Lossless compression algorithms

Guillaume Tochon

guillaume.tochon@lrde.epita.fr

LRDE, EPITA

Lossless compression

Data before compression and after decompression are strictly identical.

Lossless compression

Data before compression and after decompression are strictly identical.

Lossless compression algorithms exploit data statistical redundancy in two steps :

1) Derive a statistical model for data to compress :
   - → *Static* models : analyze whole data and build model according to it (Huffman, bzip2...).
   - → *Adaptive* models : start with a trivial model and improve it during compression (adapative Huffman, LZW...).

Lossless compression

Data before compression and after decompression are strictly identical.

Lossless compression algorithms exploit data statistical redundancy in two steps :

1) Derive a statistical model for data to compress :
   - → *Static* models : analyze whole data and build model according to it (Huffman, bzip2...).
   - → *Adaptive* models : start with a trivial model and improve it during compression (adapative Huffman, LZW...).
2) Use statistical model to encode input data :

probable symbol	$\Leftrightarrow$	short encoding
improbable symbol	$\Leftrightarrow$	longer encoding

Lossless compression

Data before compression and after decompression are strictly identical.

Lossless compression algorithms exploit data statistical redundancy in two steps :

1) Derive a statistical model for data to compress :
   - → *Static* models : analyze whole data and build model according to it (Huffman, bzip2...).
   - → *Adaptive* models : start with a trivial model and improve it during compression (adapative Huffman, LZW...).
2) Use statistical model to encode input data :

probable symbol	$\Leftrightarrow$	short encoding
improbable symbol	$\Leftrightarrow$	longer encoding

$\Rightarrow$ Super effective on noise-free data (text documents, source codes, synthetic images).

Lossless compression algorithms

RLE compression algorithm

What would be the most naive way to compress the following file F?

$$F = \left\{ \text{AAAAAABBBBCCCCCCC} \right\}$$

RLE compression algorithm

What would be the most naive way to compress the following file F?

$$F = \left\{ \text{AAAAAABBBBCCCCCCC} \right\}$$

$$\Rightarrow F^{\#} = \left\{ \text{6A4B7C} \right\}$$

Run-length encoding

Encode *runs* of symbols (*i.e.* when a symbol is repeated several times in a row) as the number of occurrences in the run followed by the single symbol.

$$\ldots s_j \underbrace{s_i\, s_i \ldots s_i}_{N \text{ times}} s_k \ldots \xrightarrow{\text{RLE}} \ldots s_j\, N\, s_i\, s_k \ldots$$

RLE compression algorithm

Run-length encoding

Encode *runs* of symbols (*i.e.* when a symbol is repeated several times in a row) as the number of occurrences in the run followed by the single symbol.

$$\dots s_j \underbrace{s_i\, s_i \dots s_i}_{N \text{ times}} s_k \dots \stackrel{\text{RLE}}{\Longrightarrow} \dots s_j\, N\, s_i\, s_k \dots$$

Not suited for text compression: `HELLO` becomes `1H1E2L1O` or `HE2LO`.

→ The runs must be sufficiently long.

→ Suitable for synthesis/noise-free images. Used in bitmap (.bmp) format.

RLE compression algorithm

Run-length encoding

Encode *runs* of symbols (*i.e.* when a symbol is repeated several times in a row) as the number of occurrences in the run followed by the single symbol.

$$\dots s_j \underbrace{s_i\, s_i \dots s_i}_{N \text{ times}} s_k \dots \overset{\text{RLE}}{\Longrightarrow} \dots s_j\, N\, s_i\, s_k \dots$$

Not suited for text compression: HELLO becomes 1H1E2L1O or HE2LO.

→ The runs must be sufficiently long.

→ Suitable for synthesis/noise-free images. Used in bitmap (.bmp) format.

Problem when encoding numbers: 22231444457 $\overset{\text{RLE}}{\Longrightarrow}$ 32314457.

→ How to decide whether it is a number of occurrence or a symbol?

RLE compression algorithm

Run-length encoding

Encode *runs* of symbols (*i.e.* when a symbol is repeated several times in a row) as the number of occurrences in the run followed by the single symbol.

$$\dots s_j \underbrace{s_i\, s_i \dots s_i}_{N \text{ times}} s_k \dots \xrightarrow{\text{RLE}} \dots s_j \, \# N \, s_i \, s_k \dots$$

Not suited for text compression: HELLO becomes 1H1E2L1O or HE2LO.

- → The runs must be sufficiently long.
- → Suitable for synthesis/noise-free images. Used in bitmap (.bmp) format.

Problem when encoding numbers: 22231444457 $\xrightarrow{\text{RLE}}$ #3231#4457.

- → How to decide whether it is a number of occurrence or a symbol?
- → Use a special character # to inform the decompression stage.

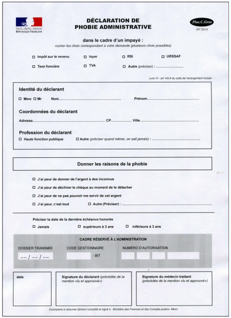
Application to fax transmission of binarized documents

Fax: prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

Application to fax transmission of binarized documents

Fax: prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

The document is scanned line by line by the fax. Each line is binarized and converted into an analog signal (electric impulses) that is sent through telephone lines.



Liberté • Égalité • Fraternité
République Française

DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros
09°2014

dans le cadre d'un impayé :
cocher les choix correspondant à votre demande (plusieurs choix possibles)

- ☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF
- ☐ Taxe foncière ☐ TVA ☐ Autre *(préciser)* : ....

Livre VI - art 145-9 du code de l'arrangement mutuel

Identité du déclarant
☐ Mme ☐ Mr Nom.. Prénom..

Coordonnées du déclarant
Adresse.. CP................ Ville..

Profession du déclarant
☐ Haute fonction publique ☐ Autre *(préciser quand même, on sait jamais)* : ....

Donner les raisons de la phobie

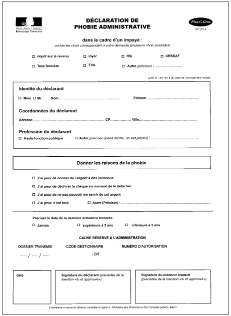
- ☐ J'ai peur de donner de l'argent à des inconnus
- ☐ J'ai peur de déchirer le chèque au moment de le détacher
- ☐ J'ai peur de ne pas pouvoir me servir de cet argent
- ☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée
☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION
DOSSIER TRANSMIS -- / -- / -- CODE GESTIONNAIRE - BIT NUMÉRO D'AUTORISATION

date | Signature du déclarant *(précédée de la mention «lu et approuvé»)* | Signature du médecin traitant *(précédée de la mention «lu et approuvé»)*

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci.

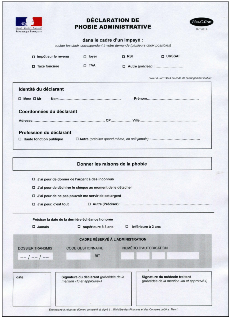


DÉCLARATION DE PHOBIE ADMINISTRATIVE

Application to fax transmission of binarized documents

<u>Fax:</u> prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

The document is scanned line by line by the fax. Each line is binarized and converted into an analog signal (electric impulses) that is sent through telephone lines.



RÉPUBLIQUE FRANÇAISE

DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros
09*2014

dans le cadre d'un impayé :
cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF
☐ Taxe foncière ☐ TVA ☐ Autre *(préciser)* : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant
☐ Mme ☐ Mr Nom.. Prénom..

Coordonnées du déclarant
Adresse.. CP............... Ville..

Profession du déclarant
☐ Haute fonction publique ☐ Autre *(préciser quand même, on sait jamais)* : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus
☐ J'ai peur de déchirer le chèque au moment de le détacher
☐ J'ai peur de ne pas pouvoir me servir de cet argent
☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

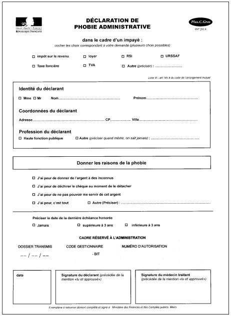
Préciser la date de la dernière échéance honorée
☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION

DOSSIER TRANSMIS	CODE GESTIONNAIRE	NUMÉRO D'AUTORISATION
-- / -- / --	- BIT	

date	Signature du déclarant *(précédée de la mention «lu et approuvé»)*	Signature du médecin traitant *(précédée de la mention «lu et approuvé»)*

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci



RÉPUBLIQUE FRANÇAISE

DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros
09*2014

dans le cadre d'un impayé :
cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF
☐ Taxe foncière ☐ TVA ☐ Autre *(préciser)* : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant
☐ Mme ☐ Mr Nom.. Prénom..

Coordonnées du déclarant
Adresse.. CP............... Ville..

Profession du déclarant
☐ Haute fonction publique ☐ Autre *(préciser quand même, on sait jamais)* : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus
☐ J'ai peur de déchirer le chèque au moment de le détacher
☐ J'ai peur de ne pas pouvoir me servir de cet argent
☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée
☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION

DOSSIER TRANSMIS -- / -- / -- CODE GESTIONNAIRE - BIT NUMÉRO D'AUTORISATION

date | Signature du déclarant *(précédée de la mention «lu et approuvé»)* | Signature du médecin traitant *(précédée de la mention «lu et approuvé»)*

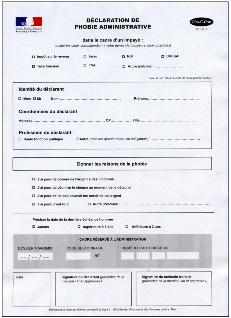
Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci

<u>Rough idea:</u> compress each line independently

Application to fax transmission of binarized documents

Fax: prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

The document is scanned line by line by the fax. Each line is binarized and converted into an analog signal (electric impulses) that is sent through telephone lines.


DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros
09°2014

dans le cadre d'un impayé :
cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF
☐ Taxe foncière ☐ TVA ☐ Autre *(préciser)* : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant
☐ Mme ☐ Mr Nom.................... Prénom....................

Coordonnées du déclarant
Adresse.................... CP.......... Ville....................

Profession du déclarant
☐ Haute fonction publique ☐ Autre *(préciser quand même, on sait jamais)* : ....

Donner les raisons de la phobie

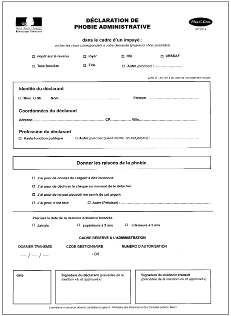
☐ J'ai peur de donner de l'argent à des inconnus
☐ J'ai peur de déchirer le chèque au moment de le détacher
☐ J'ai peur de ne pas pouvoir me servir de cet argent
☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée
☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION
DOSSIER TRANSMIS CODE GESTIONNAIRE NUMÉRO D'AUTORISATION
-- / -- / -- - BIT

date | Signature du déclarant *(précédée de la mention «lu et approuvé»)* | Signature du médecin traitant *(précédée de la mention «lu et approuvé»)*

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci


DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros
09°2014

dans le cadre d'un impayé :
cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF
☐ Taxe foncière ☐ TVA ☐ Autre *(préciser)* : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant
☐ Mme ☐ Mr Nom.................... Prénom....................

Coordonnées du déclarant
Adresse.................... CP.......... Ville....................

Profession du déclarant
☐ Haute fonction publique ☐ Autre *(préciser quand même, on sait jamais)* : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus
☐ J'ai peur de déchirer le chèque au moment de le détacher
☐ J'ai peur de ne pas pouvoir me servir de cet argent
☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée
☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION
DOSSIER TRANSMIS CODE GESTIONNAIRE NUMÉRO D'AUTORISATION
-- / -- / -- - BIT

date | Signature du déclarant *(précédée de la mention «lu et approuvé»)* | Signature du médecin traitant *(précédée de la mention «lu et approuvé»)*

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci

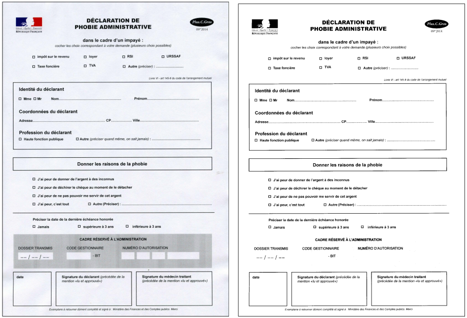
Rough idea: compress each line independently

$\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1} \cdots\cdots\cdots\cdots \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$ $\Rightarrow$ no text, RLE is super effective.

Application to fax transmission of binarized documents

<ins>Fax:</ins> prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

The document is scanned line by line by the fax. Each line is binarized and converted into an analog signal (electric impulses) that is sent through telephone lines.


DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros 09°2014

dans le cadre d'un impayé :

cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF

☐ Taxe foncière ☐ TVA ☐ Autre (préciser) : ....

Livre VI - art 145-9 du code de l'arrangement mutuel

Identité du déclarant

☐ Mme ☐ Mr Nom.................... Prénom....................

Coordonnées du déclarant

Adresse.................... CP.......... Ville....................

Profession du déclarant

☐ Haute fonction publique ☐ Autre (préciser quand même, on sait jamais) : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus

☐ J'ai peur de déchirer le chèque au moment de le détacher

☐ J'ai peur de ne pas pouvoir me servir de cet argent

☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée

☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION

DOSSIER TRANSMIS CODE GESTIONNAIRE NUMÉRO D'AUTORISATION

-- / -- / -- - BIT

date | Signature du déclarant (précédée de la mention «lu et approuvé») | Signature du médecin traitant (précédée de la mention «lu et approuvé»)

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci

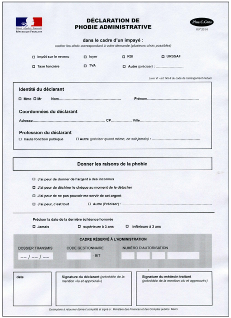
<ins>Rough idea:</ins> compress each line independently

1	1	1	1	1	···························	1	1	1	1	1	⇒ no text, RLE is super effective.

1	1	···	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	···	1	1	⇒ text, but RLE remains effective.

Fax: prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

The document is scanned line by line by the fax. Each line is binarized and converted into an analog signal (electric impulses) that is sent through telephone lines.



DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros 09°2014

dans le cadre d'un impayé :

cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF

☐ Taxe foncière ☐ TVA ☐ Autre (préciser) : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant

☐ Mme ☐ Mr Nom.................... Prénom....................

Coordonnées du déclarant

Adresse.................... CP.......... Ville....................

Profession du déclarant

☐ Haute fonction publique ☐ Autre (préciser quand même, on sait jamais) : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus

☐ J'ai peur de déchirer le chèque au moment de le détacher

☐ J'ai peur de ne pas pouvoir me servir de cet argent

☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée

☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

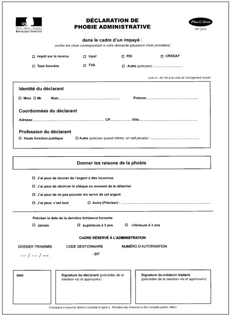
CADRE RÉSERVÉ À L'ADMINISTRATION

DOSSIER TRANSMIS CODE GESTIONNAIRE NUMÉRO D'AUTORISATION

-- / -- / -- - BIT

date | Signature du déclarant (précédée de la mention «lu et approuvé») | Signature du médecin traitant (précédée de la mention «lu et approuvé»)

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci



DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros 09°2014

dans le cadre d'un impayé :

cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF

☐ Taxe foncière ☐ TVA ☐ Autre (préciser) : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant

☐ Mme ☐ Mr Nom.................... Prénom....................

Coordonnées du déclarant

Adresse.................... CP.......... Ville....................

Profession du déclarant

☐ Haute fonction publique ☐ Autre (préciser quand même, on sait jamais) : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus

☐ J'ai peur de déchirer le chèque au moment de le détacher

☐ J'ai peur de ne pas pouvoir me servir de cet argent

☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée

☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

CADRE RÉSERVÉ À L'ADMINISTRATION

DOSSIER TRANSMIS CODE GESTIONNAIRE NUMÉRO D'AUTORISATION

-- / -- / -- - BIT

date | Signature du déclarant (précédée de la mention «lu et approuvé») | Signature du médecin traitant (précédée de la mention «lu et approuvé»)

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci

Smarter idea: compress difference between consecutive lines

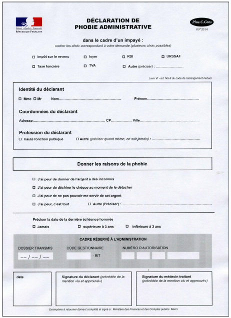
1	1	···	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	···	1	1
1	1	···	1	1	0	0	1	1	1	0	0	1	1	1	0	0	1	···	1	1

$\Rightarrow$ Very few differences between two lines.

Application to fax transmission of binarized documents

Fax: prehistoric machine that used to send documents by wire transmission (dated circa 1500 BC).

The document is scanned line by line by the fax. Each line is binarized and converted into an analog signal (electric impulses) that is sent through telephone lines.



RÉPUBLIQUE FRANÇAISE

DÉCLARATION DE PHOBIE ADMINISTRATIVE

Plus.C.Gros 09°2014

dans le cadre d'un impayé :

cocher les choix correspondant à votre demande (plusieurs choix possibles)

☐ impôt sur le revenu ☐ loyer ☐ RSI ☐ URSSAF

☐ Taxe foncière ☐ TVA ☐ Autre *(préciser)* : ....

Livre VI - art 145-8 du code de l'arrangement mutuel

Identité du déclarant

☐ Mme ☐ Mr Nom........................ Prénom........................

Coordonnées du déclarant

Adresse........................ CP.............. Ville........................

Profession du déclarant

☐ Haute fonction publique ☐ Autre *(préciser quand même, on sait jamais)* : ....

Donner les raisons de la phobie

☐ J'ai peur de donner de l'argent à des inconnus

☐ J'ai peur de déchirer le chèque au moment de le détacher

☐ J'ai peur de ne pas pouvoir me servir de cet argent

☐ J'ai peur, c'est tout ☐ Autre (Préciser) : ....

Préciser la date de la dernière échéance honorée

☐ Jamais ☐ supérieure à 3 ans ☐ inférieure à 3 ans

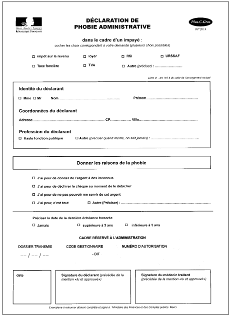
CADRE RÉSERVÉ À L'ADMINISTRATION

DOSSIER TRANSMIS CODE GESTIONNAIRE NUMÉRO D'AUTORISATION

-- / -- / -- - BIT

date | Signature du déclarant *(précédée de la mention «lu et approuvé»)* | Signature du médecin traitant *(précédée de la mention «lu et approuvé»)*

Exemplaire à retourner dûment complété et signé à : Ministère des Finances et des Comptes publics. Merci



Smarter idea: compress difference between consecutive lines

l_i	1	1	⋯	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	⋯	1	1
l_{i+1}	1	1	⋯	1	1	0	0	1	1	1	0	0	1	1	1	0	0	1	⋯	1	1
$l_i \bigoplus l_{i+1}$	0	0	⋯	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	⋯	0	0

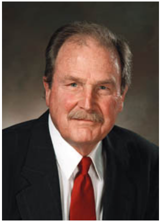
$\Rightarrow$ Compress the line difference instead, RLE is super effective.

Huffman compression algorithm

→ Proposed in 1952 by David Huffman (during its Ph.D).

→ Exploit the statistical distribution of the symbols to encode.
⇒ *entropy encoding*.

→ Frequent symbols are given shorter encoding support.
⇒ *variable-length code*.



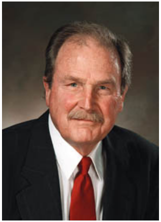
David Huffman

Huffman compression algorithm

→ Proposed in 1952 by David Huffman (during its Ph.D).

→ Exploit the statistical distribution of the symbols to encode.
$\Rightarrow$ *entropy encoding*.

→ Frequent symbols are given shorter encoding support.
$\Rightarrow$ *variable-length code*.



David Huffman

Huffman encoding

The encoding is obtained by the following two-steps procedure:

1) Compute a binary tree whose leaves are the symbols, by iteratively merging the two symbols with lowest probabilities of occurence.
2) Label each left branch by 0 and each right branch by 1 (or the other way around).

The encoding of each symbol is given by the path from the root to the leaf corresponding to the symbol.

Example (1/3)

Computing a binary tree

Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.

Example (1/3)

Computing a binary tree

Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
♻	0.1
☯	0.1
⛔	0.06
♪	0.04

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.

Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
♻	0.1
☯	0.1
⛔	0.06
♪	0.04

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.

⛔
0.06

♪
0.04

Example (1/3)

Computing a binary tree

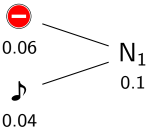
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
♻	0.1
☯	0.1
N_1	0.1

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

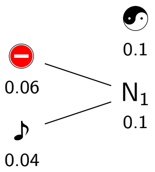
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
♻	0.1
☯	0.1
N_1	0.1

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



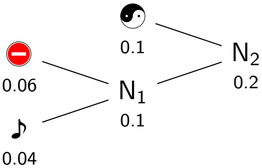
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
🍀	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
🍀	0.3
♻	0.1
N_2	0.2

Huffman encoding: 1^st^ step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

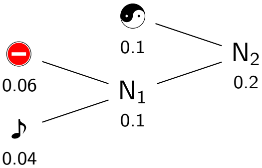
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
🍀	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
🍀	0.3
N_2	0.2
♻	0.1

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

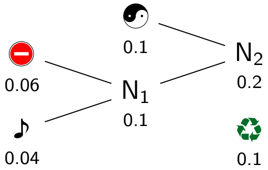
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
N_2	0.2
♻	0.1

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

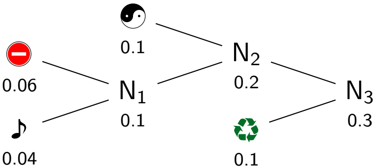
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
N_3	0.3

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

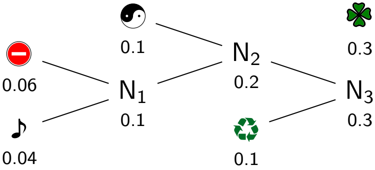
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
☘	0.3
N_3	0.3

Huffman encoding: 1st step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

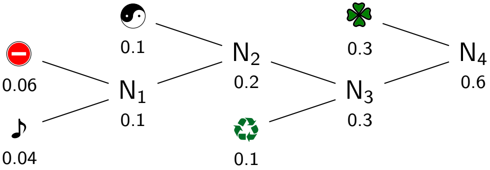
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
☢	0.4
N_4	0.6

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

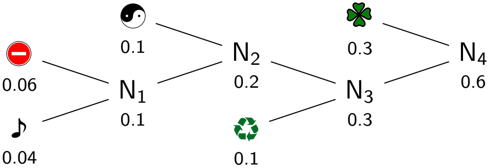
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
N_4	0.6
☢	0.4

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

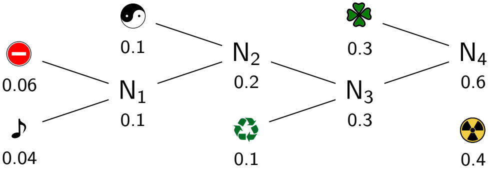
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
N_4	0.6
☢	0.4

Huffman encoding: 1^{st} step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (1/3)

Computing a binary tree

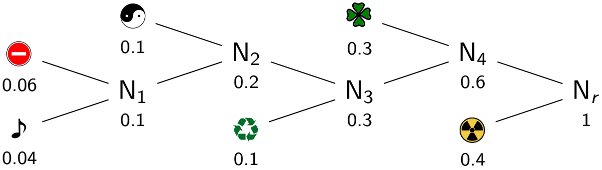
Take the following alphabet Σ with known probability distribution:

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

s_i	$\mathbb{p}_i$
N_r	1

Huffman encoding: 1st step

1) Sort table (if necessary).
2) Merge the two symbols with lowest probability of occurence.
3) Update table.
4) Repeat if more than one symbol remaining.



Example (2/3)

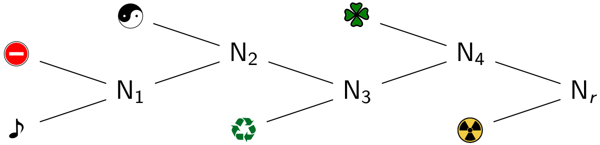
Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

Encoding

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.

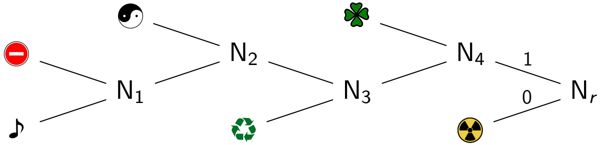


s_i	$\mathbb{p}_i$
☘	0.3
⛔	0.06
♻	0.1
☯	0.1
☢	0.4
♪	0.04

Encoding

Huffman encoding: 2^nd step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



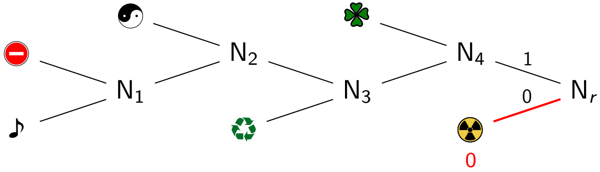
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	
⛔	0.06	
♻	0.1	
☯	0.1	
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



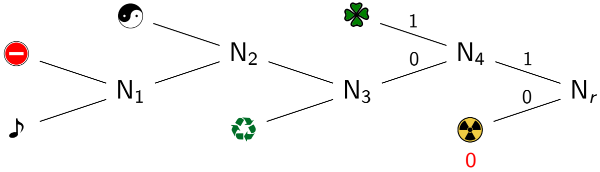
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	
⛔	0.06	
♻	0.1	
☯	0.1	
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



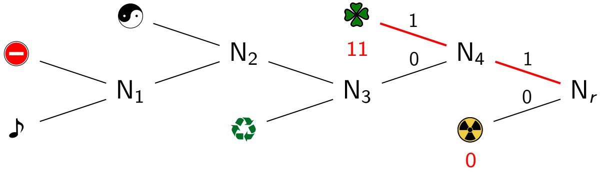
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	
♻	0.1	
☯	0.1	
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



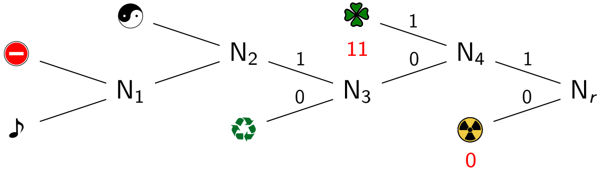
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	
♻	0.1	
☯	0.1	
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



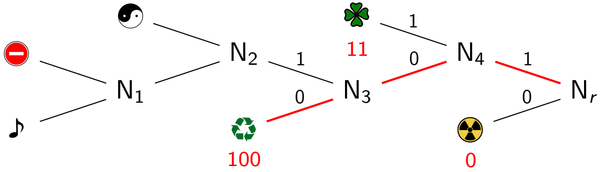
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	
♻	0.1	100
☯	0.1	
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

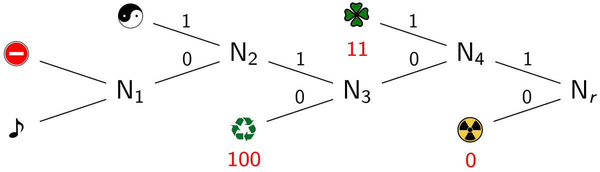
1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	
♻	0.1	100
☯	0.1	
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



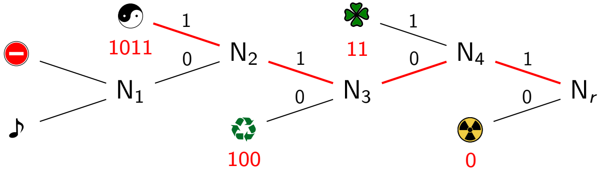
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	
♻	0.1	100
☯	0.1	1011
☢	0.4	0
♪	0.04	

Huffman encoding: 2nd step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



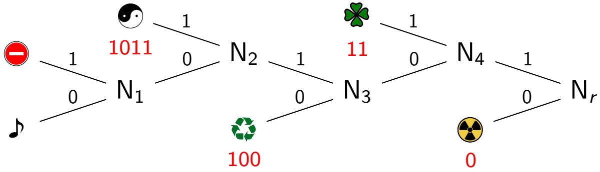
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	
♻	0.1	100
☯	0.1	1011
☢	0.4	0
♪	0.04	

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



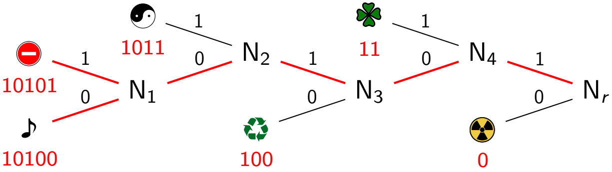
Example (2/3)

Retrieving the encoding for each symbol

s_i	$\mathbb{p}_i$	Encoding
☘	0.3	11
⛔	0.06	10101
♻	0.1	100
☯	0.1	1011
☢	0.4	0
♪	0.04	10100

Huffman encoding: 2^{nd} step

1) Traverse the tree from the root to the leaves (the symbols).
2) Always assign 1 to the left child and 0 to the right one (or the other way around).
3) Read encoding on the whole branch.



Example (3/3)

Performances of the encoding

The entropy of $\Sigma = \{$🍀, ⛔, ♻, ☯, ☢, ♪$\}$ with previously given probability distribution is $H \simeq 2.144\ Sh/symb$.

$\Rightarrow$ A theoretically optimal encoding needs an average length of 2.144 bits per symbol.

Example (3/3)

Performances of the encoding

The entropy of $\Sigma = \{$🍀, ⛔, ♻, ☯, ☢, ♪$\}$ with previously given probability distribution is $H \simeq 2.144$ $Sh/symb$.

$\Rightarrow$ A theoretically optimal encoding needs an average length of 2.144 bits per symbol.

The average length of Huffman encoding is

$$\boxed{L = \mathbb{E}[\ell(s_i)] = \sum_{s_i \in \Sigma} \mathrm{p}_i \ell(s_i)} \quad bits/symb.$$

where $\ell(s_i)$ is the encoding length of symbol s_i.

$\Rightarrow L = 0.3 \times 2 + 0.06 \times 5 + 0.1 \times 3 + 0.1 \times 4 + 0.4 \times 1 + 0.04 \times 5 = 2.2$ $bits/symb$.

Example (3/3)

Performances of the encoding

The entropy of $\Sigma = \{$☘, ⛔, ♻, ☯, ☢, ♪$\}$ with previously given probability distribution is $H \simeq 2.144$ $Sh/symb$.

$\Rightarrow$ A theoretically optimal encoding needs an average length of 2.144 bits per symbol.

The average length of Huffman encoding is

$$\boxed{L = \mathbb{E}[\ell(s_i)] = \sum_{s_i \in \Sigma} \mathrm{p}_i \ell(s_i)} \; bits/symb.$$

where $\ell(s_i)$ is the encoding length of symbol s_i.

$\Rightarrow L = 0.3 \times 2 + 0.06 \times 5 + 0.1 \times 3 + 0.1 \times 4 + 0.4 \times 1 + 0.04 \times 5 = 2.2$ $bits/symb$.

Take some file F with $N_F = 1000$ symbols drawn from Σ.

$\rightarrow$ 3000 bits are necessary to encode F without compression.

Example (3/3)

Performances of the encoding

The entropy of $\Sigma = \{$☘,⛔,♻,☯,☢, ♪$\}$ with previously given probability distribution is $H \simeq 2.144\ Sh/symb$.
$\Rightarrow$ A theoretically optimal encoding needs an average length of 2.144 bits per symbol.

The average length of Huffman encoding is

$$\boxed{L = \mathbb{E}[\ell(s_i)] = \sum_{s_i \in \Sigma} \mathrm{p}_i \ell(s_i)}\ \ bits/symb.$$

where $\ell(s_i)$ is the encoding length of symbol s_i.
$\Rightarrow L = 0.3 \times 2 + 0.06 \times 5 + 0.1 \times 3 + 0.1 \times 4 + 0.4 \times 1 + 0.04 \times 5 = 2.2\ bits/symb.$

Take some file F with $N_F = 1000$ symbols drawn from Σ.

- $\rightarrow$ 3000 bits are necessary to encode F without compression.
- $\rightarrow$ The minimal compressed size of F is $H \times N_F = 2144$ bits.

Example (3/3)

Performances of the encoding

The entropy of $\Sigma = \{$☘, ⛔, ♻, ☯, ☢, ♪$\}$ with previously given probability distribution is $H \simeq 2.144\ Sh/symb$.
$\Rightarrow$ A theoretically optimal encoding needs an average length of 2.144 bits per symbol.

The average length of Huffman encoding is

$$\boxed{L = \mathbb{E}[\ell(s_i)] = \sum_{s_i \in \Sigma} \mathrm{p}_i \ell(s_i)}\ bits/symb.$$

where $\ell(s_i)$ is the encoding length of symbol s_i.
$\Rightarrow L = 0.3 \times 2 + 0.06 \times 5 + 0.1 \times 3 + 0.1 \times 4 + 0.4 \times 1 + 0.04 \times 5 = 2.2\ bits/symb.$

Take some file F with $N_F = 1000$ symbols drawn from Σ.

- $\rightarrow$ 3000 bits are necessary to encode F without compression.
- $\rightarrow$ The minimal compressed size of F is $H \times N_F = 2144$ bits.
- $\rightarrow$ Huffman encoding allows to reach $L \times N_F = 2200$ bits (in average).

Example (3/3)

Performances of the encoding

The entropy of $\Sigma = \{$☘,⛔,♻,☯,☢,♪$\}$ with previously given probability distribution is $H \simeq 2.144\ Sh/symb$.
$\Rightarrow$ A theoretically optimal encoding needs an average length of 2.144 bits per symbol.

The average length of Huffman encoding is

$$\boxed{L = \mathbb{E}[\ell(s_i)] = \sum_{s_i \in \Sigma} \mathrm{p}_i \ell(s_i)}\ bits/symb.$$

where $\ell(s_i)$ is the encoding length of symbol s_i.
$\Rightarrow L = 0.3 \times 2 + 0.06 \times 5 + 0.1 \times 3 + 0.1 \times 4 + 0.4 \times 1 + 0.04 \times 5 = 2.2\ bits/symb.$

Take some file F with $N_F = 1000$ symbols drawn from Σ.

- $\rightarrow$ 3000 bits are necessary to encode F without compression.
- $\rightarrow$ The minimal compressed size of F is $H \times N_F = 2144$ bits.
- $\rightarrow$ Huffman encoding allows to reach $L \times N_F = 2200$ bits (in average).

Properties of Huffman encoding

Huffman encoding is a *prefix code* (no symbol encoding is a prefix of another symbol encoding).
$\Rightarrow$ it can be decoded on the fly.

Ex: 11001010100 $\Rightarrow$ |11|0|0|10101|100| $\Rightarrow$ 

Properties of Huffman encoding

Huffman encoding is a *prefix code* (no symbol encoding is a prefix of another symbol encoding).
$\Rightarrow$ it can be decoded on the fly.

Ex: 11001010100 $\Rightarrow$ |11|0|0|10101|100| $\Rightarrow$ 

Huffman encoding is *optimal* in terms of average encoding length with respect to any symbol-by-symbol encoding with prefix property.

That is, if $\mathfrak{L}$ is another encoding strategy at the symbol level with prefix property, then $L_{Huffman} \leq L_{\mathfrak{L}}$.

Properties of Huffman encoding

Huffman encoding is a *prefix code* (no symbol encoding is a prefix of another symbol encoding).
$\Rightarrow$ it can be decoded on the fly.

Ex: 11001010100 $\Rightarrow$ |11|0|0|10101|100| $\Rightarrow$ 

Huffman encoding is *optimal* in terms of average encoding length with respect to any symbol-by-symbol encoding with prefix property.

That is, if $\mathfrak{L}$ is another encoding strategy at the symbol level with prefix property, then $L_{Huffman} \leq L_{\mathfrak{L}}$.

$H = L_{Huffman}$ if the probabilities of symbols are natural powers of $1/2$, otherwise $H < L_{Huffman} < H + 1$.

Huffman encoding reaches the entropy H if $\forall s_i \in \Sigma, \mathbb{p}_i = (1/2)^k, k \in \mathbb{N}$. Otherwise, Huffman encoding is "close" to the entropy.

Limits of Huffman encoding

Huffman encoding strategy suffers from two mains drawbacks:

✗ It is a symbol-by-symbol encoding → does not handle *words* (as sequences of symbols).

Ex: In a piece of code, words such as `for`, `while`, `end`, are likely to frequently appear, and should directly be encoded at the word scale.

Limits of Huffman encoding

Huffman encoding strategy suffers from two mains drawbacks:

- ✗ It is a symbol-by-symbol encoding $\rightarrow$ does not handle *words* (as sequences of symbols).

  Ex: In a piece of code, words such as `for`, `while`, `end`, are likely to frequently appear, and should directly be encoded at the word scale.

- ✗ It relies (too) strongly on the probability distribution of the symbols to encode.
  - $\rightarrow$ The file to compress must be scanned first to estimate the probability distribution.
  - $\rightarrow$ The frequency table (or the Huffman tree) must be stored with the text for the decoding stage.

Limits of Huffman encoding

Huffman encoding strategy suffers from two mains drawbacks:

✗ It is a symbol-by-symbol encoding → does not handle *words* (as sequences of symbols).

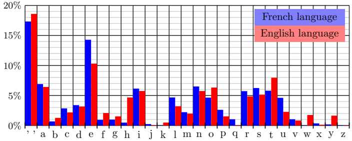
Ex: In a piece of code, words such as `for`, `while`, `end`, are likely to frequently appear, and should directly be encoded at the word scale.

✗ It relies (too) strongly on the probability distribution of the symbols to encode.

→ The file to compress must be scanned first to estimate the probability distribution.

→ The frequency table (or the Huffman tree) must be stored with the text for the decoding stage.

↪ Classical text statistics can be used (language-dependent).



Limits of Huffman encoding

Huffman encoding strategy suffers from two mains drawbacks:

✗ It is a symbol-by-symbol encoding → does not handle *words* (as sequences of symbols).

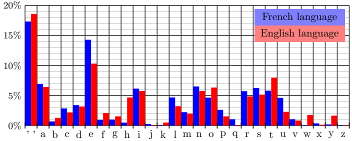
Ex: In a piece of code, words such as `for`, `while`, `end`, are likely to frequently appear, and should directly be encoded at the word scale.

✗ It relies (too) strongly on the probability distribution of the symbols to encode.

→ The file to compress must be scanned first to estimate the probability distribution.

→ The frequency table (or the Huffman tree) must be stored with the text for the decoding stage.

↪ Classical text statistics can be used (language-dependent).



→ And what if the frequencies vary with time?

Limits of Huffman encoding

Huffman encoding strategy suffers from two mains drawbacks:

✗ It is a symbol-by-symbol encoding → does not handle *words* (as sequences of symbols).

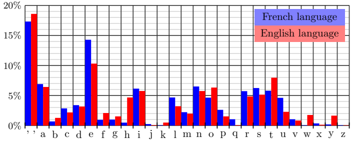
Ex: In a piece of code, words such as `for`, `while`, `end`, are likely to frequently appear, and should directly be encoded at the word scale.

✗ It relies (too) strongly on the probability distribution of the symbols to encode.

→ The file to compress must be scanned first to estimate the probability distribution.

→ The frequency table (or the Huffman tree) must be stored with the text for the decoding stage.

↪ Classical text statistics can be used (language-dependent).



→ And what if the frequencies vary with time?

↪ Adaptive Huffman algorithm.

Limits of Huffman encoding

Huffman encoding strategy suffers from two mains drawbacks:

✗ It is a symbol-by-symbol encoding → does not handle *words* (as sequences of symbols).

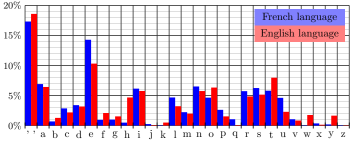
Ex: In a piece of code, words such as `for`, `while`, `end`, are likely to frequently appear, and should directly be encoded at the word scale.

✗ It relies (too) strongly on the probability distribution of the symbols to encode.

→ The file to compress must be scanned first to estimate the probability distribution.

→ The frequency table (or the Huffman tree) must be stored with the text for the decoding stage.

↪ Classical text statistics can be used (language-dependent).



→ And what if the frequencies vary with time?

↪ Adaptive Huffman algorithm.

Huffman encoding alone is nowadays hardly used for text compression, but serves as the last level of more advanced compression algorithms (bzip2, JPEG, etc).

→ Created between 1996 and 2000 by Julian Seward.

→ Initially proposed to tackle patenty issues with LZW compression algorithm.

→ Free and open-source compression algorithm.

Julian Seward

bzip2 encoding

Idea: strenghten the anisotropy of symbol probabilities.

1) Apply Burrows-Wheeler transform to convert frequently-recurring symbol sequences into runs of identical symbols.
2) Use Move-to-front (MTF) transform to replace each symbol by its index in a dynamic table.
3) Encode index sequence with Huffman algorithm.

Burrows-Wheeler transform

The encoding part

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Burrows-Wheeler transform

The encoding part

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ $\equiv$ EOF)

1st step: Construct a table containing all rotations of the string to transform.

b a n a n a $

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.

b	a	n	a	n	a	$
$	b	a	n	a	n	a

Burrows-Wheeler transform

The encoding part

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n

Burrows-Wheeler transform

The encoding part

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

Burrows-Wheeler transform

The encoding part

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.
2nd step: Sort rows into lexicographic order.

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.
2nd step: Sort rows into lexicographic order.

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

$\xrightarrow[\text{sort}]{\text{lexicographic}}$

$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.
2nd step: Sort rows into lexicographic order.
BWT: row number of initial string + last column of sorted table.

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

$\xrightarrow[\text{sort}]{\text{lexicographic}}$

1	$	b	a	n	a	n	a
2	a	$	b	a	n	a	n
3	a	n	a	$	b	a	n
4	a	n	a	n	a	$	b
5	b	a	n	a	n	a	$
6	n	a	$	b	a	n	a
7	n	a	n	a	$	b	a

Burrows-Wheeler transform

The encoding part

Invented by Michael Burrows and David Wheeler in 1994 to sort strings of characters into runs of similar characters to ease their compression, in a totally reversible way.

Ex: Consider the file banana$ ($ ≡ EOF)

1st step: Construct a table containing all rotations of the string to transform.
2nd step: Sort rows into lexicographic order.
BWT: row number of initial string + last column of sorted table.

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

$\xrightarrow[\text{sort}]{\text{lexicographic}}$

1	$	b	a	n	a	n	a
2	a	$	b	a	n	a	n
3	a	n	a	$	b	a	n
4	a	n	a	n	a	$	b
5	b	a	n	a	n	a	$
6	n	a	$	b	a	n	a
7	n	a	n	a	$	b	a

$\Rightarrow$ BWT(banana$) = 5annb$aa

BWT (How much wood would a woodchuck chuck if a woodchuck could chuck wood?$) = dfkdkkwhkaad?d$ udduuuu llooooiccccc ccccuu oooowwwwwcHmhhhhooo

And that sure looks way better. How much better?

Burrows-Wheeler transform

Not convinced?

BWT(How much wood would a woodchuck chuck if a woodchuck could chuck wood?$) = dfkdkkwhkaad?d$ udd uuuu llooooiccccc ccc-cuu oooowwwwwcHmh-hhhooo

And that sure looks way better. How much better?

RLE(dfkdkkwhkaad?d$ udd uuuu llooooiccccc ccc-cuu oooowwwwwcHmh-hhhooo) = dfkd#2kwhk#2ad?d$ u#2d #4u #2l#4oi #5c #4c#2u #4o#5 wcHm#4h#3o

BWT (How much wood would a woodchuck chuck if a woodchuck could chuck wood?$) = dfkdkkwhkaad?d$ udd uuuu lloooocccccc ccc-cuu oooowwwwwcHmh-hhhooo

And that sure looks way better. How much better?

RLE (dfkdkkwhkaad?d$ udd uuuu lloooocccccc ccc-cuu oooowwwwwcHmh-hhhooo) = dfkd#2kwhk#2ad?d$ u#2d #4u #2l#4oi #5c #4c#2u #4o#5 wcHm#4h#3o

Transforming 65 characters into ... 61 characters! 👍👏👌🤘

Burrows-Wheeler transform

Not convinced?

BWT (How much wood would a woodchuck chuck if a woodchuck could chuck wood?$) = dfkdkkwhkaad?d$ udd uuuu llooooiccccc ccccuu oooowwwwwcHmhhhhooo

And that sure looks way better. How much better?

RLE (dfkdkkwhkaad?d$ udd uuuu llooooiccccc ccccuu oooowwwwwcHmhhhhooo) = dfkd#2kwhk#2ad?d$ u#2d #4u #2l#4oi #5c #4c#2u #4o#5 wcHm#4h#3o

Transforming 65 characters into . . . 61 characters! 👍👏👌🤘

BWT is applied in practice on much longer strings thanks to some efficient and optimized implementations ⇒ initial "pre-processing" for the MTF transform.

Burrows-Wheeler transform

The decoding part

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.

a
n
n
b
$
a
a

Burrows-Wheeler transform

The decoding part

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.

$
a
a
a
b
n
n

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

a	$
n	a
n	a
b	a
$	b
a	n
a	n

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

$	b
a	$
a	n
a	n
b	a
n	a
n	a

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

a	$	b
n	a	$
n	a	n
b	a	n
$	b	a
a	n	a
a	n	a

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
$\Rightarrow$ Repeat until the whole table is recreated.

$	b	a
a	$	b
a	n	a
a	n	a
b	a	n
n	a	$
n	a	n

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

a	$	b	a
n	a	$	b
n	a	n	a
b	a	n	a
$	b	a	n
a	n	a	$
a	n	a	n

Burrows-Wheeler transform

The decoding part

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
$\Rightarrow$ Repeat until the whole table is recreated.

$	b	a	n
a	$	b	a
a	n	a	$
a	n	a	n
b	a	n	a
n	a	$	b
n	a	n	a

Burrows-Wheeler transform

The decoding part

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

a	$	b	a	n
n	a	$	b	a
n	a	n	a	$
b	a	n	a	n
$	b	a	n	a
a	n	a	$	b
a	n	a	n	a

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
$\Rightarrow$ Repeat until the whole table is recreated.

$	b	a	n	a
a	$	b	a	n
a	n	a	$	b
a	n	a	n	a
b	a	n	a	n
n	a	$	b	a
n	a	n	a	$

Burrows-Wheeler transform

The decoding part

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

a	$	b	a	n	a
n	a	$	b	a	n
n	a	n	a	$	b
b	a	n	a	n	a
$	b	a	n	a	n
a	n	a	$	b	a
a	n	a	n	a	$

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1[st] step: Concatenate annb$aa before the last column of the table.
2[nd] step: Sort the rows in lexicographic order.
$\Rightarrow$ Repeat until the whole table is recreated.

$	b	a	n	a	n
a	$	b	a	n	a
a	n	a	$	b	a
a	n	a	n	a	$
b	a	n	a	n	a
n	a	$	b	a	n
n	a	n	a	$	b

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

a	$	b	a	n	a	n
n	a	$	b	a	n	a
n	a	n	a	$	b	a
b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	n	a	$	b	a	n
a	n	a	n	a	$	b

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.

$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
⇒ Repeat until the whole table is recreated.
BWT^{-1}: Read the corresponding row in the table.

1	$	b	a	n	a	n	a
2	a	$	b	a	n	a	n
3	a	n	a	$	b	a	n
4	a	n	a	n	a	$	b
5	b	a	n	a	n	a	$
6	n	a	$	b	a	n	a
7	n	a	n	a	$	b	a

It is easily possible to recover banana$ from 5annb$aa by reconstructing the sorted table that was constructed during the computation of the BWT.

1st step: Concatenate annb$aa before the last column of the table.
2nd step: Sort the rows in lexicographic order.
$\Rightarrow$ Repeat until the whole table is recreated.
BWT^{-1}: Read the corresponding row in the table.

1	$	b	a	n	a	n	a
2	a	$	b	a	n	a	n
3	a	n	a	$	b	a	n
4	a	n	a	n	a	$	b
5	b	a	n	a	n	a	$
6	n	a	$	b	a	n	a
7	n	a	n	a	$	b	a

$\Rightarrow BWT^{-1}\left(\text{5annb\$aa}\right) = \text{banana\$}$

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana\$) = 5annb\$aa ⇒ let's encode annb\$aa $\mathcal{L}_0 =$ [a,b,n,\$]

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT$\left(\boxed{\text{banana\$}}\right) = \boxed{\text{5annb\$aa}}$ $\Rightarrow$ let's encode $\boxed{\text{annb\$aa}}$ $\quad \mathcal{L}_0 = [\text{a,b,n,\$}]$

$\boxed{\text{a}} \rightarrow 0$ first symbol of $\mathcal{L}_0$

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana$) = 5annb$aa ⇒ let's encode annb$aa $\mathcal{L}_0 =$ [a,b,n,$]

a $\rightarrow 0$ first symbol of $\mathcal{L}_0$

n $\rightarrow 2$ third symbol of $\mathcal{L}_0$ $\mathcal{L}_0 \rightarrow \mathcal{L}_1 =$ [n,a,b,$]

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana\$) = 5annb\$aa ⇒ let's encode annb\$aa $\mathcal{L}_0 = [\mathrm{a},\mathrm{b},\mathrm{n},\$]$

a $\rightarrow 0$ first symbol of $\mathcal{L}_0$

n $\rightarrow 2$ third symbol of $\mathcal{L}_0$ $\mathcal{L}_0 \rightarrow \mathcal{L}_1 = [\mathrm{n},\mathrm{a},\mathrm{b},\$]$

n $\rightarrow 0$ first symbol of $\mathcal{L}_1$

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana\$) = 5annb\$aa ⇒ let's encode annb\$aa $\mathcal{L}_0 = [\text{a,b,n,\$}]$

a $\rightarrow 0$	first symbol of $\mathcal{L}_0$		
n $\rightarrow 2$	third symbol of $\mathcal{L}_0$	$\mathcal{L}_0 \rightarrow \mathcal{L}_1 = [\text{n,a,b,\$}]$	
n $\rightarrow 0$	first symbol of $\mathcal{L}_1$		
b $\rightarrow 2$	third symbol of $\mathcal{L}_1$	$\mathcal{L}_1 \rightarrow \mathcal{L}_2 = [\text{b,n,a,\$}]$	

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana\$) = 5annb\$aa ⇒ let's encode annb\$aa $\mathcal{L}_0 = [a,b,n,\$]$

a	$\rightarrow 0$	first symbol of $\mathcal{L}_0$	
n	$\rightarrow 2$	third symbol of $\mathcal{L}_0$	$\mathcal{L}_0 \rightarrow \mathcal{L}_1 = [n,a,b,\$]$
n	$\rightarrow 0$	first symbol of $\mathcal{L}_1$	
b	$\rightarrow 2$	third symbol of $\mathcal{L}_1$	$\mathcal{L}_1 \rightarrow \mathcal{L}_2 = [b,n,a,\$]$
\$	$\rightarrow 3$	fourth symbol of $\mathcal{L}_2$	$\mathcal{L}_2 \rightarrow \mathcal{L}_3 = [\$,b,n,a]$

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana\$) = 5annb\$aa ⇒ let's encode annb\$aa $\mathcal{L}_0 = [a,b,n,\$]$

a	$\rightarrow 0$	first symbol of $\mathcal{L}_0$	
n	$\rightarrow 2$	third symbol of $\mathcal{L}_0$	$\mathcal{L}_0 \rightarrow \mathcal{L}_1 = [n,a,b,\$]$
n	$\rightarrow 0$	first symbol of $\mathcal{L}_1$	
b	$\rightarrow 2$	third symbol of $\mathcal{L}_1$	$\mathcal{L}_1 \rightarrow \mathcal{L}_2 = [b,n,a,\$]$
\$	$\rightarrow 3$	fourth symbol of $\mathcal{L}_2$	$\mathcal{L}_2 \rightarrow \mathcal{L}_3 = [\$,b,n,a]$
a	$\rightarrow 3$	fourth symbol of $\mathcal{L}_3$	$\mathcal{L}_3 \rightarrow \mathcal{L}_4 = [a,\$,b,n]$

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana\$) = 5annb\$aa ⇒ let's encode annb\$aa $\mathcal{L}_0 = [a,b,n,\$]$

a	$\rightarrow 0$	first symbol of $\mathcal{L}_0$	
n	$\rightarrow 2$	third symbol of $\mathcal{L}_0$	$\mathcal{L}_0 \rightarrow \mathcal{L}_1 = [n,a,b,\$]$
n	$\rightarrow 0$	first symbol of $\mathcal{L}_1$	
b	$\rightarrow 2$	third symbol of $\mathcal{L}_1$	$\mathcal{L}_1 \rightarrow \mathcal{L}_2 = [b,n,a,\$]$
\$	$\rightarrow 3$	fourth symbol of $\mathcal{L}_2$	$\mathcal{L}_2 \rightarrow \mathcal{L}_3 = [\$,b,n,a]$
a	$\rightarrow 3$	fourth symbol of $\mathcal{L}_3$	$\mathcal{L}_3 \rightarrow \mathcal{L}_4 = [a,\$,b,n]$
a	$\rightarrow 0$	first symbol of $\mathcal{L}_4$	

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana$) = 5annb$aa ⇒ let's encode annb$aa $\mathcal{L}_0 =$ [a,b,n,$]

a	$\rightarrow 0$	first symbol of $\mathcal{L}_0$	
n	$\rightarrow 2$	third symbol of $\mathcal{L}_0$	$\mathcal{L}_0 \rightarrow \mathcal{L}_1 =$ [n,a,b,$]
n	$\rightarrow 0$	first symbol of $\mathcal{L}_1$	
b	$\rightarrow 2$	third symbol of $\mathcal{L}_1$	$\mathcal{L}_1 \rightarrow \mathcal{L}_2 =$ [b,n,a,$]
$	$\rightarrow 3$	fourth symbol of $\mathcal{L}_2$	$\mathcal{L}_2 \rightarrow \mathcal{L}_3 =$ [$,b,n,a]
a	$\rightarrow 3$	fourth symbol of $\mathcal{L}_3$	$\mathcal{L}_3 \rightarrow \mathcal{L}_4 =$ [a,$,b,n]
a	$\rightarrow 0$	first symbol of $\mathcal{L}_4$	

MTF(annb$aa) ⇒ 0202330

0 becomes much more probable than other digits → Huffman likes that!

Move-to-front transform

Idea: encode a flow of symbols by their index in a list. The list is dynamically updated so that most recently seen symbol is moved to the front of the list.

Ex: BWT(banana$) = 5annb$aa ⇒ let's encode annb$aa $\mathcal{L}_0 = [a,b,n,\$]$

a	$\rightarrow 0$	first symbol of $\mathcal{L}_0$	
n	$\rightarrow 2$	third symbol of $\mathcal{L}_0$	$\mathcal{L}_0 \rightarrow \mathcal{L}_1 = [n,a,b,\$]$
n	$\rightarrow 0$	first symbol of $\mathcal{L}_1$	
b	$\rightarrow 2$	third symbol of $\mathcal{L}_1$	$\mathcal{L}_1 \rightarrow \mathcal{L}_2 = [b,n,a,\$]$
$	$\rightarrow 3$	fourth symbol of $\mathcal{L}_2$	$\mathcal{L}_2 \rightarrow \mathcal{L}_3 = [\$,b,n,a]$
a	$\rightarrow 3$	fourth symbol of $\mathcal{L}_3$	$\mathcal{L}_3 \rightarrow \mathcal{L}_4 = [a,\$,b,n]$
a	$\rightarrow 0$	first symbol of $\mathcal{L}_4$	

MTF(annb$aa) ⇒ 0202330

0 becomes much more probable than other digits → Huffman likes that!

$\mathcal{L} \equiv$ the alphabet Σ from which are drawn the symbols (in general the ASCII table).

Final overview of bzip2 compression algorithm

1st step: BWT → sort data to create runs of similar symbols.

2nd step: MTF → encode sorted data into digits with high anisotropy.

3rd step: Huffman → 0 gets a short encoding ⇒ huge compression.

✓ Most of the time more efficient than other compression algorithms (.zip and .gz, based on DEFLATE algorithm).

```
-rw-r--r-- 1 gtochon lrde  20K mars  23 17:54 random_english.txt
-rw-r--r-- 1 gtochon lrde 6,6K mars  23 17:54 random_english.txt.bz2
-rw-r--r-- 1 gtochon lrde 7,9K mars  23 17:54 random_english.txt.gz
-rw-r--r-- 1 gtochon lrde 8,0K mars  23 18:38 random_english.txt.zip
```

✗ Notably slower for the compression stage (but not for decompression).

LZW compression algorithm

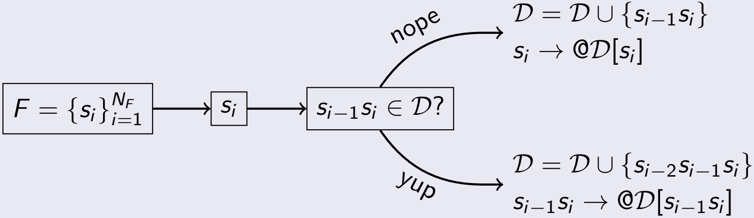
- → Improvement of the LZ78 algorithm proposed by Abraham Lempel and Jacob Ziv in 1978.
- → Published by Terry Welch in 1984.
- → Can be considered with LZ78 as the first unsupervised dictionary learning algorithms.


Abraham Lempel


Jacob Ziv

LZW encoding

Idea: Create a dictionary $\mathcal{D}$ based on the sequences of symbols encountered in the data, and replace known sequences of symbols by their address in $\mathcal{D}$.



Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			

What is happening?

1. T is the first read character.
2. Buffer is empty
3. T is already in $\mathcal{D}$.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]

What is happening?

1. T is put in the buffer.
2. O is read.
3. The sequence TO is created and tested to belong to $\mathcal{D}$.
4. TO is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 256.
5. T is encoded by its address in $\mathcal{D}$.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]

What is happening?

1. O is put in the buffer.
2. ␣ is read.
3. The sequence O␣ is created and tested to belong to $\mathcal{D}$.
4. O␣ is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 257.
5. O is encoded by its address in $\mathcal{D}$.

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]

What is happening?

1. ␣ is put in the buffer.
2. B is read.
3. The sequence ␣B is created and tested to belong to $\mathcal{D}$.
4. ␣B is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 258.
5. ␣ is encoded by its address in $\mathcal{D}$.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	_	O_	257	$@\mathcal{D}$ [O]
_	B	_B	258	$@\mathcal{D}$ [_]
B	E	BE	259	$@\mathcal{D}$ [B]
E	_	E_	260	$@\mathcal{D}$ [E]
_	O	_O	261	$@\mathcal{D}$ [_]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]
O	_	O_	257	@$\mathcal{D}$ [O]
_	B	_B	258	@$\mathcal{D}$ [_]
B	E	BE	259	@$\mathcal{D}$ [B]
E	_	E_	260	@$\mathcal{D}$ [E]
_	O	_O	261	@$\mathcal{D}$ [_]
O	R	OR	262	@$\mathcal{D}$ [O]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR...

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]
O	_	O_	257	@$\mathcal{D}$ [O]
_	B	_B	258	@$\mathcal{D}$ [_]
B	E	BE	259	@$\mathcal{D}$ [B]
E	_	E_	260	@$\mathcal{D}$ [E]
_	O	_O	261	@$\mathcal{D}$ [_]
O	R	OR	262	@$\mathcal{D}$ [O]
R	_	R_	263	@$\mathcal{D}$ [R]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR...

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]
O	␣	O␣	257	@$\mathcal{D}$ [O]
␣	B	␣B	258	@$\mathcal{D}$ [␣]
B	E	BE	259	@$\mathcal{D}$ [B]
E	␣	E␣	260	@$\mathcal{D}$ [E]
␣	O	␣O	261	@$\mathcal{D}$ [␣]
O	R	OR	262	@$\mathcal{D}$ [O]
R	␣	R␣	263	@$\mathcal{D}$ [R]
␣	N	␣N	264	@$\mathcal{D}$ [␣]
N	O	NO	265	@$\mathcal{D}$ [N]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR, NO...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]
O	␣	O␣	257	@$\mathcal{D}$ [O]
␣	B	␣B	258	@$\mathcal{D}$ [␣]
B	E	BE	259	@$\mathcal{D}$ [B]
E	␣	E␣	260	@$\mathcal{D}$ [E]
␣	O	␣O	261	@$\mathcal{D}$ [␣]
O	R	OR	262	@$\mathcal{D}$ [O]
R	␣	R␣	263	@$\mathcal{D}$ [R]
␣	N	␣N	264	@$\mathcal{D}$ [␣]
N	O	NO	265	@$\mathcal{D}$ [N]
O	T	OT	266	@$\mathcal{D}$ [O]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR, NO...

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]
O	_	O_	257	@$\mathcal{D}$ [O]
_	B	_B	258	@$\mathcal{D}$ [_]
B	E	BE	259	@$\mathcal{D}$ [B]
E	_	E_	260	@$\mathcal{D}$ [E]
_	O	_O	261	@$\mathcal{D}$ [_]
O	R	OR	262	@$\mathcal{D}$ [O]
R	_	R_	263	@$\mathcal{D}$ [R]
_	N	_N	264	@$\mathcal{D}$ [_]
N	O	NO	265	@$\mathcal{D}$ [N]
O	T	OT	266	@$\mathcal{D}$ [O]
T	_	T_	267	@$\mathcal{D}$ [T]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR, NO...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	_	O_	257	$@\mathcal{D}$ [O]
_	B	_B	258	$@\mathcal{D}$ [_]
B	E	BE	259	$@\mathcal{D}$ [B]
E	_	E_	260	$@\mathcal{D}$ [E]
_	O	_O	261	$@\mathcal{D}$ [_]
O	R	OR	262	$@\mathcal{D}$ [O]
R	_	R_	263	$@\mathcal{D}$ [R]
_	N	_N	264	$@\mathcal{D}$ [_]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	_	T_	267	$@\mathcal{D}$ [T]
_	T	_T	268	$@\mathcal{D}$ [_]

What is happening?

It keeps adding in $\mathcal{D}$ new sequences of two symbols that were never encountered before...

Note that some sequences should come back pretty often, such as TO, BE, OR, NO...

And surely enough...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O	TO		

What is happening?

1. T is put in the buffer.
2. O is read.
3. The sequence TO is created and tested to belong to $\mathcal{D}$.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O	TO		

What is happening?

1. T is put in the buffer.
2. O is read.
3. The sequence TO is created and tested to belong to $\mathcal{D}$.
4. TO is already in $\mathcal{D}$, at the address 256 $\rightarrow$ it is not inserted in $\mathcal{D}$ again.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O			✗
TO				

What is happening?

1. T is put in the buffer.
2. O is read.
3. The sequence TO is created and tested to belong to $\mathcal{D}$.
4. TO is already in $\mathcal{D}$, at the address 256 $\rightarrow$ it is not inserted in $\mathcal{D}$ again.
5. T is not encoded by its address (nothing is output), but TO is put in the buffer instead.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O			
TO	␣	TO␣	269	$@\mathcal{D}$ [TO]

What is happening?

1. TO is in the buffer.
2. ␣ is read.
3. The sequence TO␣ is created and tested to belong to $\mathcal{D}$.
4. TO␣ is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 269.
5. TO is encoded by its address in $\mathcal{D}$.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	_	O_	257	$@\mathcal{D}$ [O]
_	B	_B	258	$@\mathcal{D}$ [_]
B	E	BE	259	$@\mathcal{D}$ [B]
E	_	E_	260	$@\mathcal{D}$ [E]
_	O	_O	261	$@\mathcal{D}$ [_]
O	R	OR	262	$@\mathcal{D}$ [O]
R	_	R_	263	$@\mathcal{D}$ [R]
_	N	_N	264	$@\mathcal{D}$ [_]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	_	T_	267	$@\mathcal{D}$ [T]
_	T	_T	268	$@\mathcal{D}$ [_]
T	O			
TO	_	TO_	269	256

What is happening?

1. TO is in the buffer.
2. _ is read.
3. The sequence TO_ is created and tested to belong to $\mathcal{D}$.
4. TO_ is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 269.
5. TO is encoded by its address in $\mathcal{D}$, *i.e.*, 256.

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O			$@\mathcal{D}$ [%]
TO	␣	TO␣	269	256

What is happening?

1. TO is in the buffer.
2. ␣ is read.
3. The sequence TO␣ is created and tested to belong to $\mathcal{D}$.
4. TO␣ is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 269.
5. TO is encoded by its address in $\mathcal{D}$, *i.e.*, 256.
6. But wait, 256 requires 9 bits to be encoded (instead of 8 bits so far). $\rightarrow$ needs to emit a special character % to prevent the decompressor that it shall read further addresses on 9 bits and no longer on 8 bits.

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O			$@\mathcal{D}$ [%]
TO	␣	TO␣	269	256
␣	B			

What is happening?

1. ␣ is put in the buffer.
2. B is read.
3. The sequence ␣B is created and tested to belong to $\mathcal{D}$.
4. ␣ is already in $\mathcal{D}$, at the address 258 $\rightarrow$ it is not inserted in $\mathcal{D}$ again.
5. Nothing is output, and ␣B is put in the buffer.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	@$\mathcal{D}$ [T]
O	_	O_	257	@$\mathcal{D}$ [O]
_	B	_B	258	@$\mathcal{D}$ [_]
B	E	BE	259	@$\mathcal{D}$ [B]
E	_	E_	260	@$\mathcal{D}$ [E]
_	O	_O	261	@$\mathcal{D}$ [_]
O	R	OR	262	@$\mathcal{D}$ [O]
R	_	R_	263	@$\mathcal{D}$ [R]
_	N	_N	264	@$\mathcal{D}$ [_]
N	O	NO	265	@$\mathcal{D}$ [N]
O	T	OT	266	@$\mathcal{D}$ [O]
T	_	T_	267	@$\mathcal{D}$ [T]
_	T	_T	268	@$\mathcal{D}$ [_]
T	O			@$\mathcal{D}$ [%]
TO	_	TO_	269	256
_	B			
_B	E	_BE	270	258

What is happening?

1. _B is in the buffer.
2. E is read.
3. The sequence _BE is created and tested to belong to $\mathcal{D}$.
4. _BE is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 270.
5. _B is encoded by its address in $\mathcal{D}$, *i.e.*, 258.

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O			$@\mathcal{D}$ [%]
TO	␣	TO␣	269	256
␣	B			
␣B	E	␣BE	270	258
E	$			$@\mathcal{D}$ [E]

What is happening?

1. E is put in the buffer.
2. The EOF character $ is read.
3. The dictionary update stops, and E is encoded by its address in $\mathcal{D}$.

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	␣	O␣	257	$@\mathcal{D}$ [O]
␣	B	␣B	258	$@\mathcal{D}$ [␣]
B	E	BE	259	$@\mathcal{D}$ [B]
E	␣	E␣	260	$@\mathcal{D}$ [E]
␣	O	␣O	261	$@\mathcal{D}$ [␣]
O	R	OR	262	$@\mathcal{D}$ [O]
R	␣	R␣	263	$@\mathcal{D}$ [R]
␣	N	␣N	264	$@\mathcal{D}$ [␣]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	␣	T␣	267	$@\mathcal{D}$ [T]
␣	T	␣T	268	$@\mathcal{D}$ [␣]
T	O			$@\mathcal{D}$ [%]
TO	␣	TO␣	269	256
␣	B			
␣B	E	␣BE	270	258
E	$			$@\mathcal{D}$ [E]

Compression performance:

- Without compression $\rightarrow 18 \times 8 = 144$ bits.
- With compression $\rightarrow 14 \times 8 + 3 \times 9 = 139$ bits.

OK, it is not so impressive for this example...

Example

LZW encoding

Let's encode $F =$ TO BE OR NOT TO BE with $\mathcal{D} \equiv$ ASCII table.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	T			
T	O	TO	256	$@\mathcal{D}$ [T]
O	_	O_	257	$@\mathcal{D}$ [O]
_	B	_B	258	$@\mathcal{D}$ [_]
B	E	BE	259	$@\mathcal{D}$ [B]
E	_	E_	260	$@\mathcal{D}$ [E]
_	O	_O	261	$@\mathcal{D}$ [_]
O	R	OR	262	$@\mathcal{D}$ [O]
R	_	R_	263	$@\mathcal{D}$ [R]
_	N	_N	264	$@\mathcal{D}$ [_]
N	O	NO	265	$@\mathcal{D}$ [N]
O	T	OT	266	$@\mathcal{D}$ [O]
T	_	T_	267	$@\mathcal{D}$ [T]
_	T	_T	268	$@\mathcal{D}$ [_]
T	O			$@\mathcal{D}$ [%]
TO	_	TO_	269	256
_	B			
_B	E	_BE	270	258
E	$			$@\mathcal{D}$ [E]

Compression performance:

But on longer strings, when the dictionary $\mathcal{D}$ has sufficiently grown, the gain becomes really significant.

Ex: How much wood would a woodchuck chuck if a woodchuck could chuck wood?

- Without compression $\rightarrow 70 \times 8 = 560$ bits.
- With compression $\rightarrow 14 \times 8 + 9 \times 31 = 391$ bits (check it yourself).

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			

What is happening?

1. $@\mathcal{D}$ [T] is received by the decoder.
2. T is recognized by looking up in $\mathcal{D}$.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	@$\mathcal{D}$ [T]			
T	@$\mathcal{D}$ [O]	TO	256	T

What is happening?

1. T is put in the buffer.
2. O is recognized from @$\mathcal{D}$ [O].
3. The sequence TO is created and tested to belong to $\mathcal{D}$.
4. TO is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 256.
5. T is output.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O

What is happening?

1. O is put in the buffer.
2. _ is recognized from $@\mathcal{D}$ [_].
3. The sequence O_ is created and tested to belong to $\mathcal{D}$.
4. O_ is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 257.
5. O is output.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O
_	$@\mathcal{D}$ [B]	_B	258	_
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [_]	E_	260	E
_	$@\mathcal{D}$ [O]	_O	261	_
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [_]	R_	263	R
_	$@\mathcal{D}$ [N]	_N	264	_
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [_]	T_	267	T

What is happening?

And so on, as long as no special character is encountered.

The sequences that are recreated by this process are strictly identical to those that were generated during the encoding stage.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [␣]	O␣	257	O
␣	$@\mathcal{D}$ [B]	␣B	258	␣
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [␣]	E␣	260	E
␣	$@\mathcal{D}$ [O]	␣O	261	␣
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [␣]	R␣	263	R
␣	$@\mathcal{D}$ [N]	␣N	264	␣
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [␣]	T␣	267	T
␣	$@\mathcal{D}$ [%]			

What is happening?

1. ␣ is put in the buffer.
2. The special character % is recognized from $@\mathcal{D}$ [%] $\rightarrow$ all following addresses will have to be decoded on 9 bits instead of 8.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	@$\mathcal{D}$ [T]			
T	@$\mathcal{D}$ [O]	TO	256	T
O	@$\mathcal{D}$ [␣]	O␣	257	O
␣	@$\mathcal{D}$ [B]	␣B	258	␣
B	@$\mathcal{D}$ [E]	BE	259	B
E	@$\mathcal{D}$ [␣]	E␣	260	E
␣	@$\mathcal{D}$ [O]	␣O	261	␣
O	@$\mathcal{D}$ [R]	OR	262	O
R	@$\mathcal{D}$ [␣]	R␣	263	R
␣	@$\mathcal{D}$ [N]	␣N	264	␣
N	@$\mathcal{D}$ [O]	NO	265	N
O	@$\mathcal{D}$ [T]	OT	266	O
T	@$\mathcal{D}$ [␣]	T␣	267	T
␣	@$\mathcal{D}$ [%]			
␣				

What is happening?

1. ␣ is in the buffer.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O
_	$@\mathcal{D}$ [B]	_B	258	_
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [_]	E_	260	E
_	$@\mathcal{D}$ [O]	_O	261	_
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [_]	R_	263	R
_	$@\mathcal{D}$ [N]	_N	264	_
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [_]	T_	267	T
_	$@\mathcal{D}$ [%]			
_	256			

What is happening?

1. _ is in the buffer.
2. 256 is received.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [␣]	O␣	257	O
␣	$@\mathcal{D}$ [B]	␣B	258	␣
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [␣]	E␣	260	E
␣	$@\mathcal{D}$ [O]	␣O	261	␣
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [␣]	R␣	263	R
␣	$@\mathcal{D}$ [N]	␣N	264	␣
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [␣]	T␣	267	T
␣	$@\mathcal{D}$ [%]			
␣	256			

What is happening?

1. ␣ is in the buffer.
2. 256 is received, that is, $@\mathcal{D}$ [TO].

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O
_	$@\mathcal{D}$ [B]	_B	258	_
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [_]	E_	260	E
_	$@\mathcal{D}$ [O]	_O	261	_
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [_]	R_	263	R
_	$@\mathcal{D}$ [N]	_N	264	_
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [_]	T_	267	T
_	$@\mathcal{D}$ [%]			
_	$@\mathcal{D}$ [TO]	_T		

What is happening?

1. _ is in the buffer.
2. 256 is received, that is, $@\mathcal{D}$ [TO].
3. The sequence _T is created using _ in the buffer and the first letter of the received sequence TO.

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O
_	$@\mathcal{D}$ [B]	_B	258	_
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [_]	E_	260	E
_	$@\mathcal{D}$ [O]	_O	261	_
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [_]	R_	263	R
_	$@\mathcal{D}$ [N]	_N	264	_
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [_]	T_	267	T
_	$@\mathcal{D}$ [%]			
_	$@\mathcal{D}$ [TO]	_T	268	

What is happening?

1. _ is in the buffer.
2. 256 is received, that is, $@\mathcal{D}$ [TO].
3. The sequence _T is created using _ in the buffer and the first letter of the received sequence TO.
4. _T is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 268.

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O
_	$@\mathcal{D}$ [B]	_B	258	_
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [_]	E_	260	E
_	$@\mathcal{D}$ [O]	_O	261	_
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [_]	R_	263	R
_	$@\mathcal{D}$ [N]	_N	264	_
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [_]	T_	267	T
_	$@\mathcal{D}$ [%]			
_	$@\mathcal{D}$ [TO]	_T	268	_

What is happening?

1. _ is in the buffer.
2. 256 is received, that is, $@\mathcal{D}$ [TO].
3. The sequence _T is created using _ in the buffer and the first letter of the received sequence TO.
4. _T is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 268.
5. _ is output.

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [␣]	O␣	257	O
␣	$@\mathcal{D}$ [B]	␣B	258	␣
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [␣]	E␣	260	E
␣	$@\mathcal{D}$ [O]	␣O	261	␣
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [␣]	R␣	263	R
␣	$@\mathcal{D}$ [N]	␣N	264	␣
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [␣]	T␣	267	T
␣	$@\mathcal{D}$ [%]			
␣	$@\mathcal{D}$ [TO]	␣T	268	␣
TO	258			

What is happening?

1. TO is put in the buffer.
2. 258 is received.

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [␣]	O␣	257	O
␣	$@\mathcal{D}$ [B]	␣B	258	␣
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [␣]	E␣	260	E
␣	$@\mathcal{D}$ [O]	␣O	261	␣
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [␣]	R␣	263	R
␣	$@\mathcal{D}$ [N]	␣N	264	␣
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [␣]	T␣	267	T
␣	$@\mathcal{D}$ [%]			
␣	$@\mathcal{D}$ [TO]	␣T	268	␣
TO	$@\mathcal{D}$ [␣B]			

What is happening?

1. TO is put in the buffer.
2. 258 is received, that is, ␣B.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	$@\mathcal{D}[\cdot]$	Output
	$@\mathcal{D}$ [T]			
T	$@\mathcal{D}$ [O]	TO	256	T
O	$@\mathcal{D}$ [_]	O_	257	O
_	$@\mathcal{D}$ [B]	_B	258	_
B	$@\mathcal{D}$ [E]	BE	259	B
E	$@\mathcal{D}$ [_]	E_	260	E
_	$@\mathcal{D}$ [O]	_O	261	_
O	$@\mathcal{D}$ [R]	OR	262	O
R	$@\mathcal{D}$ [_]	R_	263	R
_	$@\mathcal{D}$ [N]	_N	264	_
N	$@\mathcal{D}$ [O]	NO	265	N
O	$@\mathcal{D}$ [T]	OT	266	O
T	$@\mathcal{D}$ [_]	T_	267	T
_	$@\mathcal{D}$ [%]			
_	$@\mathcal{D}$ [TO]	_T	268	_
TO	$@\mathcal{D}$ [_B]	TO_	269	TO

What is happening?

1. TO is put in the buffer.
2. 258 is received, that is, _B.
3. The sequence TO_ is created using TO in the buffer and the first letter _ of the received sequence _B.
4. TO_ is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 269.
5. TO is output.

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D} \cup \{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	@$\mathcal{D}$ [T]			
T	@$\mathcal{D}$ [O]	TO	256	T
O	@$\mathcal{D}$ [␣]	O␣	257	O
␣	@$\mathcal{D}$ [B]	␣B	258	␣
B	@$\mathcal{D}$ [E]	BE	259	B
E	@$\mathcal{D}$ [␣]	E␣	260	E
␣	@$\mathcal{D}$ [O]	␣O	261	␣
O	@$\mathcal{D}$ [R]	OR	262	O
R	@$\mathcal{D}$ [␣]	R␣	263	R
␣	@$\mathcal{D}$ [N]	␣N	264	␣
N	@$\mathcal{D}$ [O]	NO	265	N
O	@$\mathcal{D}$ [T]	OT	266	O
T	@$\mathcal{D}$ [␣]	T␣	267	T
␣	@$\mathcal{D}$ [%]			
␣	@$\mathcal{D}$ [TO]	␣T	268	␣
TO	@$\mathcal{D}$ [␣B]		269	TO
␣B	@$\mathcal{D}$ [E]	␣BE	270	␣B

What is happening?

1. ␣B is put in the buffer.
2. E is recognized from @$\mathcal{D}$ [E].
3. The sequence ␣BE is created and tested to belong to $\mathcal{D}$.
4. ␣BE is not in $\mathcal{D}$ yet $\rightarrow$ it is inserted at the next available address, *i.e.* 270.
5. ␣B is output.

Example

LZW decoding

The LZW decoding scheme reconstructs the dictionary $\mathcal{D}$ from the ASCII table in a very symmetric fashion.

Buffer	Input	$\mathcal{D}\cup\{\cdot\}$	@$\mathcal{D}[\cdot]$	Output
	@$\mathcal{D}$ [T]			
T	@$\mathcal{D}$ [O]	TO	256	T
O	@$\mathcal{D}$ [_]	O_	257	O
_	@$\mathcal{D}$ [B]	_B	258	_
B	@$\mathcal{D}$ [E]	BE	259	B
E	@$\mathcal{D}$ [_]	E_	260	E
_	@$\mathcal{D}$ [O]	_O	261	_
O	@$\mathcal{D}$ [R]	OR	262	O
R	@$\mathcal{D}$ [_]	R_	263	R
_	@$\mathcal{D}$ [N]	_N	264	_
N	@$\mathcal{D}$ [O]	NO	265	N
O	@$\mathcal{D}$ [T]	OT	266	O
T	@$\mathcal{D}$ [_]	T_	267	T
_	@$\mathcal{D}$ [%]			
_	@$\mathcal{D}$ [TO]	_T	268	_
TO	@$\mathcal{D}$ [_B]		269	TO
_B	@$\mathcal{D}$ [E]	_BE	270	_B
E	$			E

What is happening?

1. E is put in the buffer.
2. The EOF character $ is read.
3. The dictionary update stops, E is recognized from @$\mathcal{D}$ [E] and is output.

Concluding remarks on LZW

The dictionaries generated by the encoding and the decoding stages are strictly identical.

The dictionary $\mathcal{D}$ cannot grow indefinitely.

$\rightarrow$ Most of the time limited to 12 bits, *i.e.*, 4096 entries.

$\rightarrow$ Can be reset with another special character if needed.

LZW is used in the GIF (*Graphics Interchange Format*) encoding format for images.

$\rightarrow$ Pixel values are between 0 and 255, hence all 8 bits combinaisons are required.

$\rightarrow$ Directly mapped to 9 bits encoding (also to accomodate for special characters).

$\rightarrow$ Dictionary size is precisely limited to 12 bits.