CS 240 – Data Structures and Data Management

Module 10: Data Compression

Olga Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2025

version 2025–03–19 15:00

Outline

Outline

10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

Data Compression Introduction

The problem: How to store and transmit data efficiently?

Source text The original data, string S of characters from the **source alphabet** Σ_S

Coded text The encoded data, string C of characters from the **code alphabet** Σ_C

Encoding An algorithm mapping source texts to coded texts

Decoding An algorithm mapping coded texts back to their original source text

$$S \xrightarrow{\text{encode}} C \overset{\text{transmit}}{\dashrightarrow} C \xrightarrow{\text{decode}} S$$

- Source "text" can be any sort of data (not always text!)
- The code alphabet Σ_C is usually $\{0, 1\}$.
- We consider here only **lossless** compression: Can recover S from C without error.

Judging Encoding Schemes

Main objective: For data compression, we want to minimize the size of the coded text. We will measure the **compression ratio**:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

Examples:

$$73 = (73)_{10} \quad \rightarrow \quad (1001001)_2 \quad \text{has compression ratio } \tfrac{7 \cdot \log 2}{2 \cdot \log 10} \approx 1.05$$

$$127 = (127)_{10} \quad \rightarrow \quad (7F)_{16} \quad \text{has compression ratio } \tfrac{2 \cdot \log 16}{3 \cdot \log 10} \approx 0.8$$

Judging Encoding Schemes

Main objective: For data compression, we want to minimize the size of the coded text. We will measure the **compression ratio**:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

Examples:

$73 = (73)_{10} \quad \rightarrow \quad (1001001)_2$ has compression ratio $\frac{7 \cdot \log 2}{2 \cdot \log 10} \approx 1.05$

$127 = (127)_{10} \quad \rightarrow \quad (7F)_{16}$ has compression ratio $\frac{2 \cdot \log 16}{3 \cdot \log 10} \approx 0.8$

We also consider the efficiency of the encoding/decoding algorithms.

- We always need time $\Omega(|S| + |C|)$, but sometimes need more

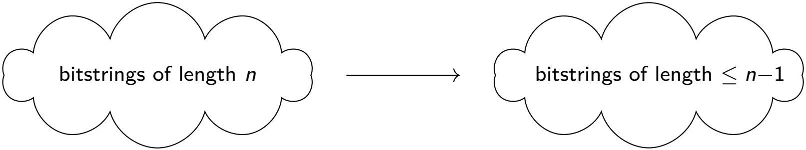
Other possible goals (not studied here $\rightarrow$ CS489):

- Reliability (e.g. error-correcting codes)
- Security (e.g. encryption)

Impossibility of compressing

Observation: No lossless encoding scheme can have compression ratio < 1 for *all* input strings.

Proof (only for $\Sigma_S = \Sigma_C = \{0, 1\}$): Fix one size n.
Assume for contradiction that all length-n strings get shorter.

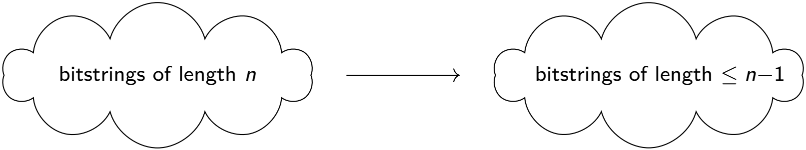


How big are these sets?

Impossibility of compressing

Observation: No lossless encoding scheme can have compression ratio < 1 for *all* input strings.

Proof (only for $\Sigma_S = \Sigma_C = \{0, 1\}$): Fix one size n.
Assume for contradiction that all length-n strings get shorter.

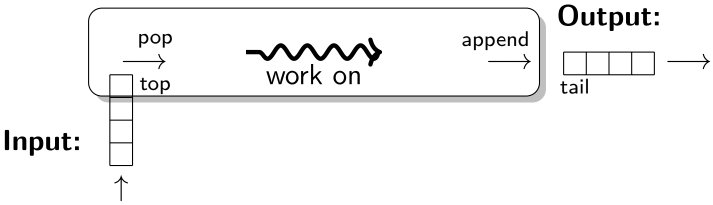


How big are these sets?

- So we cannot hope to prove good worst-case compression bounds.
- But real-life data can (usually) be compressed well due to patterns.

Detour: Streams (Review)

The texts are often *huge* and will not fit onto the computer's memory.
Therefore we usually use streams to store S and C.



- Input-stream ($\sim$ `std::cin`): Read one character at a time.
  (We use a stack-like interface: *top*/*pop*/*is-empty*.)
- Output-stream ($\sim$ `std::cout`): Write one character at a time.
  (We use queue-like interface: *append*.)
- Advantage: can start processing text while loading.
- Some algorithm will need to *reset* the input-stream.

Outline

10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

Character Encodings

A **character encoding** (or **single-character encoding**) maps each character in the source alphabet to a string in code alphabet.

$$E : \Sigma_S \to \Sigma_C^*$$

Example: ASCII

NULL	...	!	...	0	...	A	...
0	...	21	...	30	...	65	...
0000000	...	0010101	...	0011110	...	1000001	...

- ASCII is a (width-7) **fixed-width code**: Each **codeword** $E(c)$ has the same length.
- Encoding/Decoding is easy: just concatenate/decode the next 7 bits.

$APPLE \leftrightarrow (65, 80, 80, 76, 69) \leftrightarrow 1000001 \frown 1010000 \frown 1010000 \frown 1001100 \frown 1000101$

- Other (earlier) examples of fixed-width codes: Caesar shift, Baudot code, Murray code
- Fixed-width codes do not compress.

Variable-Length Codes

Better idea: **Variable-length codes**

- **Motivation**: Some letters in Σ_S occur more often than others. For example, consider the frequency of letters in typical English text:

e	12.70%	d	4.25%	p	1.93%
t	9.06%	l	4.03%	b	1.49%
a	8.17%	c	2.78%	v	0.98%
o	7.51%	u	2.76%	k	0.77%
i	6.97%	m	2.41%	j	0.15%
n	6.75%	w	2.36%	x	0.15%
s	6.33%	f	2.23%	q	0.10%
h	6.09%	g	2.02%	z	0.07%
r	5.99%	y	1.97%		

Variable-Length Codes

Better idea: **Variable-length codes**

- **Motivation**: Some letters in Σ_S occur more often than others.
  For example, consider the frequency of letters in typical English text:

e	12.70%	d	4.25%	p	1.93%
t	9.06%	l	4.03%	b	1.49%
a	8.17%	c	2.78%	v	0.98%
o	7.51%	u	2.76%	k	0.77%
i	6.97%	m	2.41%	j	0.15%
n	6.75%	w	2.36%	x	0.15%
s	6.33%	f	2.23%	q	0.10%
h	6.09%	g	2.02%	z	0.07%
r	5.99%	y	1.97%		

- **Idea:** Let's use shorter codes for more frequent characters.

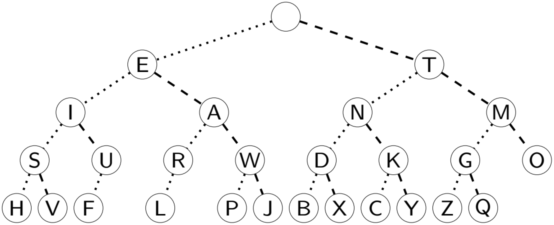
So as before, map source alphabet to codewords $E : \Sigma_S \rightarrow \Sigma_C^*$

- But not all codewords must have the same length.
- This ought to make the code text shorter.

Variable-Length Codes

Example 1: Morse code.

A	•−	N	−•
B	−•••	O	−−−
C	−•−•	P	•−−•
D	−••	Q	−−•−
E	•	R	•−•
F	••−•	S	•••
G	−−•	T	−
H	••••	U	••−
I	••	V	•••−
J	•−−−	W	•−−
K	−•−	X	−••−
L	•−••	Y	−•−−
M	−−	Z	−−••



Example 2: UTF-8 encoding of Unicode ($\sim$ 150,000 characters):

- Encodes any Unicode character using 1-4 bytes

Encoding

Assume that we have some character encoding $E : \Sigma_S \to \Sigma_C^*$.

- Note that E is a dictionary with keys in Σ_S.

> *singleChar::encoding*(E, S, C)
> E : the encoding dictionary
> S: input-stream with characters in Σ_S, C: output-stream of bits
> 1. **while** S is non-empty
> 2. $\quad$ $w \leftarrow E.$*search*($S.$*pop*())
> 3. $\quad$ append each bit of w to C

Example: Encode `AN␣ANT` with the following code:

$c \in \Sigma_S$	␣	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

`AN␣ANT` $\to$

Encoding

Assume that we have some character encoding $E : \Sigma_S \to \Sigma_C^*$.

- Note that E is a dictionary with keys in Σ_S.

singleChar::encoding(E, S, C)
E : the encoding dictionary
S: input-stream with characters in Σ_S, C: output-stream of bits

while S is non-empty
    $w \leftarrow E.$*search*$(S.$*pop*$())$
    append each bit of w to C

Example: Encode `AN␣ANT` with the following code:

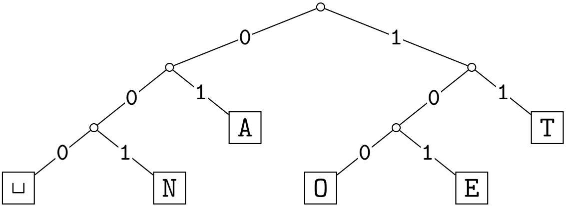
$c \in \Sigma_S$	␣	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

`AN␣ANT` $\to$ 010010000100111

Decoding

The **decoding algorithm** must map Σ_C^* to Σ_S^*.

- The code must be lossless, i.e., *uniquely decodable*.
  - This is false for Morse code as described!
    • ▬ ▬ • ▬ ▬ ▬ decodes to `WATT` and `ANO` and `WJ`.
    (Morse code uses pause as end-sentinel to avoid ambiguity.)
- From now on only consider **prefix-free** codes E:
  no codeword is a prefix of another
- This corresponds to a *trie* with characters of Σ_S only at the leaves.



- The codewords need no end-sentinel $ if codes are prefix-free.

Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable.

prefixFree::decoding(C, S, T)
C: input-stream with characters in Σ_C, S: output-stream, T : encoding-trie

while C is non-empty
    $z \leftarrow T.root$
    while z is not a leaf
        if C is empty or z has no child labelled $C.$*top*()
            return "invalid encoding"
        $z \leftarrow$ child of z that is labelled with $C.$*pop*()
    $S.$*append*(character stored at z)

Run-time: $O(|C|)$.

Example: Decode 111000001010111

Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable.

prefixFree::decoding(C, S, T)
C: input-stream with characters in Σ_C, S: output-stream, T : encoding-trie

```
while C is non-empty
    z ← T.root
    while z is not a leaf
        if C is empty or z has no child labelled C.top()
            return "invalid encoding"
        z ← child of z that is labelled with C.pop()
    S.append(character stored at z)
```

Run-time: $O(|C|)$.

Example: Decode 111000001010111 $\rightarrow$ `TO␣EAT`

Encoding from the Trie

We already explained encoding if code is stored in table:

$c \in \Sigma_S$	␣	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

Example: Encode AN␣ANT $\rightarrow$ 010010000100111

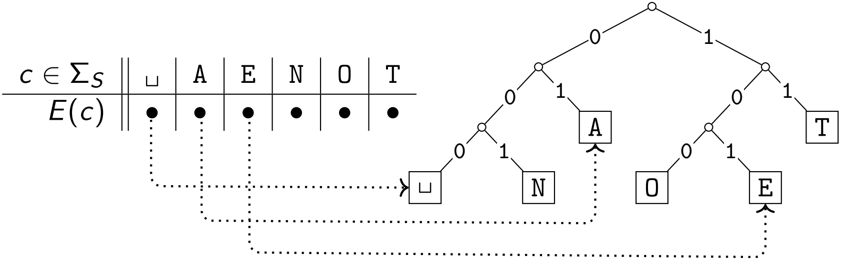
Encoding from the Trie

We already explained encoding if code is stored in table:

$c \in \Sigma_S$	␣	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

Example: Encode AN␣ANT $\rightarrow$ 010010000100111

- The table wastes space: Codewords may be quite long.
- Better idea: Store codewords via links to leaves in trie.



Encoding from trie

prefixFree::encoding(S, C, T)
S: input-stream with characters in Σ_S, C: output-stream, T: encoding trie

```
1.  E ← array of trie-nodes indexed by Σ_S
2.  for all leaves ℓ in T do E[character at ℓ] ← ℓ
3.  while S is non-empty
4.      w ← empty list of bits                  // codeword
5.      z ← E[S.pop()]
6.      while z is not the root
7.          w.add-to-front(character on link from z to its parent)
8.          z ← z.parent
9.      append each bit of w to C
```

- Run-time: $O(|T| + |C|)$
- We assume that all interior nodes of T have two children, otherwise encoding scheme can be improved (how?)
- Therefore $|T| \leq 2|\Sigma_S| - 1$ and run-time is $O(|\Sigma_S| + |C|)$

Outline

Huffman's Algorithm: Building the best trie

How to determine the "best" trie (for a given source text S)?

Idea: Frequent characters should have short codewords.
Equivalently: Infrequent characters should be 'far down' in trie.

Huffman's Algorithm: Building the best trie

How to determine the "best" trie (for a given source text S)?

Idea: Frequent characters should have short codewords.
Equivalently: Infrequent characters should be 'far down' in trie.

Greedy-algorithm: Always pair up most infrequent characters.

1. We store a set of encoding-tries.
   Initially each $c \in \Sigma_S$ adds "$\boxed{c}$" (height-0 trie holding c).
2. Our tries have a *weight*: sum of frequencies of all letters in trie.
   (We assume character-frequencies are pre-computed.)
3. Find the two tries with the minimum weight and merge them.
   (Corresponds to adding one bit to the encoding of each character.)
4. Repeat Step 3 until there is only one trie left

How should we store the tries to make this efficient?
A min-ordered heap! Step 3 needs two *delete-min*s and one *insert*

Example: Huffman tree construction

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:

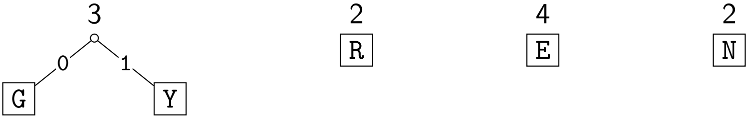
2	2	4	2	1
G	R	E	N	Y

Example: Huffman tree construction

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:

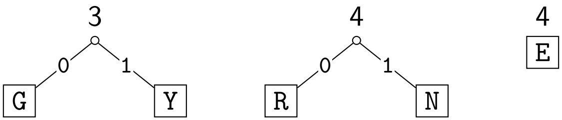


Example: Huffman tree construction

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:

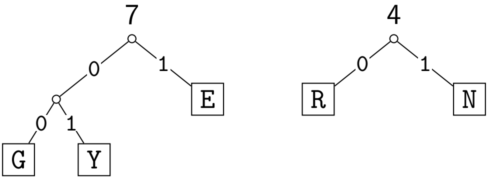


Example: Huffman tree construction

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:

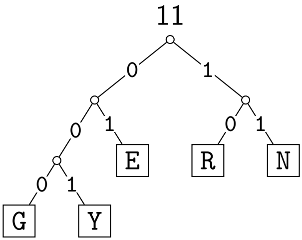


Example: Huffman tree construction

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:



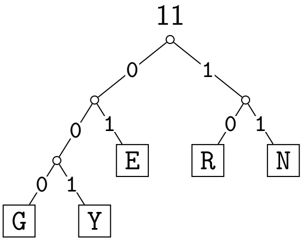
GREENENERGY $\rightarrow$

Example: Huffman tree construction

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:



GREENENERGY → 000‿10‿01‿01‿11‿01‿11‿01‿10‿000‿001

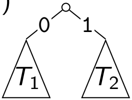
Compression ratio: $\frac{25}{11 \cdot \log 5} \approx 97\%$

Huffman's Algorithm: Pseudocode

Huffman::encoding(S, C)
S: input-stream with characters in Σ_S, C: output-stream

$f \leftarrow$ array indexed by Σ_S, initially all-0 `// frequencies`
while S is non-empty **do** increase $f[S.pop()]$ by 1

$Q \leftarrow$ min-oriented priority queue that stores tries `// initialize PQ`
for all $c \in \Sigma_S$ with $f[c] > 0$ **do** $Q.insert(\boxed{c}, f[c])$

while $Q.size() > 1$ **do** `// build decoding trie`
    $(T_1, f_1) \leftarrow Q.delete\text{-}min()$
    $(T_2, f_2) \leftarrow Q.delete\text{-}min()$
    $Q.insert$ (combined trie  $, f[c])$

$T \leftarrow Q.delete\text{-}min()$

$C.append$(encoding trie T) `// send trie`

This assumes that S has at least two distinct characters.

Huffman Coding Discussion

Can show: The constructed trie is *optimal* in the sense that C is shortest (among prefix-free single-character encodings with $\Sigma_C = \{0,1\}$) .

But:

- Constructed trie is *not unique* (unless we give tie-breaking rules).
- Decoder does not know the trie:
  - Either send decoding trie along (how to convert to bitstring?)
  - Or send character-frequencies and tie-breaking rules.
  - Either way, this adds to length of encoded text.
- *encoding* must pass through text twice (to compute frequencies and to encode). Cannot use a stream unless it can be re-set.

Run-time:

- Encoding: $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
- Decoding: $O(|C|)$

Many variations (what to do with unused characters, estimate frequencies, adaptively change encoding, ...)

Outline

Longer Patterns in Input

Huffman take advantage of frequent *single characters*.

Observation: Certain *substrings* are much more frequent than others.

- English text:
  Most frequent digraphs: `TH`, `ER`, `ON`, `AN`, `RE`, `HE`, `IN`, `ED`, `ND`, `HA`
  Most frequent trigraphs: `THE`, `AND`, `THA`, `ENT`, `ION`, `TIO`, `FOR`, `NDE`
- HTML: "`<a href`", "`<img src`", "`<br>`"
- Video: repeated background between frames, shifted sub-image

We now cover *Lempel-Ziv-Welch compression*, a **multi-character encoding**, i.e., multiple characters are encoded with one code-word.

Ingredient 1 for Lempel-Ziv-Welch compression: take advantage of frequent substrings *without* needing to know beforehand what they are.

Adaptive Dictionaries

ASCII, and UTF-8 use *fixed* dictionaries.

In Huffman, the dictionary is not fixed, but it is *static*: the dictionary is the same for the entire encoding/decoding.

Ingredient 2 for LZW: *adaptive encoding*:

- There is a fixed initial dictionary D_0. (Usually ASCII.)
- For $i \geq 0$, D_i is used to determine the ith output character
- After writing the ith character to output, encoder updates D_i to D_{i+1}

Challenge: Decoder must know (or reconstruct from coded text) how encoder changed the dictionary.

Lempel-Ziv-Welch Overview

For now: convert source-text S into a list of code-numbers in $\mathbb{N}_0$.

- Start with dictionary D_0 that stores ASCII (code-numbers $0, \ldots, 127$).
- Every step adds to dictionary a multi-character string, using code-numbers $128, 129, \ldots$.
- Encoding alternates two steps:
  1. *Find longest string* w (among remaining characters) that is already in D_i. So all of w can be encoded with one number.
  2. *Add the substring that would have been useful* to dictionary: add wc where c is the character that follows w in S.

Lempel-Ziv-Welch Overview

For now: convert source-text S into a list of code-numbers in $\mathbb{N}_0$.

- Start with dictionary D_0 that stores ASCII (code-numbers $0, \ldots, 127$).
- Every step adds to dictionary a multi-character string, using code-numbers $128, 129, \ldots$.
- Encoding alternates two steps:
  1. *Find longest string* w (among remaining characters) that is already in D_i. So all of w can be encoded with one number.
  2. *Add the substring that would have been useful* to dictionary: add wc where c is the character that follows w in S.

What data structure should the dictionary use?

Lempel-Ziv-Welch Overview

For now: convert source-text S into a list of code-numbers in $\mathbb{N}_0$.

- Start with dictionary D_0 that stores ASCII (code-numbers $0, \dots, 127$).
- Every step adds to dictionary a multi-character string, using code-numbers $128, 129, \dots$.
- Encoding alternates two steps:
  1. *Find longest string* w (among remaining characters) that is already in D_i. So all of w can be encoded with one number.
  2. *Add the substring that would have been useful* to dictionary: add wc where c is the character that follows w in S.

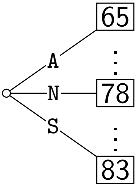
What data structure should the dictionary use?

- Need to match characters $\rightarrow$ use a trie
- To find w: parse in trie until we hit 'no such child'
- To add to D_i: add suitable child at that node

LZW Example

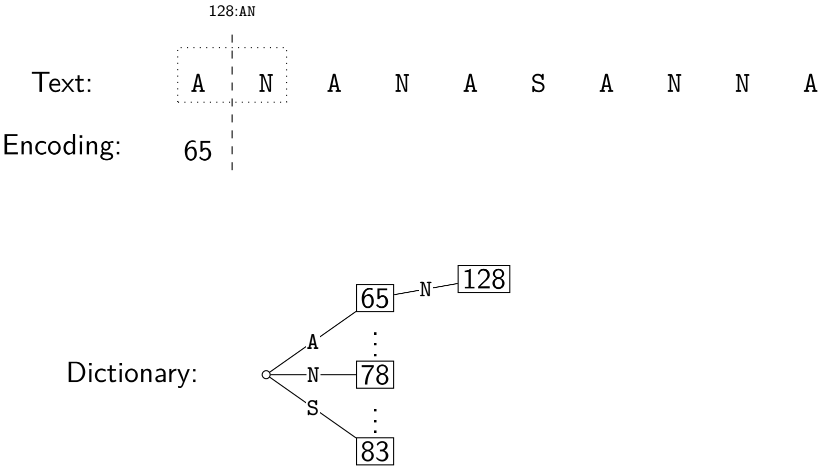
Text: A N A N A S A N N A

Dictionary:



(Parts of dictionary not shown)

LZW Example


128:AN
Text:
A N A N A S A N N A
Encoding:
65
Dictionary:
A
N
S
65
N
128
78
83

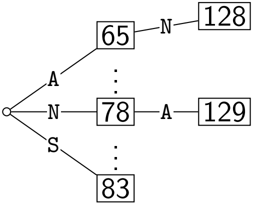
LZW Example

128:AN 129:NA

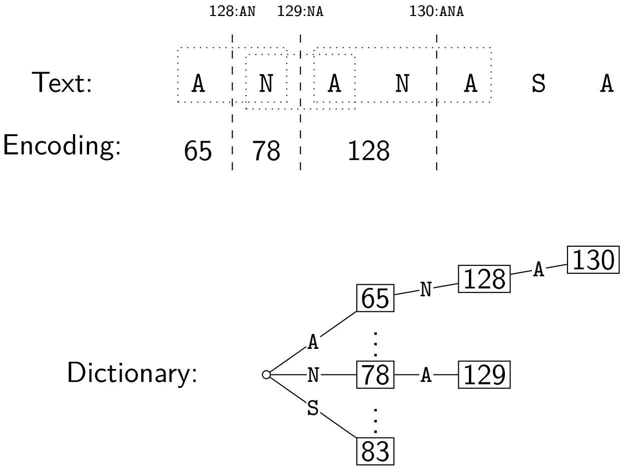
Text: A N A N A S A N N A

Encoding: 65 78

Dictionary:



LZW Example



128:AN
129:NA
130:ANA
Text:
A N A N A S A N N A
Encoding:
65 78 128
Dictionary:
A
N
S
65
78
83
N
128
A
130
A
129

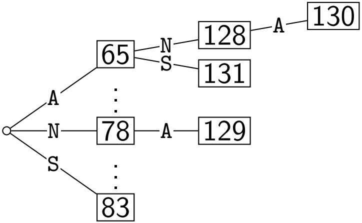
LZW Example

128:AN 129:NA 130:ANA 131:AS

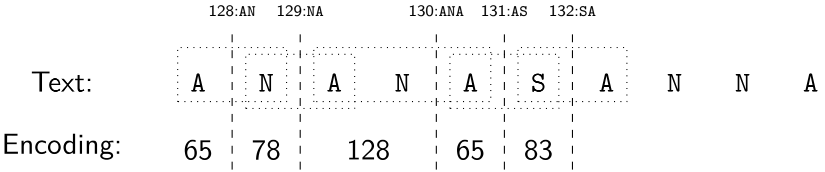
Text: A | N | A N | A | S A N N A

Encoding: 65 | 78 | 128 | 65 |

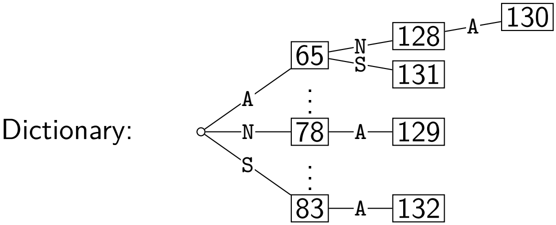
Dictionary:



LZW Example

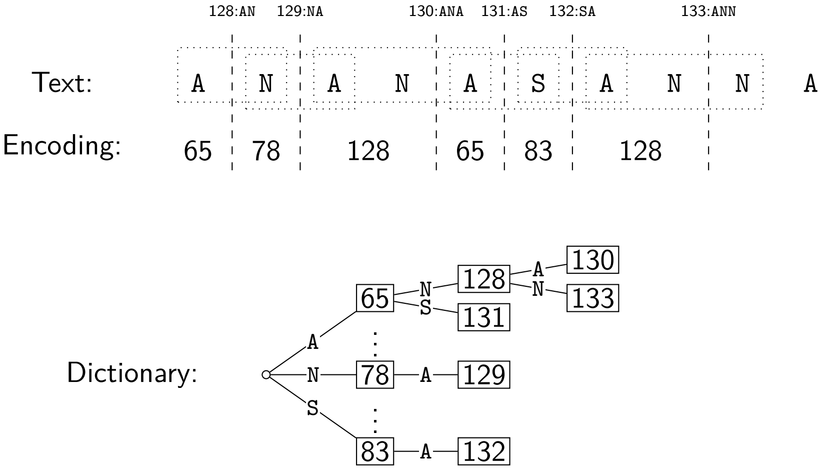


128:AN
129:NA
130:ANA
131:AS
132:SA
Text: A N A N A S A N N A
Encoding: 65 78 128 65 83

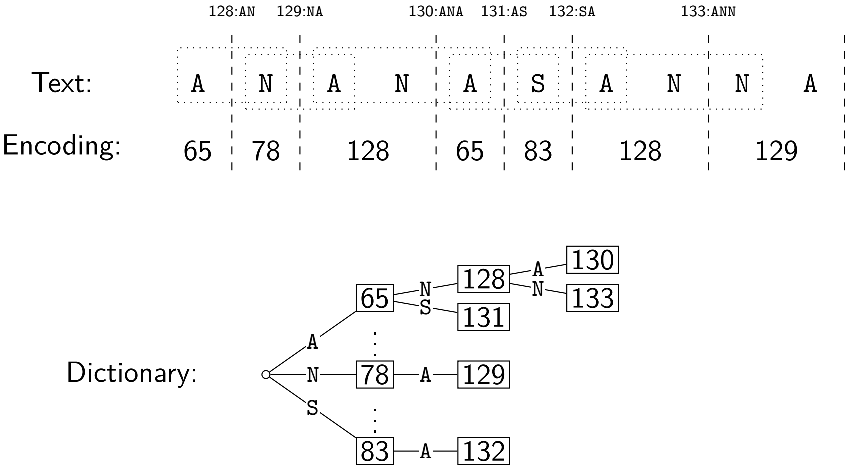


Dictionary:
A
N
S
65
78
83
N
S
A
A
128
131
129
132
A
130

LZW Example


128:AN 129:NA 130:ANA 131:AS 132:SA 133:ANN
Text: A N A N A S A N N A
Encoding: 65 78 128 65 83 128
Dictionary:
A N S
65 78 83
N S A A
128 131 129 132
A N
130 133

LZW Example


128:AN
129:NA
130:ANA
131:AS
132:SA
133:ANN
Text:
A N A N A S A N N A
Encoding:
65 78 128 65 83 128 129
Dictionary:
A
N
S
65
78
83
N
S
A
A
128
131
129
132
A
N
130
133

LZW encoding pseudocode

```
LZW::encoding(S, C)
S : input-stream of characters, C: output-stream of integers
Initialize dictionary D with ASCII in a trie
idx ← 128
while S is non-empty do
    z ← root of trie D            // find longest string
    while (S is non-empty and z has a child c labelled S.top())
        z ← c; S.pop()
    C.append(code-number stored at z)

    if S is non-empty             // add to dictionary
        create child of z labelled S.top() with code-number idx
        idx++
```

Run-time: $O(|S|)$, assuming we can look up child in $O(1)$ time.

LZW decoding

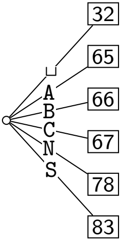
- Build dictionary while reading string by imitating encoder.
- We are one step behind.

67 65 78 32 66 129 133 83

- Example:

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:



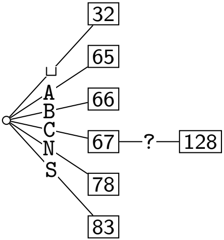
LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

67	65	78	32	66	129	133	83
C							

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:



LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

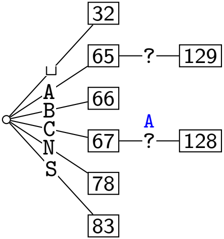
  67 65 78 32 66 129 133 83

- Example: C A

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

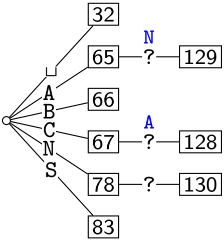
67	65	78	32	66	129	133	83
C	A	N					

ASCII

...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding

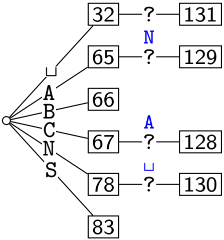
- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

67	65	78	32	66	129	133	83
C	A	N	␣				

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding

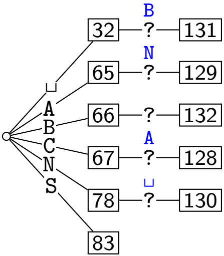
- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

67	65	78	32	66	129	133	83
C	A	N	␣	B			

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding

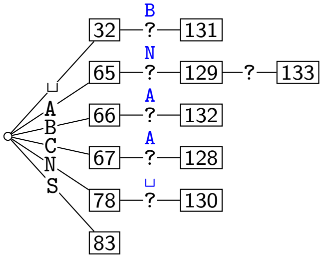
- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

67	65	78	32	66	129	133	83
C	A	N	␣	B	AN		

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding

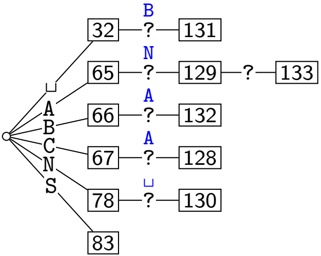
- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

	67	65	78	32	66	129	133	83
Example:	C	A	N	␣	B	AN	???	

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding pseudocode – incomplete

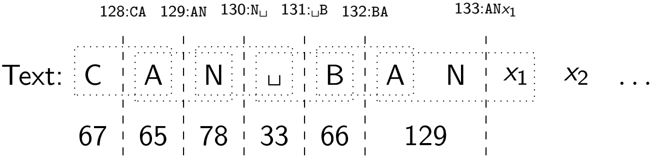
```
LZW::decoding(C, S)
C: input-stream of integers, S: output-stream
D ← dictionary that maps {0, ..., 127} to ASCII
idx ← 128
k ← C.pop(); s ← D.search(k); S.append(s)
while there are more codes in C do
    s_prev ← s; k ← C.pop()
    if k < idx do s ← D.search(k)
    else ???

    append each character of s to S
    D.insert(idx, s_prev ⁀ s[0])
    idx++
```

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!

LZW decoding: the catch

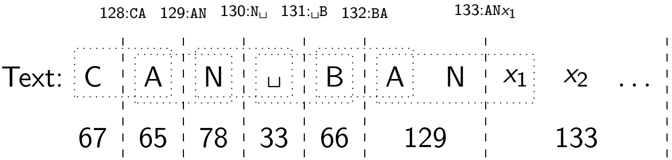
- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?



(for some unknown characters $x_1, x_2, \ldots$)

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?

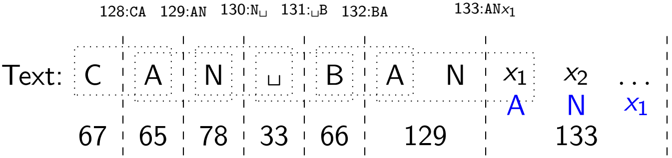


(for some unknown characters $x_1, x_2, \ldots$)

- We know: 133 encodes ANx_1

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?

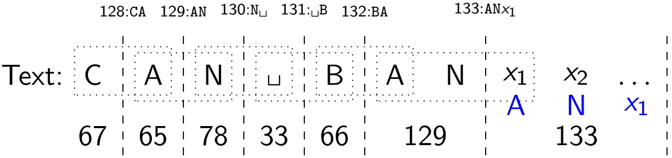


(for some unknown characters $x_1, x_2, \ldots$)

- We know: 133 encodes ANx_1
- We know: Next step uses $133 =$ ANx_1

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?

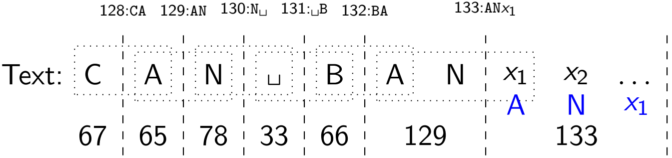


(for some unknown characters $x_1, x_2, \ldots$)

- We know: 133 encodes ANx_1
- We know: Next step uses 133 $=$ ANx_1
- So x_1 =A and 133 encodes ANA

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?



(for some unknown characters $x_1, x_2, \ldots$)

- We know: 133 encodes ANx_1
- We know: Next step uses 133 = ANx_1
- So x_1 =A and 133 encodes ANA

Generally: If code number is about to be added to D, then it encodes

"previous string $\frown$ first character of previous string"

LZW decoding pseudocode

LZW::decoding(C, S)

C: input-stream of integers, S: output-stream

```
D ← dictionary that maps {0, ..., 127} to ASCII
idx ← 128
k ← C.pop(); s ← D.search(k); S.append(s)
while there are more codes in C do
    s_prev ← s; k ← C.pop()
    if k < idx do s ← D.search(k)
    else if k = idx do s ← s_prev ⁀ s_prev[0]   // special situation
    else return FAIL                             // invalid encoding!
    append each character of s to S
    D.insert(idx, s_prev ⁀ s[0])
    idx++
```

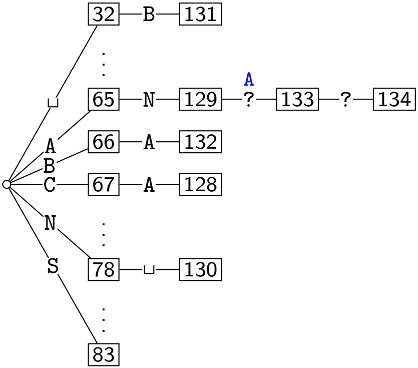
LZW decoding example revisited

67	65	78	32	66	129	133
C	A	N	␣	B	AN	ANA

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



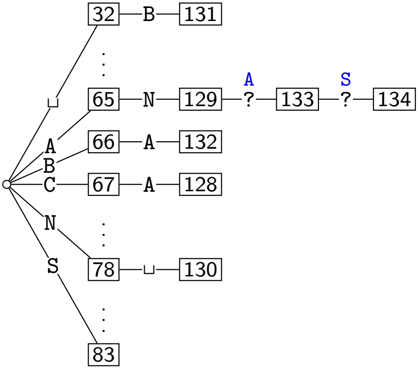
LZW decoding example revisited

67	65	78	32	66	129	133	83
C	A	N	␣	B	AN	ANA	S

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



LZW decoding - second example

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a					

- No need to build a trie; store dictionary D as array.

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary *D* continued	
input	**decodes to**	**Code #**	**String**
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
		133	
		134	
		135	
		136	
		137	

LZW decoding - second example

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a	bar				

- No need to build a trie; store dictionary D as array.

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary D continued	
input	**decodes to**	**Code #**	**String**
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
		134	
		135	
		136	
		137	

LZW decoding - second example

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a	bar	barb			

- No need to build a trie; store dictionary *D* as array.

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary *D* continued	
input	**decodes to**	**Code #**	**String**
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
		135	
		136	
		137	

LZW decoding - second example

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a	bar	barb	ar		

- No need to build a trie; store dictionary D as array.

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary D continued	
input	**decodes to**	**Code #**	**String**
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
129	ar	135	barba
		136	
		137	

LZW decoding - second example

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a	bar	barb	ar	e	

- No need to build a trie; store dictionary D as array.

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary *D* continued	
input	**decodes to**	**Code #**	**String**
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
129	ar	135	barba
101	e	136	are
		137	

LZW decoding - second example

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a	bar	barb	ar	e	n

- No need to build a trie; store dictionary D as array.

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary D continued	
input	**decodes to**	**Code #**	**String**
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
129	ar	135	barba
101	e	136	are
110	n	137	en

Lempel-Ziv-Welch decoding details

- Dictionary D maps consecutive integers to words. Use an array!
- Run-time: $O(|S|)$.
  - $\Theta(|s|)$ each round to append s to output.
  - Everything else takes no longer
- Dictionary wastes space: words may get long
  - To save space, store string as code of prefix + one character.
  - Can still look up s in $O(|s|)$ time.
  - So run-time remains $O(|S|)$

LZW::dictionary-lookup(D, k)
1. $s \leftarrow$ empty word
2. **while** $k > 127$ **do** $(k, c) \leftarrow D[k]$, $s.$*prepend*(c)
3. $s.$*prepend*$(D[k])$

LZW decoding - second example revisited

98	97	114	128	114	97	131	134	129	101	110
b	a	r	ba	r	a	bar	barb	ar	e	n

D (**ASCII**)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary *D* continued		
input	**decodes to**	**Code #**	**String (human)**	**String (computer)**
98	b			
97	a	128	ba	98, a
114	r	129	ar	97, r
128	ba	130	rb	114, b
114	r	131	bar	128, r
97	a	132	ra	114, a
131	bar	133	ab	97, b
134	barb	134	barb	131, b
129	ar	135	barba	134, a
101	e	136	are	129, e
110	n	137	en	101, n

Lempel-Ziv-Welch discussion

- Encoding and decoding take $O(|S|)$ time (assuming constant alphabet)
- Encoding and decoding need to go through the string only *once*
  $\Rightarrow$ can do compression while streaming the text
- So far, we encoded with numbers. How to convert to bitstring?
  Use width-12 **fixed-width encoding**, e.g. $129 = 000010000001$.
  (Stop adding to D once we are at code-number $2^{12}-1 = 4095$.)
- Compresses quite well ($\approx 45\%$ on English text).

Brief history:

LZ77 Original version ("sliding window")
Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, ...
DEFLATE used in (pk)`zip`, `gzip`, `PNG`

LZ78 Second (slightly improved) version
Derivatives: LZW, LZMW, LZAP, LZY, ...
LZW used in `compress`, `GIF` (patent issues!)

Outline

bzip2 overview

Idea: Combine multiple compression schemes!

Example: `bzip2`. Key ingredient is to us *text transforms*: Change input into a different text that compresses better.

T_0

↓ Burrows-Wheeler transform

T_1 is a permutation. If T_0 has repeated substrings, then T_1 has *long runs of repeated characters*.

T_1

↓ Move-to-front transform

T_2 uses ASCII-numbers. If T_1 has long runs of characters, then T_2 has *long runs of 0* and *skewed frequencies*.

T_2

↓ 0-runs encoding

If T_2 has long runs of zeroes, then T_3 is *shorter*. Skewed frequencies remain.

T_3

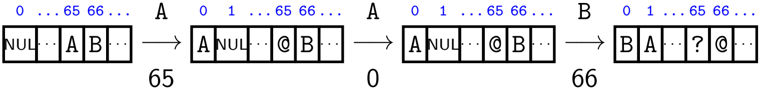
↓ Huffman encoding

Compresses well since frequencies are skewed.

T_4

bzip2 — the easy steps

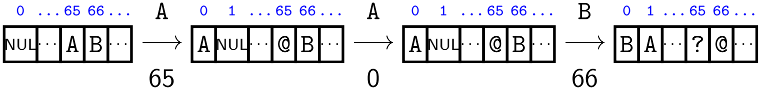
- Move-to-front transform:
  - Dictionary $D : \{0, .., 127\} \to ASCII$ is unordered array with MTF.
  - Character c gets mapped to index i with $D[i] = c$



  - A character in S repeats k times $\Leftrightarrow$ C has run of $k-1$ zeroes
  - We would expect lots of small numbers in the output.

bzip2 — the easy steps

- Move-to-front transform:
  - Dictionary $D : \{0, .., 127\} \rightarrow ASCII$ is unordered array with MTF.
  - Character c gets mapped to index i with $D[i] = c$



  - A character in S repeats k times $\Leftrightarrow$ C has run of $k-1$ zeroes
  - We would expect lots of small numbers in the output.

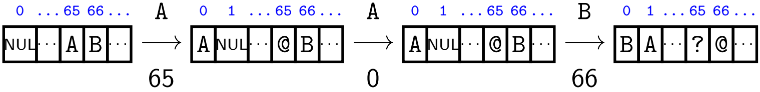
- 0-runs encoding:
  - Input consists of 'characters' in $\{0, \ldots, 127\}$ with long runs of zeroes
  - Idea: replace k consecutive zeroes by $(k)_2$ ($\approx \log k$ bits) using two new characters A, B.

    (We actually use "bijective binary encoding" to save a bit.)

    '65' '0' '0' '0' '0' '67' '0' '0' '72' becomes '65' A B '67' B '72'

bzip2 — the easy steps

- Move-to-front transform:
  - Dictionary $D : \{0, .., 127\} \rightarrow ASCII$ is unordered array with MTF.
  - Character c gets mapped to index i with $D[i] = c$



  - A character in S repeats k times $\Leftrightarrow$ C has run of $k-1$ zeroes
  - We would expect lots of small numbers in the output.

- 0-runs encoding:
  - Input consists of 'characters' in $\{0, \ldots, 127\}$ with long runs of zeroes
  - Idea: replace k consecutive zeroes by $(k)_2$ ($\approx \log k$ bits) using two new characters A, B.

    (We actually use "bijective binary encoding" to save a bit.)

    '65' '0' '0' '0' '0' '67' '0' '0' '72' becomes '65' A B '67' B '72'

- Huffman encoding: exactly as seen before.

Outline

10. Data Compression
    - Background
    - Single-Character Encodings
    - Huffman's Encoding Trie
    - Lempel-Ziv-Welch
    - Combining Compression Schemes: bzip2
    - Burrows-Wheeler Transform

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

S:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.

- ***i*th cyclic shift**: move i characters from front to back.
- We treat (exceptionally) end-sentinel $ like any other character

start-index	
0	alf␣eats␣alfalfa$
1	lf␣eats␣alfalfa$a
2	f␣eats␣alfalfa$al
3	␣eats␣alfalfa$alf
4	eats␣alfalfa$alf␣
5	ats␣alfalfa$alf␣e
6	ts␣alfalfa$alf␣ea
7	s␣alfalfa$alf␣eat
8	␣alfalfa$alf␣eats
9	alfalfa$alf␣eats␣
10	lfalfa$alf␣eats␣a
11	falfa$alf␣eats␣al
12	alfa$alf␣eats␣alf
13	lfa$alf␣eats␣alfa
14	fa$alf␣eats␣alfal
15	a$alf␣eats␣alfalf
16	$alf␣eats␣alfalfa

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.

- ***i*th cyclic shift**: move i characters from front to back.
- We treat (exceptionally) end-sentinel $ like any other character

(We do not need to write cyclic shifts explicitly; they can be read via start-index from S.)

start-index	
0	alf␣eats␣alfalfa$
1	lf␣eats␣alfalfa$a
2	f␣eats␣alfalfa$al
3	␣eats␣alfalfa$alf
4	eats␣alfalfa$alf␣
5	ats␣alfalfa$alf␣e
6	ts␣alfalfa$alf␣ea
7	s␣alfalfa$alf␣eat
8	␣alfalfa$alf␣eats
9	alfalfa$alf␣eats␣
10	lfalfa$alf␣eats␣a
11	falfa$alf␣eats␣al
12	alfa$alf␣eats␣alf
13	lfa$alf␣eats␣alfa
14	fa$alf␣eats␣alfal
15	a$alf␣eats␣alfalf
16	$alf␣eats␣alfalfa

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.

- ***i*th cyclic shift**: move i characters from front to back.
- We treat (exceptionally) end-sentinel $ like any other character

(We do not need to write cyclic shifts explicitly; they can be read via start-index from S.)

Observe: Every column contains a permutation of S.

start-index	
0	alf␣eats␣alfalfa$
1	lf␣eats␣alfalfa$a
2	f␣eats␣alfalfa$al
3	␣eats␣alfalfa$alf
4	eats␣alfalfa$alf␣
5	ats␣alfalfa$alf␣e
6	ts␣alfalfa$alf␣ea
7	s␣alfalfa$alf␣eat
8	␣alfalfa$alf␣eats
9	alfalfa$alf␣eats␣
10	lfalfa$alf␣eats␣a
11	falfa$alf␣eats␣al
12	alfa$alf␣eats␣alf
13	lfa$alf␣eats␣alfa
14	fa$alf␣eats␣alfal
15	a$alf␣eats␣alfalf
16	$alf␣eats␣alfalfa

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.
Step 2: Sort lexicographically.

- Use MSD-radix-sort.
- $\Theta(n)$ strings of length $\Theta(n)$
  $\Rightarrow \Theta(n^2)$ worst-case time
- But usually much faster.

start-index	
16	$alf␣eats␣alfalfa
8	␣alfalfa$alf␣eats
3	␣eats␣alfalfa$alf
15	a$alf␣eats␣alfalf
0	alf␣eats␣alfalfa$
12	alfa$alf␣eats␣alf
9	alfalfa$alf␣eats␣
5	ats␣alfalfa$alf␣e
4	eats␣alfalfa$alf␣
2	f␣eats␣alfalfa$al
14	fa$alf␣eats␣alfal
11	falfa$alf␣eats␣al
1	lf␣eats␣alfalfa$a
13	lfa$alf␣eats␣alfa
10	lfalfa$alf␣eats␣a
7	s␣alfalfa$alf␣eat
6	ts␣alfalfa$alf␣ea

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.
Step 2: Sort lexicographically.

- Use MSD-radix-sort.
- $\Theta(n)$ strings of length $\Theta(n)$ $\Rightarrow \Theta(n^2)$ worst-case time
- But usually much faster.

Observe: Every column continues to contain a permutation of S.

start-index	
16	$alf␣eats␣alfalfa
8	␣alfalfa$alf␣eats
3	␣eats␣alfalfa$alf
15	a$alf␣eats␣alfalf
0	alf␣eats␣alfalfa$
12	alfa$alf␣eats␣alf
9	alfalfa$alf␣eats␣
5	ats␣alfalfa$alf␣e
4	eats␣alfalfa$alf␣
2	f␣eats␣alfalfa$al
14	fa$alf␣eats␣alfal
11	falfa$alf␣eats␣al
1	lf␣eats␣alfalfa$a
13	lfa$alf␣eats␣alfa
10	lfalfa$alf␣eats␣a
7	s␣alfalfa$alf␣eat
6	ts␣alfalfa$alf␣ea

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.
Step 2: Sort lexicographically.
Step 3: Extract rightmost column.

The **Burrows-Wheeler transform** consists of the last characters of the cyclic shifts of S after sorting them lexicographically.

start-index	
16	$alf␣eats␣alfalfa
8	␣alfalfa$alf␣eats
3	␣eats␣alfalfa$alf
15	a$alf␣eats␣alfalf
0	alf␣eats␣alfalfa$
12	alfa$alf␣eats␣alf
9	alfalfa$alf␣eats␣
5	ats␣alfalfa$alf␣e
4	eats␣alfalfa$alf␣
2	f␣eats␣alfalfa$al
14	fa$alf␣eats␣alfal
11	falfa$alf␣eats␣al
1	lf␣eats␣alfalfa$a
13	lfa$alf␣eats␣alfa
10	lfalfa$alf␣eats␣a
7	s␣alfalfa$alf␣eat
6	ts␣alfalfa$alf␣ea

Burrows-Wheeler Transform

Given: Source text S as an *array* with end-sentinel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	$

Step 1: Write down cyclic shifts.
Step 2: Sort lexicographically.
Step 3: Extract rightmost column.

The **Burrows-Wheeler transform** consists of the last characters of the cyclic shifts of S after sorting them lexicographically.

Observe: C is a permutation of S.

Observe: Substring `alf` occurs three times in S and causes runs `lll` and `aaa` in C (why?)

start-index	
16	$alf␣eats␣alfalfa
8	␣alfalfa$alf␣eats
3	␣eats␣alfalfa$alf
15	a$alf␣eats␣alfalf
0	alf␣eats␣alfalfa$
12	alfa$alf␣eats␣alf
9	alfalfa$alf␣eats␣
5	ats␣alfalfa$alf␣e
4	eats␣alfalfa$alf␣
2	f␣eats␣alfalfa$al
14	fa$alf␣eats␣alfal
11	falfa$alf␣eats␣al
1	lf␣eats␣alfalfa$a
13	lfa$alf␣eats␣alfa
10	lfalfa$alf␣eats␣a
7	s␣alfalfa$alf␣eat
6	ts␣alfalfa$alf␣ea

Fast Burrows-Wheeler Encoding

$S =$ `alf␣eats␣alfalfa$`

	i	ith cyclic shift
0	16	$alf␣eats␣alfalfa
1	8	␣alfalfa$alf␣eats
2	3	␣eats␣alfalfa$alf
3	15	a$alf␣eats␣alfalf
4	0	alf␣eats␣alfalfa$
5	12	alfa$alf␣eats␣alf
6	9	alfalfa$alf␣eats␣
7	5	ats␣alfalfa$alf␣e
8	4	eats␣alfalfa$alf␣
9	2	f␣eats␣alfalfa$al
10	14	fa$alf␣eats␣alfal
11	11	falfa$alf␣eats␣al
12	1	lf␣eats␣alfalfa$a
13	13	lfa$alf␣eats␣alfa
14	10	lfalfa$alf␣eats␣a
15	7	s␣alfalfa$alf␣eat
16	6	ts␣alfalfa$alf␣ea

	A^s	corresponding suffix
0	16	$alf␣eats␣alfalf a
1	8	␣alfalfa$alf␣eat s
2	3	␣eats␣alfalfa$al f
3	15	a$alf␣eats␣alfal f
4	0	alf␣eats␣alfalfa $
5	12	alfa$alf␣eats␣al f
6	9	alfalfa$alf␣eats ␣
7	5	ats␣alfalfa$alf␣ e
8	4	eats␣alfalfa$alf ␣
9	2	f␣eats␣alfalfa$a l
10	14	fa$alf␣eats␣alfa l
11	11	falfa$alf␣eats␣a l
12	1	lf␣eats␣alfalfa$ a
13	13	lfa$alf␣eats␣alf a
14	10	lfalfa$alf␣eats␣ a
15	7	s␣alfalfa$alf␣ea t
16	6	ts␣alfalfa$alf␣e a

- **Observe:** Same sorting permutation for cyclic shifts and suffixes.

Fast Burrows-Wheeler Encoding

$S =$ `alf␣eats␣alfalfa$`

	i	ith cyclic shift
0	16	`$alf␣eats␣alfalfa`
1	8	`␣alfalfa$alf␣eats`
2	3	`␣eats␣alfalfa$alf`
3	15	`a$alf␣eats␣alfalf`
4	0	`alf␣eats␣alfalfa$`
5	12	`alfa$alf␣eats␣alf`
6	9	`alfalfa$alf␣eats␣`
7	5	`ats␣alfalfa$alf␣e`
8	4	`eats␣alfalfa$alf␣`
9	2	`f␣eats␣alfalfa$al`
10	14	`fa$alf␣eats␣alfal`
11	11	`falfa$alf␣eats␣al`
12	1	`lf␣eats␣alfalfa$a`
13	13	`lfa$alf␣eats␣alfa`
14	10	`lfalfa$alf␣eats␣a`
15	7	`s␣alfalfa$alf␣eat`
16	6	`ts␣alfalfa$alf␣ea`

	A^s	corresponding suffix
0	16	`$alf␣eats␣alfalf a`
1	8	`␣alfalfa$alf␣eat s`
2	3	`␣eats␣alfalfa$al f`
3	15	`a$alf␣eats␣alfal f`
4	0	`alf␣eats␣alfalfa $`
5	12	`alfa$alf␣eats␣al f`
6	9	`alfalfa$alf␣eats ␣`
7	5	`ats␣alfalfa$alf␣ e`
8	4	`eats␣alfalfa$alf ␣`
9	2	`f␣eats␣alfalfa$a l`
10	14	`fa$alf␣eats␣alfa l`
11	11	`falfa$alf␣eats␣a l`
12	1	`lf␣eats␣alfalfa$ a`
13	13	`lfa$alf␣eats␣alf a`
14	10	`lfalfa$alf␣eats␣ a`
15	7	`s␣alfalfa$alf␣ea t`
16	6	`ts␣alfalfa$alf␣e a`

- **Observe:** Same sorting permutation for cyclic shifts and suffixes.
- That's the suffix array A^s! Can compute this in $O(n \log n)$ time.
- Read BWT encoding from it: $C[i] = \begin{cases} S\left[A^s[i]-1\right] & \text{if } A^s[i] > 0 \\ \$ & \text{otherwise} \end{cases}$

BWT Decoding

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

We can reconstruct the *first* and *last column* of the matrix of cyclic shifts.

1. Last column: C
2. First column: C sorted
   - It was a permutation of S.
   - It was in sorted order.
   - C was also a permutation of S.

```
...........a
...........r
...........d
...........$
...........r
...........c
...........a
...........a
...........a
...........a
...........b
...........b
```

BWT Decoding

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

We can reconstruct the *first* and *last column* of the matrix of cyclic shifts.

1. Last column: C
2. First column: C sorted
   - It was a permutation of S.
   - It was in sorted order.
   - C was also a permutation of S.
3. What was the first character of S?

```
$.........a
a.........r
a.........d
a.........$
a.........r
a.........c
b.........a
b.........a
c.........a
d.........a
r.........b
r.........b
```

BWT Decoding

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

We can reconstruct the *first* and *last column* of the matrix of cyclic shifts.

1. Last column: C
2. First column: C sorted
   - It was a permutation of S.
   - It was in sorted order.
   - C was also a permutation of S.
3. What was the first character of S?
4. Presume you know which a of C this is: What is the next character of S?

Question: Which character on the left corresponds to which character on the right?

```
$.........a
a.........r
a.........d
a.........$
a.........r
a.........c
b.........a
b.........a
c.........a
d.........a
r.........b
r.........b
```

BWT Decoding - Disambiguation

Idea: Attach row-index at each character of C.

- Two cyclic shifts end in `b`, call them w_{10}`(b,10)` and w_{11}`(b,11)`
- *Without* knowing w_{10}, w_{11}, we know w_{10}`(b,10)` $<_{\text{lex}}$ w_{11}`(b,11)` (why?)

```
$.........(a,0)
a.........(r,1)
a.........(d,2)
a.........($,3)
a.........(r,4)
a.........(c,5)
b.........(a,6)
b.........(a,7)
c.........(a,8)
d.........(a,9)
r.........(b,10)
r.........(b,11)
```

BWT Decoding - Disambiguation

Idea: Attach row-index at each character of C.

- Two cyclic shifts end in `b`, call them w_{10}`(b,10)` and w_{11}`(b,11)`
- *Without* knowing w_{10}, w_{11}, we know w_{10}`(b,10)` $<_{\text{lex}}$ w_{11}`(b,11)` (why?)
- Therefore $w_{10} <_{\text{lex}} w_{11}$
- Two cyclic shifts start with `b`: `(b,10)`w_{10} and `(b,11)`w_{11}.

```
$......... (a,0)
a......... (r,1)
a......... (d,2)
a......... ($,3)
a......... (r,4)
a......... (c,5)
b......... (a,6)
b......... (a,7)
c......... (a,8)
d......... (a,9)
r.........(b,10)
r.........(b,11)
```

BWT Decoding - Disambiguation

Idea: Attach row-index at each character of C.

- Two cyclic shifts end in `b`, call them w_{10}(b,10) and w_{11}(b,11)
- *Without* knowing w_{10}, w_{11}, we know w_{10}(b,10) $<_{\text{lex}}$ w_{11}(b,11) (why?)
- Therefore $w_{10} <_{\text{lex}} w_{11}$
- Two cyclic shifts start with `b`: (b,10)w_{10} and (b,11)w_{11}.
- We know (b,10)w_{10} $<_{\text{lex}}$ (b,11)w_{11}.

```
$.........(a,0)
a.........(r,1)
a.........(d,2)
a.........($,3)
a.........(r,4)
a.........(c,5)
b.........(a,6)
b.........(a,7)
c.........(a,8)
d.........(a,9)
r.........(b,10)
r.........(b,11)
```

BWT Decoding - Disambiguation

Idea: Attach row-index at each character of C.

- Two cyclic shifts end in `b`, call them w_{10}`(b,10)` and w_{11}`(b,11)`
- *Without* knowing w_{10}, w_{11}, we know w_{10}`(b,10)` $<_{\text{lex}}$ w_{11}`(b,11)` (why?)
- Therefore $w_{10} <_{\text{lex}} w_{11}$
- Two cyclic shifts start with `b`: `(b,10)`w_{10} and `(b,11)`w_{11}.
- We know `(b,10)`w_{10} $<_{\text{lex}}$ $(\mathrm{b}, 11)w_{11}$.
- Therefore `(b,10)` comes *before* `(b,11)` in first column.

```
$.........(a,0)
a.........(r,1)
a.........(d,2)
a.........($,3)
a.........(r,4)
a.........(c,5)
b.........(a,6)
b.........(a,7)
c.........(a,8)
d.........(a,9)
r.........(b,10)
r.........(b,11)
```

Result: Equal characters are in the same order in the first and last column.

BWT Decoding Continued

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*

(\$,3)	(a,0)
(a,0)	(r,1)
(a,6)	(d,2)
(a,7)	(\$,3)
(a,8)	(r,4)
(a,9)	(c,5)
(b,10)	(a,6)
(b,11)	(a,7)
(c,5)	(a,8)
(d,2)	(a,9)
(r,1)	(b,10)
(r,4)	(b,11)

BWT Decoding Continued

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*
3. Find index j of end-sentinel \$ in C

(\$,3)	(a,0)
(a,0)	(r,1)
(a,6)	(d,2)
(a,7)	(\$,3)
(a,8)	(r,4)
(a,9)	(c,5)
(b,10)	(a,6)
(b,11)	(a,7)
(c,5)	(a,8)
(d,2)	(a,9)
(r,1)	(b,10)
(r,4)	(b,11)

BWT Decoding Continued

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*
3. Find index j of end-sentinel \$ in C
4. Starting from $A[j]$, recover S by looking up next character and next index.

S = a

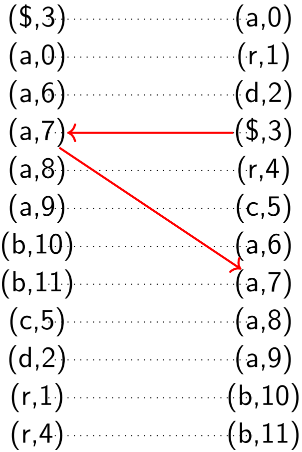
($,3)	(a,0)
(a,0)	(r,1)
(a,6)	(d,2)
(a,7) ←	($,3)
(a,8)	(r,4)
(a,9)	(c,5)
(b,10)	(a,6)
(b,11)	(a,7)
(c,5)	(a,8)
(d,2)	(a,9)
(r,1)	(b,10)
(r,4)	(b,11)

BWT Decoding Continued

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*
3. Find index j of end-sentinel \$ in C
4. Starting from $A[j]$, recover S by looking up next character and next index.

S = a

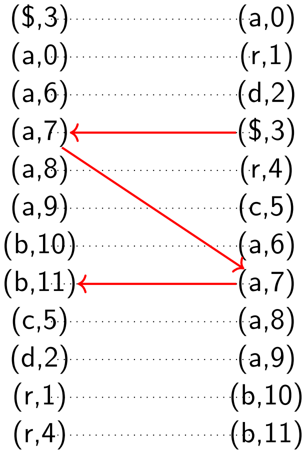


BWT Decoding Continued

Given: String $C = \texttt{ard\$rcaaaabb}$ (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*
3. Find index j of end-sentinel \$ in C
4. Starting from $A[j]$, recover S by looking up next character and next index.

S = ab

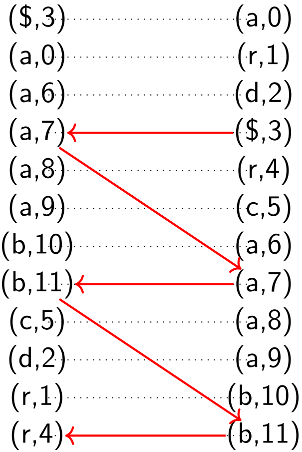


BWT Decoding Continued

Given: String $C =$ `ard$rcaaaabb` (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*
3. Find index j of end-sentinel \$ in C
4. Starting from $A[j]$, recover S by looking up next character and next index.

S = `abr`

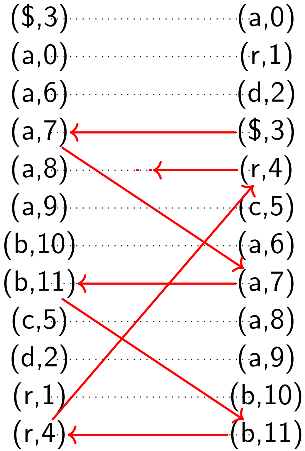


BWT Decoding Continued

Given: String C = `ard$rcaaaabb` (obtained from a BWT encoding).

1. Last column: C
   - Disambiguate by row-index.
2. First column A: C sorted *stably*
3. Find index j of end-sentinel \$ in C
4. Starting from $A[j]$, recover S by looking up next character and next index.

S = `abracadabra$`



BWT Decoding

BWT::decoding(C, S)
C : string of characters, one of which (not necessarily last one) is $
S: output-stream

```
Initialize array A          // leftmost column
for all indices i of C
        A[i] ← (C[i], i)    // store character and index
Stably sort A by first aspect

for all indices j of C      // where is the $-char?
        if C[j] = $ break

do                          // extract source text
        S.append(character stored in A[j])
        j ← index stored in A[j]
while appended character is not $
```

What sorting-algorithm would you use?

BWT and bzip2 Discussion

BWT encoding cost: $O(n \log n)$

- Read encoding from the suffix array.

BWT decoding cost: $O(n + |\Sigma_S|)$ (faster than encoding)

Encoding and decoding both use $O(n)$ space.

They need *all* of the text (no streaming possible). BWT (hence bzip2) is a **block compression method** that compresses one block at a time.

bzip2 **encoding cost**: $O(n(\log n + |\Sigma|))$ with a big constant.
bzip2 **decoding cost**: $O(n|\Sigma|)$ with a big constant.

bzip2 tends to be slower than other methods, but gives better compression.

Compression summary

Huffman	**Lempel-Ziv-Welch**	**bzip2** (uses Burrows-Wheeler)
variable-length	fixed-width	multi-step
single-character	multi-character	multi-step
2-pass, must send dictionary	1-pass	not streamable
optimal 01-prefix-code	good on English text	better on English text
requires uneven frequencies	requires repeated substrings	requires repeated substrings
rarely used directly	frequently used	used but slow
part of pkzip, JPEG, MP3	GIF, some variants of PDF, compress	bzip2 and variants

Modern compression techniques (using Markov chains and probabilistic modelling) compress even better, but are beyond the scope of the course.