof

1. 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.
4. Must be $p_i = p_j$ (otherwise, $\sum_k p_k \neq 1$).
5. Huffman's tree merges this two letters, into a single "character" that have probability $2p_i$.
6. New "character" has encoding of length $l_i - 1$, by induction (on remaining $n - 1$ symbols).
7. resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$. ■

Translating lemma...

1. $p_i = 1/2^{l_i}$
2. $l_i = \lg 1/p_i$.
3. Average length of a code word is
$$\sum_i p_i \lg \frac{1}{p_i}.$$
4. 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.

Translating lemma...

1. $p_i = 1/2^{l_i}$
2. $l_i = \lg 1/p_i$.
3. Average length of a code word is
$$\sum_i p_i \lg \frac{1}{p_i}.$$
4. 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.

Translating lemma...

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

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

4. 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.

Translating lemma...

1. $p_i = 1/2^{l_i}$
2. $l_i = \lg 1/p_i$.
3. Average length of a code word is
$$\sum_i p_i \lg \frac{1}{p_i}.$$
4. 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.

Notes

Notes

Notes

Notes