CS 240 – Data Structures and Data Management

Module 10: Compression

A. Storjohann

Based on lecture notes by many previous cs240 instructors

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

Fall 2018

References: Goodrich & Tamassia 9.3

version 2018-11-15 14:11

Outline

1. Compression
   - Encoding Basics
   - Huffman Codes
   - Run-Length Encoding
   - Lempel-Ziv-Welch
   - bzip2
   - Burrows-Wheeler Transform

Outline

Data Storage and Transmission

The problem: How to store and transmit data?

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 *coded alphabet* Σ_C
Encoding	An algorithm mapping source texts to coded texts
Decoding	An algorithm mapping coded texts back to their original source text

Note: Source "text" can be any sort of data (not always text!)

Usually the coded alphabet Σ_C is just binary: $\{0, 1\}$.

Judging Encoding Schemes

We can always measure efficiency of encoding/decoding algorithms.

What other goals might there be?

- Processing speed
- Reliability (e.g. error-correcting codes)
- Security (e.g. encryption)
- **Size**

Encoding schemes that try to minimize the size of the coded text perform *data compression*. We will measure the *compression ratio*:

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

Types of Data Compression

Logical vs. Physical

- **Logical Compression** uses the meaning of the data and only applies to a certain domain (e.g. sound recordings)
- **Physical Compression** only knows the physical bits in the data, not the meaning behind them

Lossy vs. Lossless

- **Lossy Compression** achieves better compression ratios, but the decoding is approximate; the exact source text S is not recoverable
- **Lossless Compression** always decodes S exactly

For media files, lossy, logical compression is useful (e.g. JPEG, MPEG)

We will concentrate on *physical, lossless* compression algorithms.
These techniques can safely be used for any application.

Character Encodings

Definition: Map each character from the source alphabet to a string in coded alphabet.

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

- For $c \in \Sigma_S$, we call $E(c)$ the *codeword* of c
- **Fixed-length code**: All codewords have the same length.
- Example: ASCII (American Standard Code for Information Interchange), 1963:

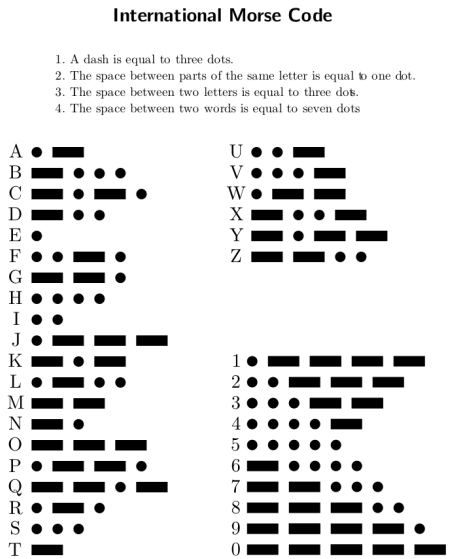
char	null	start of heading	start of text	end of text	...	0	1	...	A	B	...	~	delete
code	0	1	2	3	...	48	49	...	65	66	...	126	127

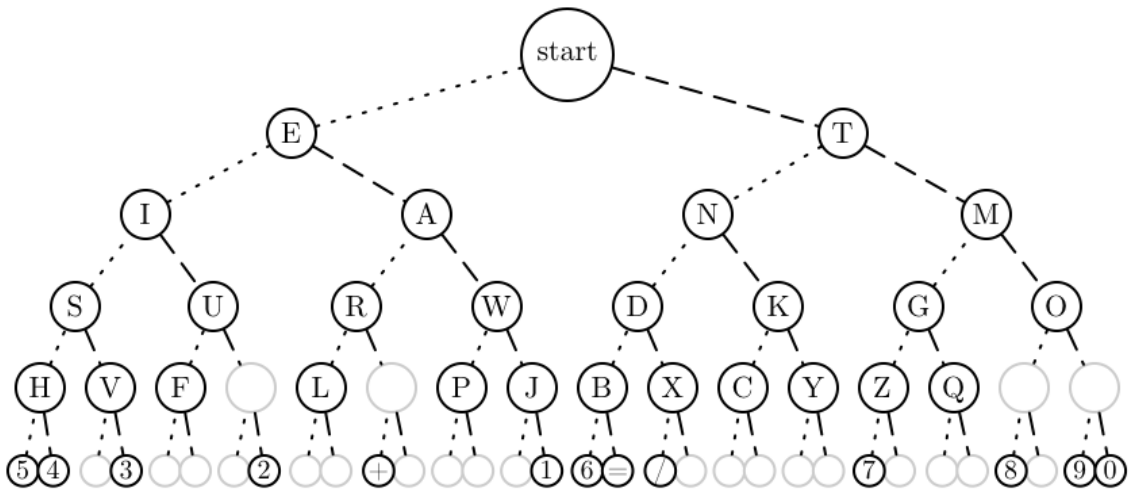
  - 7 bits to encode 128 possible characters:
    "control codes", spaces, letters, digits, punctuation
  - Not well-suited for non-English text:
    ISO-8859 extends to 8 bits, handles most Western languages
  - To decode ASCII, we look up each 7-bit pattern in a table.
- Other (earlier) fixed-length codes: Baudot code, Murray code

Variable-Length Codes

Definition: Different codewords have different lengths

Example 1: Morse code.





Pictures taken from `http://apfelmus.nfshost.com/articles/fun-with-morse-code.html`

Example 2: UTF-8 encoding of Unicode:

- Encodes any Unicode character (more than 107,000 characters) using 1-4 bytes

Encoding

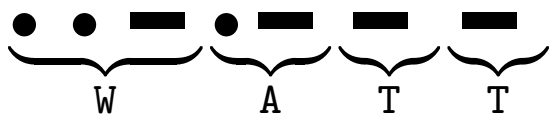
Assume we have some character encoding $E : \Sigma_S \to \Sigma_C^*$.

- Note that E is a dictionary with keys in Σ_S.
- Typically E would be stored as array indexed by Σ_S.

Encoding(E, $T[0..n-1]$)
E : the encoding dictionary, T: text with characters in Σ_S

```
1.    initialize empty string S
2.    for i = 0 ... n − 1
3.        x ← E.search(T[i])
4.        S.append(x)
5.    return S
```

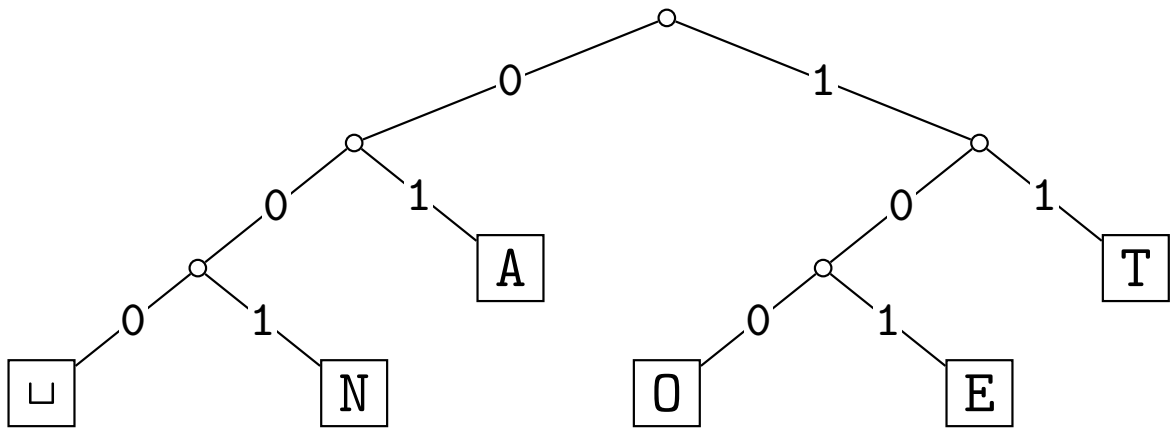
Example: encode text "`WATT`" with Morse code:



Decoding

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

- The code must be *uniquely decodable*.
  - This is false for Morse code as described!
    • ▬ ▬ • ▬ ▬ ▬ decodes to both `WATT` and `ANO`.
    (Morse code uses 'end of character' pause to avoid ambiguity.)
- From now on only consider *prefix-free* codes E:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.
- This corresponds to a *trie* with characters of Σ_S only at the leaves.



- This trie does not need the end-of-string symbol \$ since the codewords are by definition prefix-free.

Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable (why?)

```
PrefixFreeDecoding(D, C[0..n − 1])
D : the trie of a prefix-free code, C: text with characters in Σ_C
 1.    initialize empty string T
 2.    i ← 0
 3.    while i < n
 4.        r ← D.root
 5.        while r is not a leaf
 6.            if i = n return "invalid encoding"
 7.            c ← child of r that is labelled with C[i]
 8.            i ← i + 1
 9.            r ← c
10.        T.append(character stored at r)
11.    return T
```

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

Encoding from the Trie

We can also encode directly from the trie.

```
PrefixFreeEncodingFromTrie(D, S[0..n − 1])
D : the trie of a prefix-free code, S: text with characters in Σ_S
 1.   L ← array of nodes in D indexed by Σ_S
 2.   for all leaves ℓ in D
 3.        L[character at ℓ] ← ℓ
 4.   initialize empty string C
 5.   for i = 0 to n − 1
 6.        w ← empty string; v ← L[S[i]]
 7.        while v is not the root
 8.             w.prepend(character from v to its parent)
 9.        // Now w is the encoding of S[i].
10.        C.append(w)
11.   return C
```

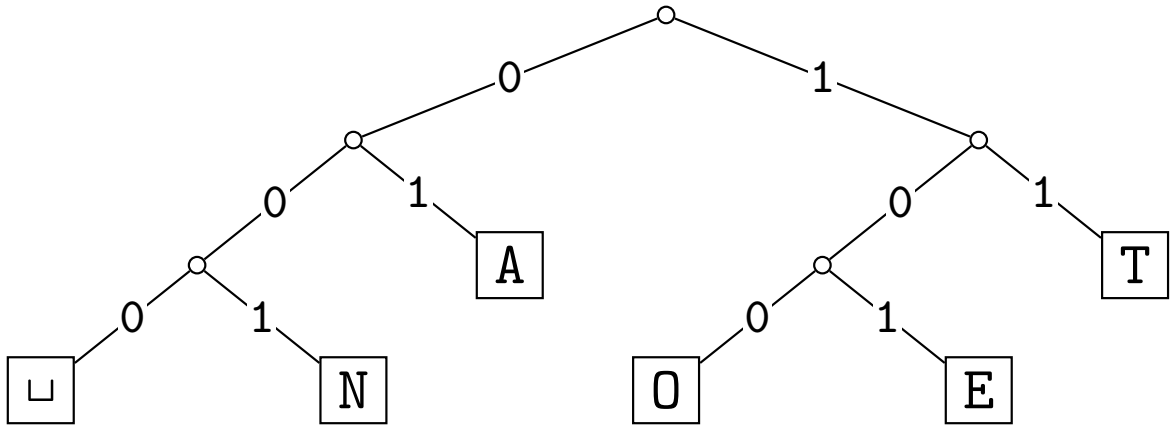
Run-time: $O(|D| + |C|) = O(|\Sigma_S| + |C|)$.

Example: Prefix-free Encoding/Decoding

Code as table:

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

Code as trie:



- Encode `AN␣ANT` $\rightarrow$ `010010000100111`
- Decode `111000001010111` $\rightarrow$ `TO␣EAT`

Outline

Character Frequency

Overall goal: Find an encoding that is short.

Observation: Some letters in Σ occur more often than others.
So let's use shorter codes for more frequent characters.

For example, the frequency of letters in typical English text is:

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%		

Huffman's Algorithm: Building the best trie

For a given source text S, how to determine the "best" trie that minimizes the length of C?

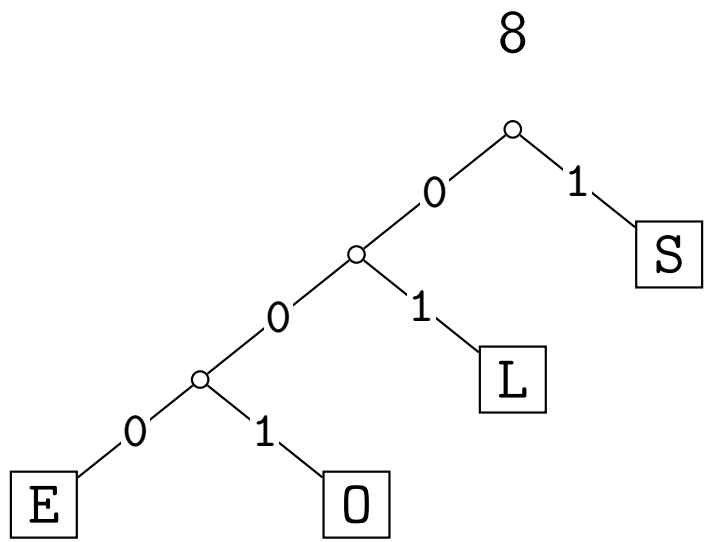
1. Determine frequency of each character $c \in \Sigma$ in S
2. For each $c \in \Sigma$, create "$\boxed{c}$" (height-0 trie holding c).
3. Assign a "weight" to each trie: sum of frequencies of all letters in trie. Initially, these are just the character frequencies.
4. Find the two tries with the minimum weight.
5. Merge these tries with new interior node; new weight is the sum. (Corresponds to adding one bit to the encoding of each character.)
6. Repeat Steps 4–5 until there is only 1 trie left; this is D.

What data structure should we store the tries in to make this efficient?

Example: Huffman tree construction

Example text: LOSSLESS, $\Sigma_S = \{L, O, S, E\}$

Character frequencies: E : 1, L : 2, O : 1, S : 4



LOSSLESS $\rightarrow$ 01001110100011

Compression ratio: $\frac{14}{8 \cdot \log 4} = \frac{14}{16} \approx 88\%$

Huffman's Algorithm: Pseudocode

Huffman-Encoding($S[0..n-1]$)
S: text over some alphabet Σ_S

```
1.   f ← array indexed by Σ_S, initially all-0
2.   for i = 0 to n − 1 do f[S[i]]++
3.   Q ← min-oriented priority queue that stores tries
4.   for all c ∈ Σ_S with f[c] > 0
5.       Q.insert(single-node trie for c with weight f[c])
6.   while Q.size() > 1
7.       T_1 ← Q.deleteMin(), T_2 ← Q.deleteMin()
8.       Q.insert(trie with T_1, T_2 as subtries and weight w(T_1)+w(T_2))
9.   D ← Q.deleteMin // decoding trie
10.  C ← PrefixFreeEncodingFromTrie(D, S)
11.  return C and D
```

Huffman Coding Evaluation

- Note: constructed trie is **not unique** (why?)
  So decoding trie must be transmitted along with the coded text C.
- This may make encoding bigger than source text!
- Encoding must pass through text twice (to compute frequencies and to encode)
- Encoding run-time: $O(n + |\Sigma_S| \log |\Sigma_S| + |C|)$
- Decoding run-time: $O(|C|)$ (this is an *asymmetric* scheme).
- The constructed trie is *optimal* in the sense that C is shortest (among all prefix-free character-encodings with $\Sigma_C = \{0, 1\}$).
  We will not go through the proof.
- Many variations (give tie-breaking rules, estimate frequencies, adaptively change encoding, ...)

Outline

Run-Length Encoding

- Variable-length code
- Example of **multi-character encoding**: multiple source-text characters receive one code-word.
- The source alphabet and coded alphabet are both binary: $\{0, 1\}$.
- Decoding dictionary is uniquely defined and not explicitly stored.

Useful if: S has long runs: $\underbrace{00000}\,\underbrace{111}\,\underbrace{0000}$

Encoding idea:

- Give the first bit of S (either 0 or 1)
- Then give a sequence of integers indicating run lengths.
- We don't have to give the bit for runs since they alternate.

Example becomes: $0, 5, 3, 4$

Question: How to encode a run length k in binary?

Prefix-free Encoding for Positive Integers

Use *Elias gamma code* to encode k:

- $\lfloor \log k \rfloor$ copies of 0, followed by
- binary representation of k (always starts with 1)

k	$\lfloor \log k \rfloor$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
⋮	⋮	⋮	⋮

RLE Encoding

```
RLE-Encoding(S[0...n−1])
S: bitstring
 1.    initialize output string C ← S[0]
 2.    i ← 0
 3.    while i < n
 4.        k ← 1
 5.        while (i + k < n and S[i + k] = S[i])
 6.            k++
 7.        for (ℓ = 1 to ⌊log k⌋)
 8.            C.append(0)
 9.        C.append(binary encoding of k)
10.        i ← i + k
11.    return C
```

RLE Decoding

```
RLE-Decoding(C)
C: stream of bits
 1.    initialize output string S
 2.    b ← C.pop() // bit-value for the current run
 3.    while C has bits left
 4.        ℓ ← 0
 5.        while C.pop() = 0   ℓ++
 6.        k ← 1
 7.        for (j = 1 to ℓ)   k ← k * 2 + C.pop()
 8.            // if C runs out of bits then encoding was invalid
 9.        for (j = 1 to k)   S.append(b)
10.        b ← 1 − b
11.    return S
```

RLE Example

Encoding:
$S = 11111110010000000000000000000011111111111$

Decoding:
$C = 0000110100100101 0$
$S = 00000000000001111011$

RLE Properties

- An all-0 string of length n would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits.
- Usually, we are not that lucky:
  - No compression until run-length $k \geq 6$
  - **Expansion** when run-length $k = 2$ or 4
- Method can be adapted to larger alphabet sizes
- Used in some image formats (e.g. TIFF)

Outline

Longer Patterns in Input

Huffman and RLE mostly take advantage of
frequent or repeated *single characters*.

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

Examples:

- 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

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

Adaptive Dictionaries

ASCII, UTF-8, and RLE 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, both encoder and decoder update D_i to D_{i+1}

Encoder and decoder must both know how the dictionary changes.

Lempel-Ziv

Lempel-Ziv is a family of *adaptive* compression algorithms.

Main Idea: Each character in the coded text C either refers to a single character in Σ_S, or a *substring* of S that both encoder and decoder have already seen.

Variants:

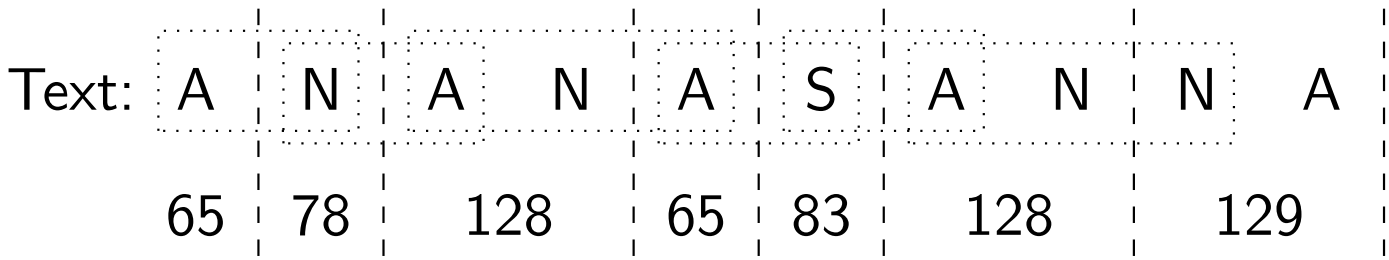
LZ77 Original version ("sliding window")
Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, ...
DEFLATE used in (`pk`)`zip`, `gzip`, `PNG`

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

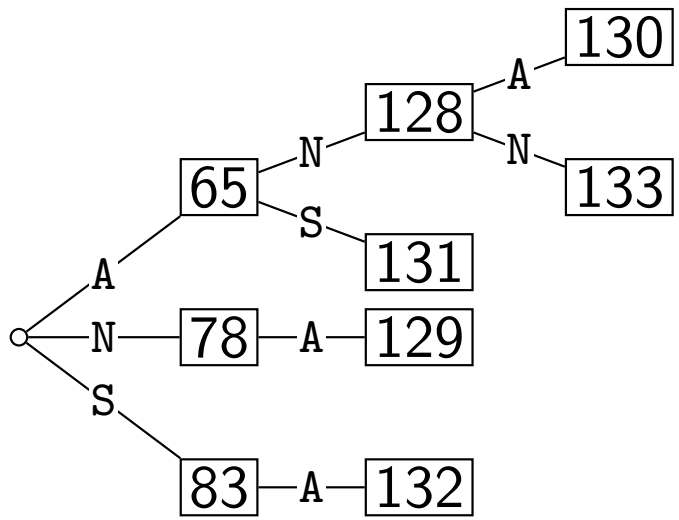
LZW Overview

- Start with dictionary D_0 for $|\Sigma_S|$.
  Usually $\Sigma_S = ASCII$, then this uses codenumbers $0, \ldots, 127$.
- Every step adds to dictionary a multi-character string, using codenumbers $128, 129, \ldots$.
- Encoding:
  - Store current dictionary D_i as a trie.
  - Parse trie to find longest prefix x already in D_i.
    So all of x can be encoded with one number.
  - Add to dictionary the *prefix that would have been useful*:
    add xK where K is the character that follows x in S.
  - This creates one child in trie at the leaf where we stopped.
- Output is a list of numbers. This is usually converted to bit-string with fixed-width encoding using 12 bits.
  - This limits the codenumbers to 4096.

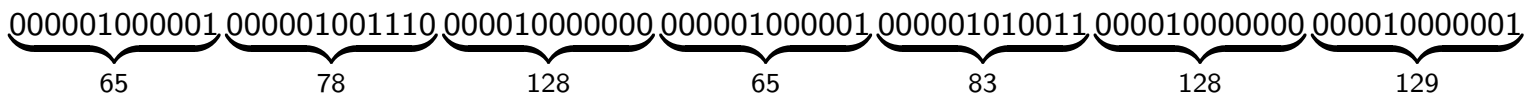
LZW Example



Dictionary:



Final output:



LZW encoding pseudocode

```
LZW-encode(S)
S : stream of characters
 1.    Initialize dictionary D with ASCII in a trie
 2.    idx ← 128
 3.    while there is input in S do {
 4.        v ← root of trie D
 5.        K ← S.peek()
 6.        while (v has a child c labelled K)
 7.            v ← c; S.pop()
 8.            if there is no more input in S break    (goto 10)
 9.            K ← S.peek()
10.        output codenumber stored at v
11.        if there is more input in S
12.            create child of v labelled K with codenumber idx
13.            idx++
14.    }
```

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: **67 65 78 32 66 129 133**

$D =$

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

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	**128**	CA	**67,** A
78	N	**129**	AN	**65,** N
32	␣	**130**	N␣	**78,** ␣
66	B	**131**	␣B	**32,** B
129	AN	**132**	BA	**66,** A
133	???	**133**		

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- Generally: decoder is "one step behind" in creating dictionary
- So problem occurs if *we want to use a code that we are about to build*.
- But then we actually know what is going on!
  - Input: code number k at the time when we are assigning k.
  - Decoder knows $s_{prev} =$ string decoded in previous step
  - Decoder wants $s =$ string k decodes to
  - We know: Encoder assigned $s_{prev} + s[0]$ to code number k
  - Also know: $s[0] =$ first character of what k decodes to
  - Therefore $s[0] =$ first character of s_{prev}
  - Therefore $s = s_{prev} + s_{prev}[0]$

LZW decoding pseudocode

```
LZW-decode(C)
C: stream of integers
1.    D ← dictionary that maps {0, ..., 127} to ASCII
2.    idx ← 128
3.    S ← empty string
4.    code ← first code from C
5.    s ← D(code); S.append(s)
6.    while there are more codes in C do
7.        s_prev ← s
8.        code ← next code of C
9.        if code = idx
10.           s ← s_prev + s_prev[0]
11.       else
12.           s ← D(code)
13.       S.append(s)
14.       D.insert(idx, s_prev + s[0])
15.       idx++
16.   return S
```

LZW decoding example revisited

- Example: 67 65 78 32 66 129 133 83

$D =$

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

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	ANA	133	ANA	129, A
83	S	134	ANAS	133, S

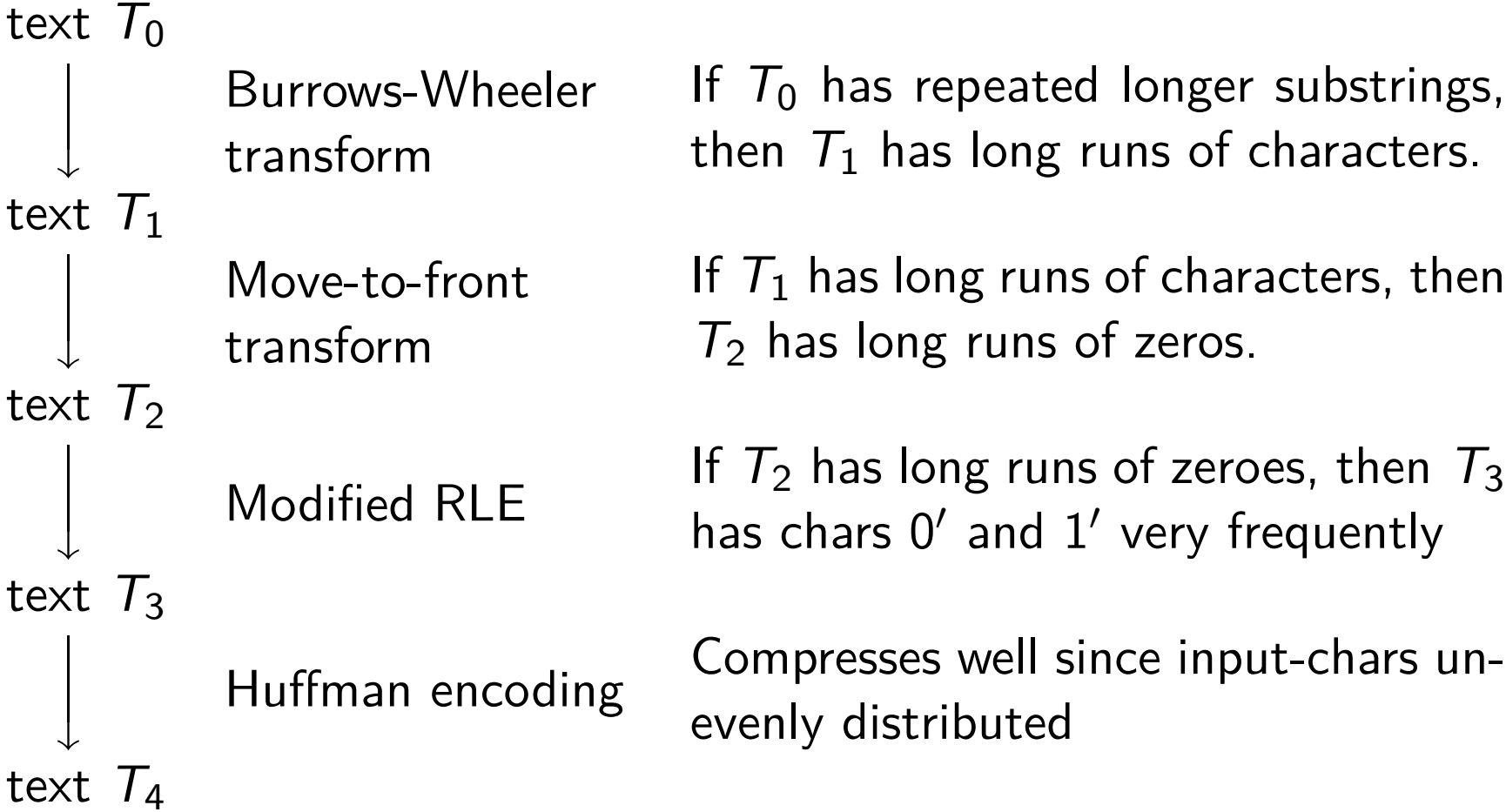
Compression summary

Huffman	**Run-length encoding**	**Lempel-Ziv-Welch**
variable-length	variable-length	fixed-length
single-character	multi-character	multi-character
2-pass	1-pass	1-pass
60% compression on English text	bad on text	45% compression on English text
optimal 01-prefix-code	good on long runs (e.g., pictures)	good on English text
must send dictionary	can be worse than ASCII	can be worse than ASCII
rarely used directly	rarely used directly	frequently used
part of pkzip, JPEG, MP3	fax machines, old picture-formats	GIF, some variants of PDF, Unix compress

Outline

bzip2 overview

To achieve even better compression, bzip2 uses *text transform*: Change input into a different text that is not necessarily shorter, but that has other desirable qualities.



Move-to-Front transform

Recall the MTF heuristic for self-organizing search:

- Dictionary is stored as an unsorted array or linked list
- After an element is accessed, move it to the front of the dictionary

How can we use this idea for text transformations?

Take advantage of *locality* in the data.
If we see a character now, we'll probably see it again soon.

Specifics: MTF is an *adaptive* text-transform algorithm.
If the source alphabet is Σ_S with size $|\Sigma_S| = m$,
then the coded alphabet will be $\Sigma_C = \{0, 1, \ldots, m-1\}$.

Move-to-Front Encoding/Decoding

```
MTF-encode(S)
1.    L ← array with Σ_S in some pre-agreed, fixed order
2.    while S has more characters do
3.        c ← next character of S
4.        output index i such that L[i] = c
5.        for j = i − 1 down to 0
6.            swap L[j] and L[j + 1]
```

Decoding works in *exactly* the same way:

```
MTF-decode(C)
1.    L ← array with Σ_S in some pre-agreed, fixed order
2.    while C has more characters do
3.        i ← next integer from C
4.        output L[i]
5.        for j = i − 1 down to 0
6.            swap L[j] and L[j + 1]
```

MTF Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

$$S = \texttt{MISSISSIPPI}$$

$$C =$$

- What does a run in S encode to in C?
- What does a run in C mean about the source S?

Outline

Burrows-Wheeler Transform

- Transforms source text to a coded text with the same letters, just in a different order.
- *The coded text will be more easily compressible with MTF.*
- Required: the source text S ends with *end-of-word character* \$ that occurs nowhere else in S.
- Encoding algorithm needs *all* of S (no streaming possible). BWT is a *block* compression method.
- (As we will see) decoding is more efficient than encoding, so BWT is an *asymmetric* scheme.

Crucial ingredient: A *cyclic shift* of a string X of length n is the concatenation of $X[i+1..n-1]$ and $X[0..i]$, for $0 \leq i < n$.

BWT Algorithm and Example

$S =$ `alf␣eats␣alfalfa$`

1. **Write all cyclic shifts**
2. **Sort cyclic shifts**
3. **Extract last characters from sorted shifts**

$C =$

```
$alf␣eats␣alfalfa
␣alfalfa$alf␣eats
␣eats␣alfalfa$alf
a$alf␣eats␣alfalf
alf␣eats␣alfalfa$
alfa$alf␣eats␣alf
alfalfa$alf␣eats␣
ats␣alfalfa$alf␣e
eats␣alfalfa$alf␣
f␣eats␣alfalfa$al
fa$alf␣eats␣alfal
falfa$alf␣eats␣al
lf␣eats␣alfalfa$a
lfa$alf␣eats␣alfa
lfalfa$alf␣eats␣a
s␣alfalfa$alf␣eat
ts␣alfalfa$alf␣ea
```

BWT Decoding

Idea: Given C, we can reconstruct the *first* and *last column* of the array of cyclic shifts by sorting.

C = `ard$rcaaaabb`

1. **Last column:** C
2. **First column:** C **sorted**
3. **Disambiguate by row-index**
4. **Starting from \$, recover** S

S =

```
 $,3.........a,0
 a,0.........r,1
 a,6.........d,2
 a,7.........$,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

BWT-decoding($C[0..n-1]$)
C : string of characters over alphabet Σ_C

```
1.    A ← array of size n
2.    for i = 0 to n − 1
3.        A[i] ← (C[i], i)
4.    Stably sort A by first entry
5.    S ← empty string
6.    for j = 0 to n
7.        if C[j] = $ break
8.    repeat
9.        j ← second entry of A[j]
10.       append C[j] to S
11.   until C[j] = $
12.   return S
```

BWT Overview

Encoding cost: $O(n^2)$ (using MSD radix sort) and often better

Encoding is theoretically possible in $O(n)$ time:

- Sorting cyclic shifts of S is equivalent to sorting the suffixes of $S \cdot S$ that have length $> n$
- This can be done by traversing the suffix tree of $S \cdot S$

Decoding cost: $O(n)$ (faster than encoding)

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

Tends to be slower than other methods,
but (combined with MTF, RLE and Huffman) gives better compression.