CS 240 – Data Structures and Data Management

Module 10: Data Compression

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

Outline

- Data Compression
  - Background
  - Single-Character Encodings
  - Huffman Codes
  - Lempel-Ziv-Welch
  - Combining Compression Schemes: bzip2
  - Burrows-Wheeler Transform

Outline

- Data Compression
  - Background
  - Single-Character Encodings
  - Huffman Codes
  - Lempel-Ziv-Welch
  - Combining Compression Schemes: bzip2
  - Burrows-Wheeler Transform

Data Compression Introduction

- **The problem**: How to store and transmit data efficiently?
- Source text:
  - original data, string S of characters from ***source alphabet*** Σ_S
- Coded text
  - encoded data, string C of characters from ***coded alphabet*** Σ_C
- Encoding [scheme]
  - algorithm mapping source text to coded text
- Decoding [scheme]
  - algorithm mapping coded text back to original source text

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

- Source "text" can be any sort of data (not always text)
- Usually the coded alphabet is binary $\Sigma_C = \{0,1\}$
- Consider lossless compression: exact recovery of S from C

Judging Encoding Schemes

- Main objective: for data compression, want to minimize the size of the coded text
- Measure the ***compression ratio***

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

- Examples:

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

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

- Want to achieve compression ration smaller than 1
    - can always achieve compression ratio of 1 by sending S without changes

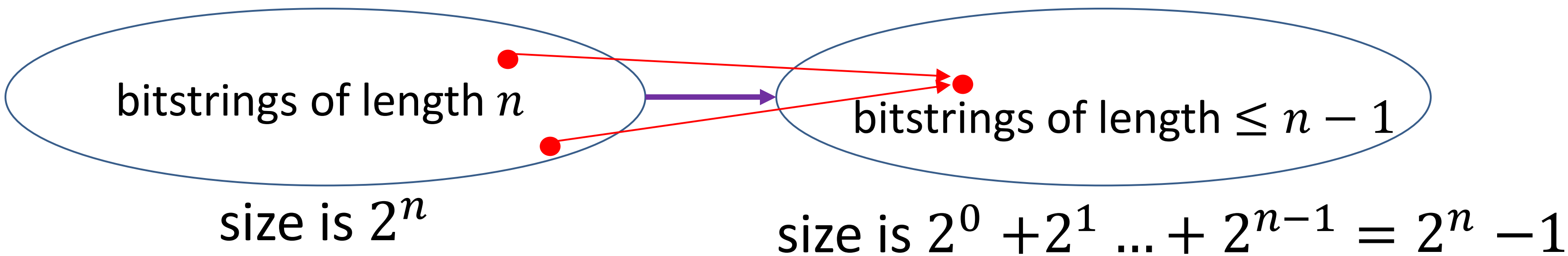
Judging Encoding Schemes

- Also measure efficiency of encoding/decoding algorithms, as for any usual algorithm
    - always need time $\Omega(|S| + |C|)$
        - sometimes need more time
- Other possible goals, not studied in this course
    - reliability (e.g. error-correcting codes)
    - security (e.g. encryption)

Impossibility of Compressing

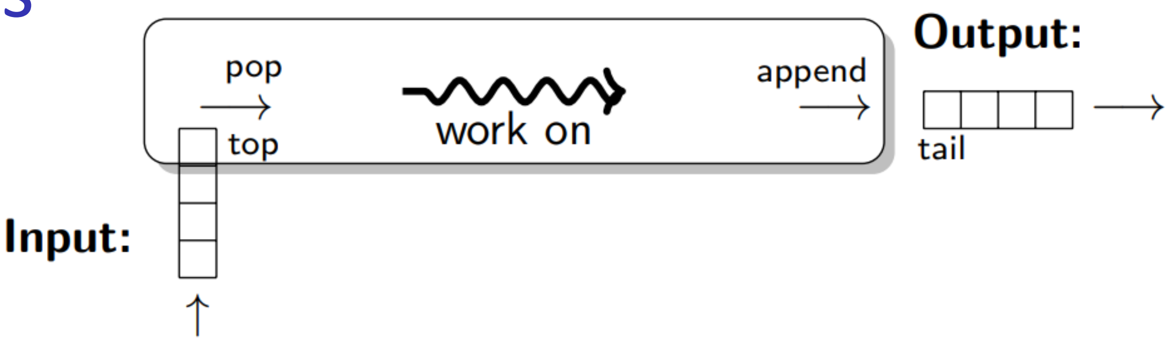
- Observation: No lossless encoding scheme can have compression ratio < 1 for **all** input strings
- Proof: (for $\Sigma_S = \Sigma_C = \{0,1\}$, by contradiction)

  Fix n, and assume all length n strings get shorter



- So impossible to provide good worst-case compression bounds
- However real-life data is usually far from random, it has some strings that occur more frequently than others
  - can design compression schemes that work well for frequently occurring strings

Detour: Streams



- Usually texts are huge and do not fit into computer memory
- Therefore usually store S and C as *streams*
  - input stream `(~std::cin)`
    - read one character at a time
      - $pop(), top(), isEmpty()$
    - sometimes need $reset()$ to start processing from the start
  - output stream `(~std::cout)`
    - write one character at a time
      - $append(),\ isEmpty()$
- Advantage of streams
  - can start processing text while it is still being loaded
  - avoids needing to hold the entire text in memory at once

Outline

- Data Compression
  - Background
  - Single-Character Encodings
  - Huffman Codes
  - Run-Length Encoding
  - Lempel-Ziv-Welch
  - Combining Compression Schemes: bzip2
  - Burrows-Wheeler Transform

Character Encodings

- A character encoding E (or single-character encoding) maps each *character* in the source alphabet to a *string* in code alphabet
$$E : \Sigma_S \to \Sigma_C^*$$
  - for $c \in \Sigma_S$, $E(c)$ is called the *codeword* (or *code*) of c
- Two possibilities
  1. Fixed-length code: all codewords have the same length

$c \in \Sigma_S$	⊔	A	E	N	O	T
$E(c)$	000	001	011	100	101	111

  2. Variable-length code: codewords may have different lengths

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

Fixed Length Character Encoding

- Example: ASCII (American Standard Code for Information Interchange), 1963

char in Σ_S	null	start of heading	start of text	...	0	1	...	A	B	...	~	delete
code	0	1	2	...	48	49	...	65	66	...	126	127
code in binary	0000000	0000001	0000010		0110000	0110001		1000001	1000010		1111110	1111111

- Each codeword $E(c)$ has length 7 bits
- Encoding/Decoding is easy: just concatenate/decode the next 7 bits
  - A P P L E ↔ (65, 80, 80, 76, 69) ↔ 1000001 1010000 1010000 1001100 1000101
  - here $|S| = 5, |C| = 5 \cdot 7, |\Sigma_S| = 128$
- Standard in all computers and often our source alphabet
- Other (earlier) fixed-length codes: Baudot code, Murray code
- Fixed-length codes do not compress
  - let $|E(c)| = b$ and assume binary code alphabet

$$\frac{|C| \cdot \log|\Sigma_C|}{|S| \cdot \log|\Sigma_S|} = \frac{b \cdot |S|}{|S| \cdot \log 2^b} = 1$$

Better Idea: Variable-Length Codes

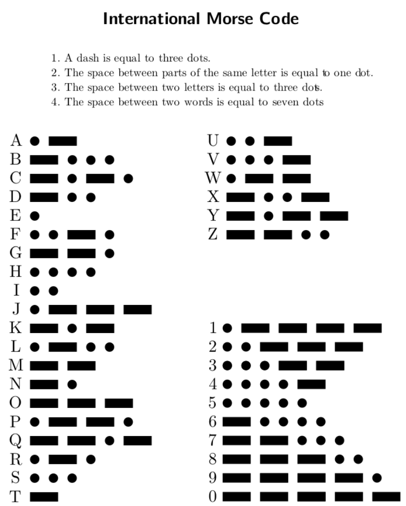
- **Observation:** Some alphabet letters occur more often than others
  - example: frequency of letters in typical English text

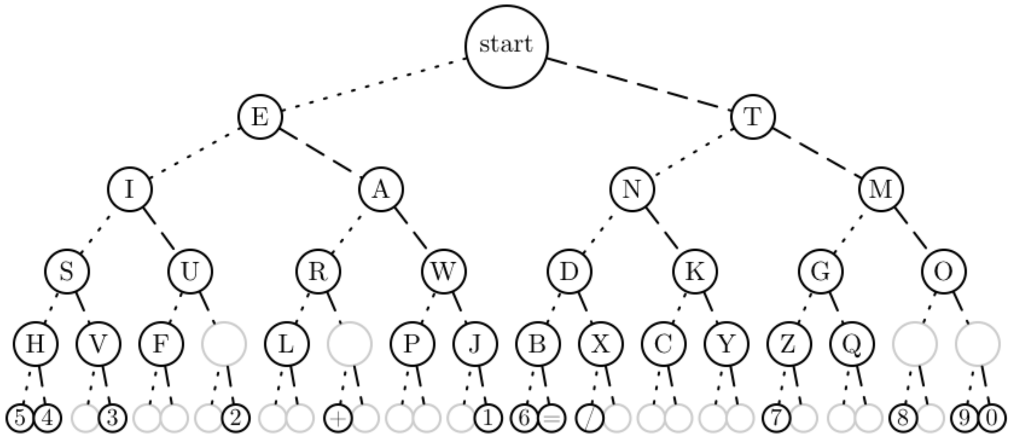
e	12.70%	**d**	4.25%	**p**	1.93%
t	9.06%	**l**	4.03%	**b**	1.49%
a	8.17%	**c**	2.78%	**v**	0.98%
o	7.51%	**u**	2.76%	**k**	0.77%
i	6.97%	**m**	2.41%	**j**	0.15%
n	6.75%	**w**	2.36%	**x**	0.15%
s	6.33%	**f**	2.23%	**q**	0.10%
h	6.09%	**g**	2.02%	**z**	0.07%
r	5.99%	**y**	1.97%		

- **Idea**: use shorter codes for more frequent characters
  - as before, map source alphabet to codewords: $E : \Sigma_S \to \Sigma_C^*$
  - but not all codewords have the same length
  - this should make the coded text shorter

Variable-Length Codes

- Example 1: Morse code





- Example 2: UTF-8 encoding of Unicode
  - there are roughly 150,000 Unicode characters
  - 1-4 bytes to encode any Unicode character

Encoding

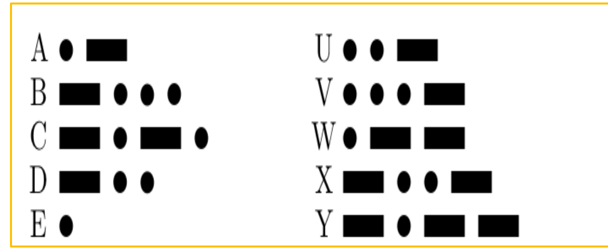
- Assume we have some character encoding $E: \Sigma_S \to \Sigma_C^*$
- E is a dictionary with keys in Σ_S

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

- Using dictionary below, encode AN⊔ANT → 01 001 000 0100111

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

Decoding



- The decoding algorithm must map Σ_C^* to Σ_S
- The code must be *uniquely decodable*
  - false for Morse code as described

  - Morse code uses 'end of character' pause to avoid ambiguity
  - this is equivalent to adding '$' at the end of each word
  - encoding is *prefix-free* if '$' added
    - no codeword is a prefix of another codeword

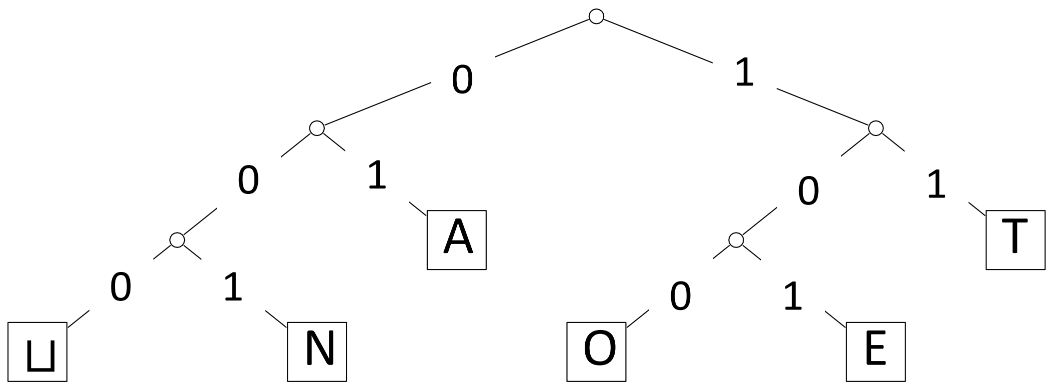
      aab$

      aab~~$~~**$
  - prefix-free codes are uniquely decodable
    - C =**** $c_1 c_2 c_3 c_4 c_5 c_6$*****

      $c_1 c_2 c_3$ is a prefix of $c_1 c_2 c_3 c_4 c_5$, contradiction
- Adding '$', would mean coded alphabet is not binary, not desirable
- So we will require encoding to be prefix free
  - example: codewords 000, 01, 101, 001, 100, 11 are prefix-free

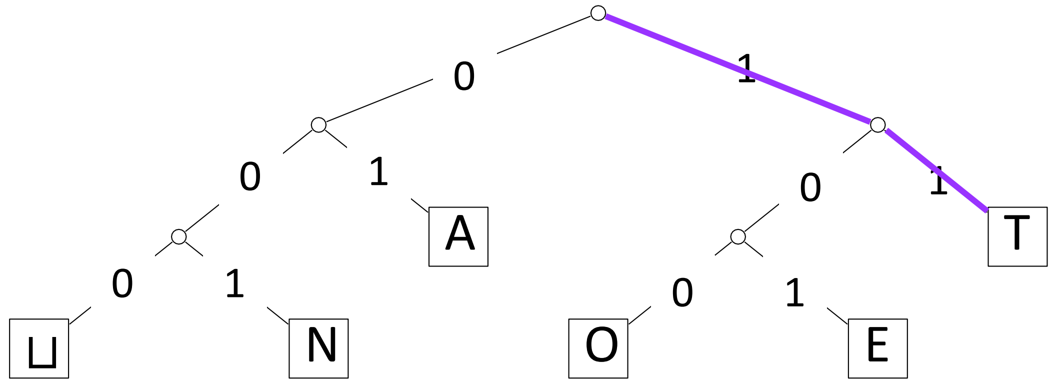
Decoding

- From now on only consider *prefix-free* codes E
  - no codeword is a prefix of another codeword
  - Uniquely decodable
- Store codes in a *trie* with characters of Σ_S at the leaves
  - prefix-free codes can be stored at the leafs without adding $



Example: Prefix-free Decoding

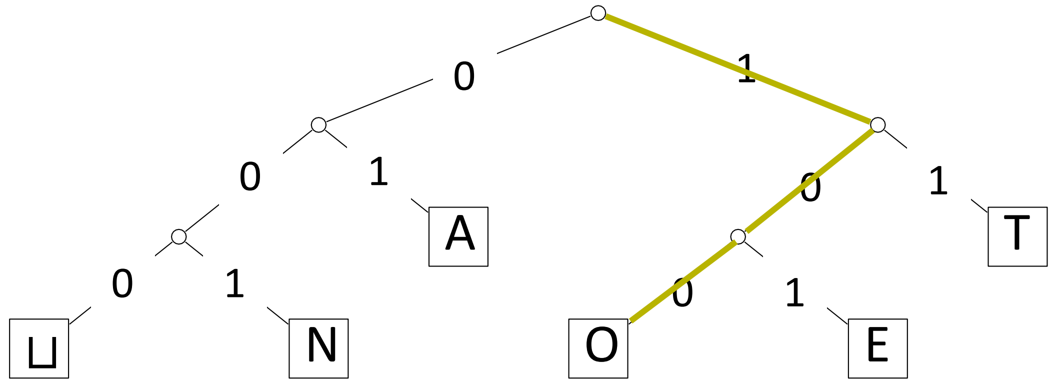
- Decode from a trie



- Decode 111000001010111 → T

Example: Prefix-free Decoding

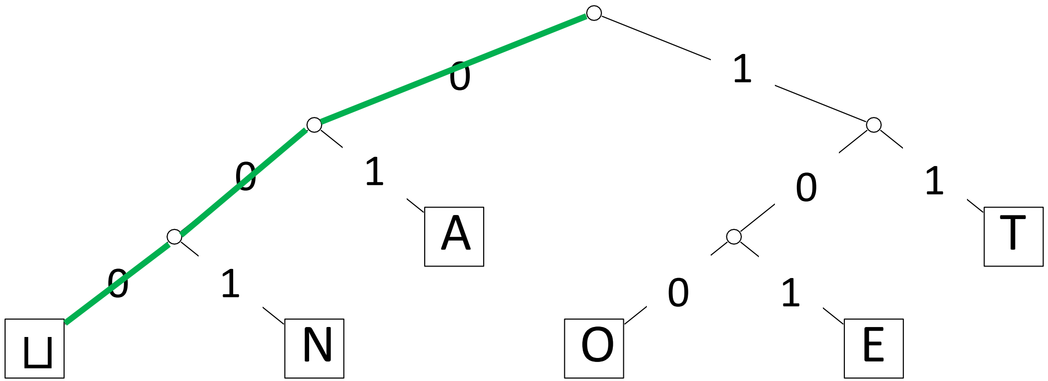
- Decode from a trie



- Decode 11100001010111 → T O

Example: Prefix-free Decoding

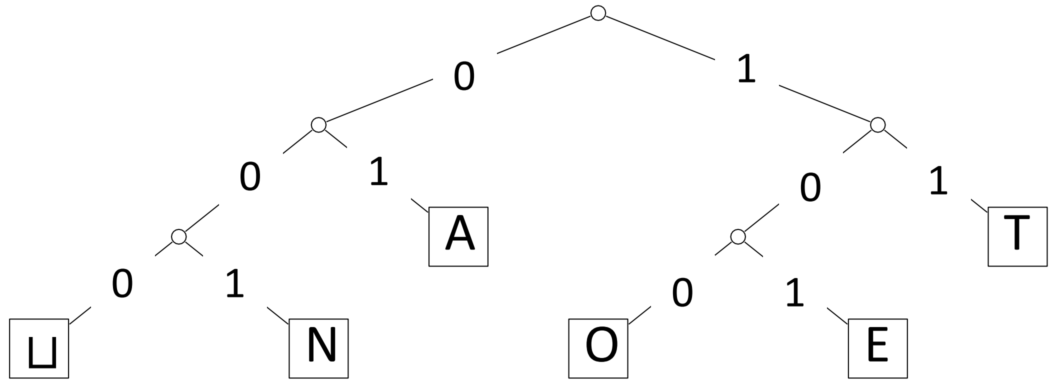
- Decode from a trie



- Decode 111000001010111 → TO⊔

Example: Prefix-free Decoding

- Decode from a trie



- Decode 11100000001010111 → TO⊔EAT
- Run-time: $O(|C|)$

Decoding of Prefix-Free Codes

prefixFree::decoding$(T,\ C,\ S)$
T: trie of a prefix-free code, C: input-stream with characters in Σ_C
S: output-stream

while C is non-empty // iterate over all codewords
    $z \leftarrow T$.*root*
    while z is not a leaf // read next codeword
        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)

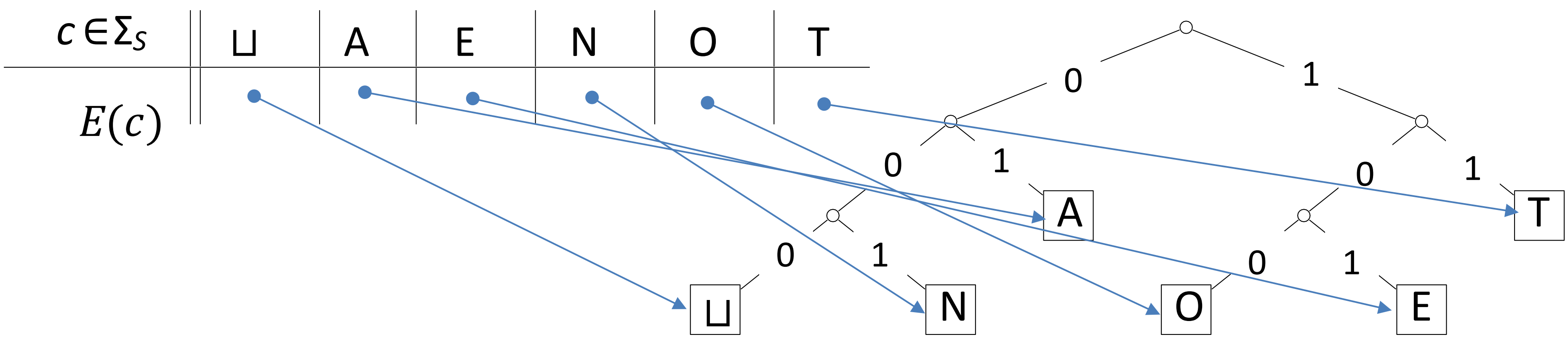
- Run-time: $O(|C|)$
- Detects if the encoding is invalid

Encoding from the Trie

- Explained previously how to encode from a table

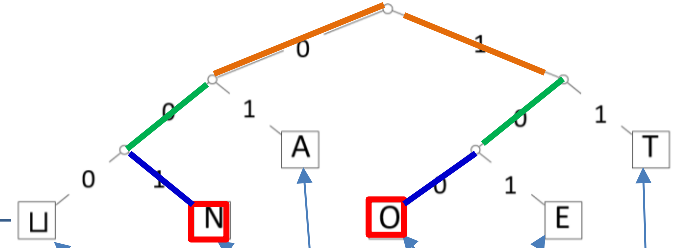
$c \in \Sigma_S$	⊔	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

- Table wastes space, codewords can be quite long
- Better idea: store codewords via links to the trie leaves



Encoding from the Trie

- Can encode directly from the trie T



$E =$

⊔	A	E	N	O	T

prefixFree::encoding$(T,\ S,\ C)$

T : prefix-free code trie, S: input-stream with characters in Σ_S

$O(|T|)$

$E \leftarrow$ array of nodes in T indexed by Σ_S
for all leaves l in T
  E[character at l] $\leftarrow l$

$O(|C|)$

while S is non-empty
  $w \leftarrow$ empty bitstring; $v \leftarrow E[S.pop()]$
  while v is not the root
    w.prepend (character from v to its parent)
    $v \leftarrow$ parent(v)
  // now w is the encoding of S
  append each bit w of to C

$S = ON$
$i = 0$ (letter O)
$w = \Lambda$
$w = 0$
$w = 00$
$w = 100$
$C = 100$
$i = 1$ (letter N)
$w = \Lambda$
$w = 1$
$w = 01$
$w = 001$
$C = 100\ 001$

- Run-time: $O(|T| + |C|)$
  - have to visit all trie nodes, and insert leaves into E
  - we assume T has no nodes with one child
    - #leaves $-1 \geq$ #internal nodes
    - $|\Sigma_S| \cdot 2 - 1 \geq$ #internal nodes + #leaves $= |T|$
    - $O(|\Sigma_S| + |C|)$

Outline

- Data Compression
  - Background
  - Single-Character Encodings
  - Huffman Codes
  - Lempel-Ziv-Welch
  - Combining Compression Schemes: bzip2
  - Burrows-Wheeler Transform

Huffman's Algorithm: Building the Best Trie

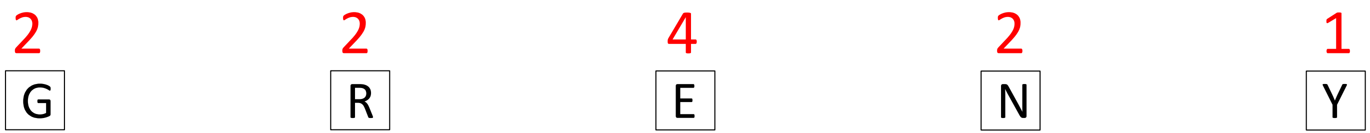
- How to determine the best trie for a given source text S ?
    - i.e. try giving shortest $|C|$
- Idea: infrequent characters should be far down in the trie

Example: Huffman Tree Construction

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

$$G{:}\,2,\ \ R{:}\ 2,\ \ E{:}\,4,\ \ N{:}\,2,\ \ Y{:}\,1$$

- Put each character into its own trie (single node, height 0)
    - each trie has a frequency
    - initially, frequency is equal to its character frequency

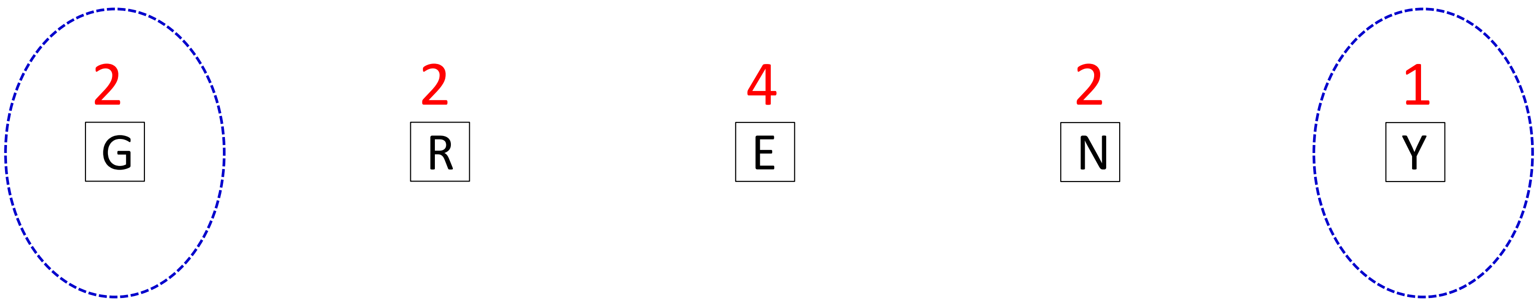


Example: Huffman Tree Construction

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

$$G{:}\,2,\ \ R{:}\ 2,\ \ E{:}\,4,\ \ N{:}\,2,\ \ Y{:}\,1$$

- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

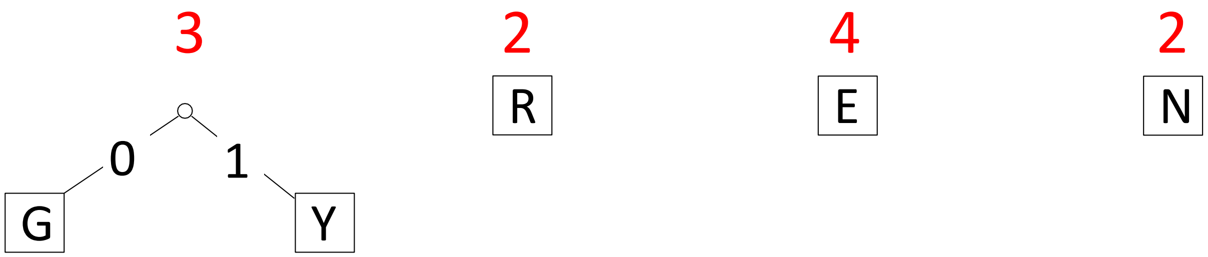


Example: Huffman Tree Construction

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

$$G: 2,\ R: 2,\ E: 4,\ N: 2,\ Y: 1$$

- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

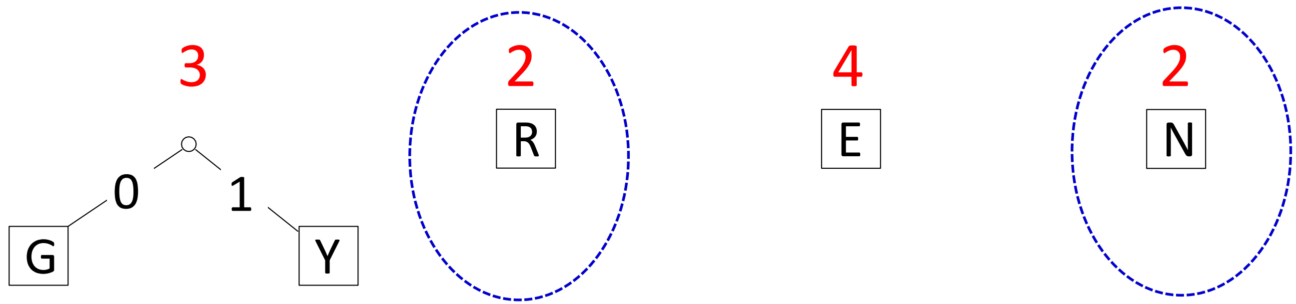


Example: Huffman Tree Construction

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

$$G{:}\,2,\ R{:}\ 2,\ E{:}\,4,\ N{:}\,2,\ Y{:}\,1$$

- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

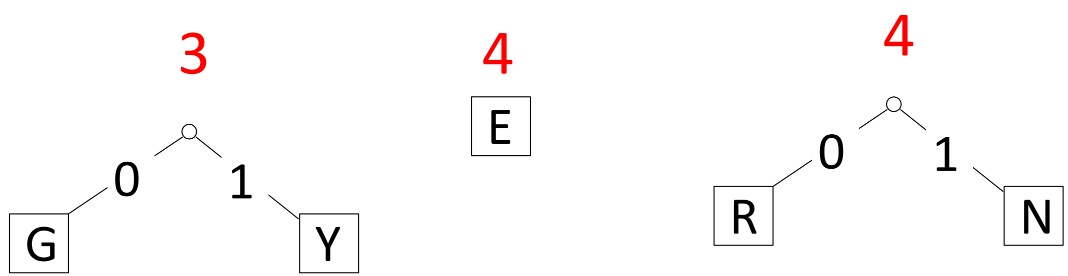


Example: Huffman Tree Construction

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

$$G\!:2,\;\; R\!:\; 2,\;\; E\!:4,\;\; N\!:2,\;\; Y\!:1$$

- Join two least frequent tries into a new trie
    - frequency of the new trie = sum of old trie frequencies

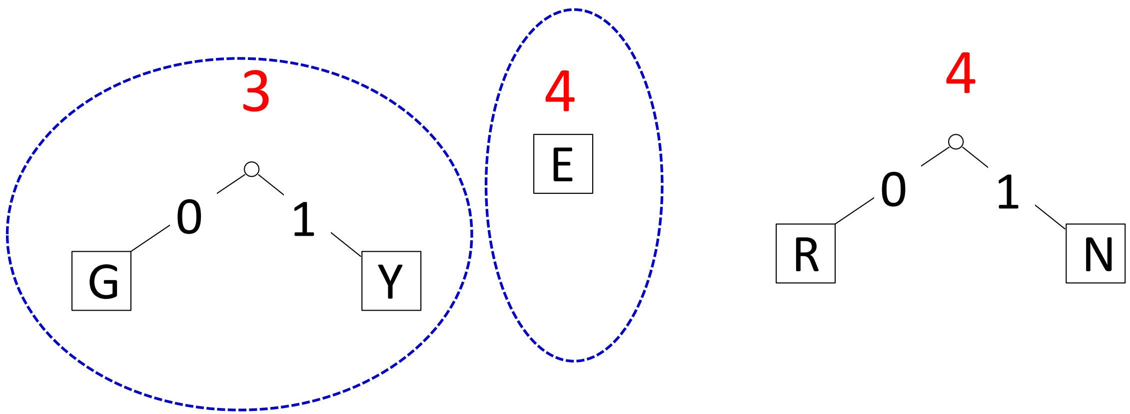


Example: Huffman Tree Construction

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

$$G: 2,\ R: 2,\ E: 4,\ N: 2,\ Y: 1$$

- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

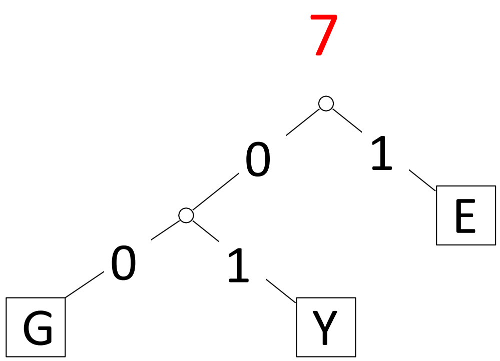


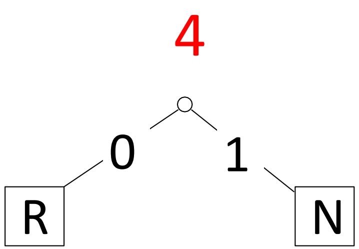
Example: Huffman Tree Construction

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

$$G{:}\,2,\ R{:}\ 2,\ E{:}\,4,\ N{:}\,2,\ Y{:}\,1$$

- Join two least frequent tries into a new trie
    - frequency of the new trie = sum of old trie frequencies



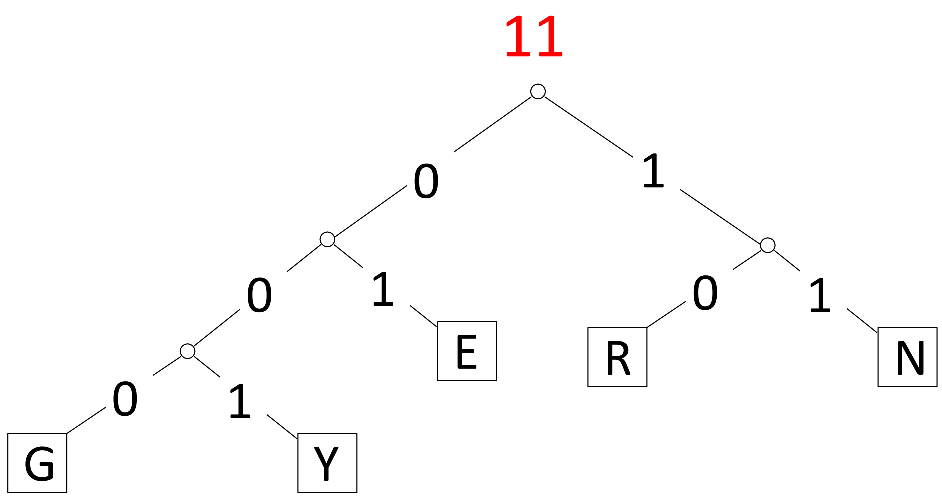


Example: Huffman Tree Construction

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

$$G\colon 2,\ R\colon\ 2,\ E\colon 4,\ N\colon 2,\ Y\colon 1$$

- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

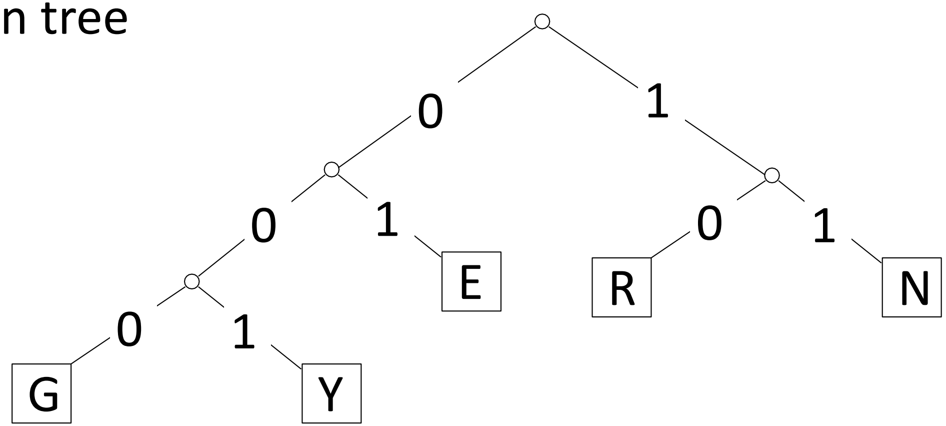


Example: Huffman Tree Construction

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

$$G\!: 2,\ \ R\!:\ 2,\ \ E\!: 4,\ \ N\!: 2,\ \ Y\!: 1$$

- Final Huffman tree



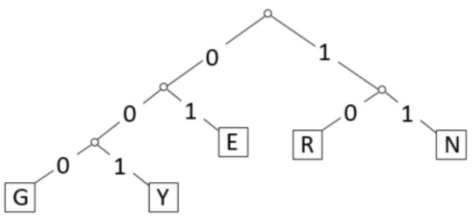
- GREENENERGY → 000 10 01 01 11 01 11 01 10 000 001
- Compression ratio

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

- Frequencies are not skewed enough to lead to good compression

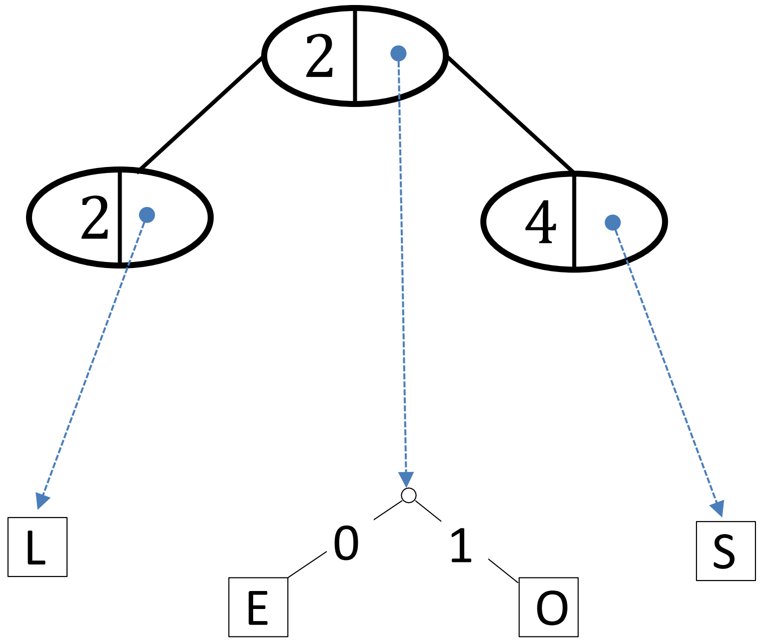
Huffman Algorithm Summary

- **Greedy algorithm**: always pair up least frequent characters
  1) determine frequency of each character $c \in \Sigma$ in S
  2) for each $c \in \Sigma$, create trie of height 0 holding only c
     - call it c-trie
  3) assign weight to each trie
     - weight trie character
  4) find and merge two tries with the minimum weight
     - new interior node added
     - the new weight is the sum of merged tries weights
     - corresponds to adding one bit to encoding of each character
  5) repeat Steps 4 until there is only 1 trie left
     - this is D, the final decoder
- Min-heap for efficient implementation: step 4 is two *delete-min* one *insert*



Heap Storing Tries during Huffman Tree Construction

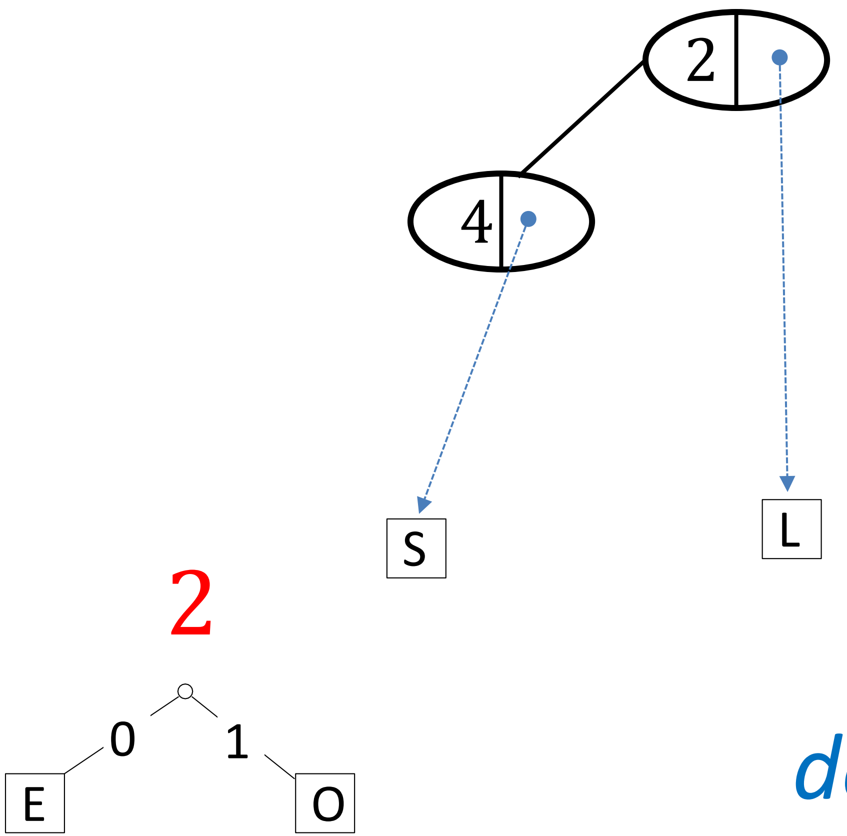
- (key,value) = (trie weight, link to trie)
- step 4 is two *delete-min*s, one *insert*



delete-min

Heap Storing Tries during Huffman Tree Construction

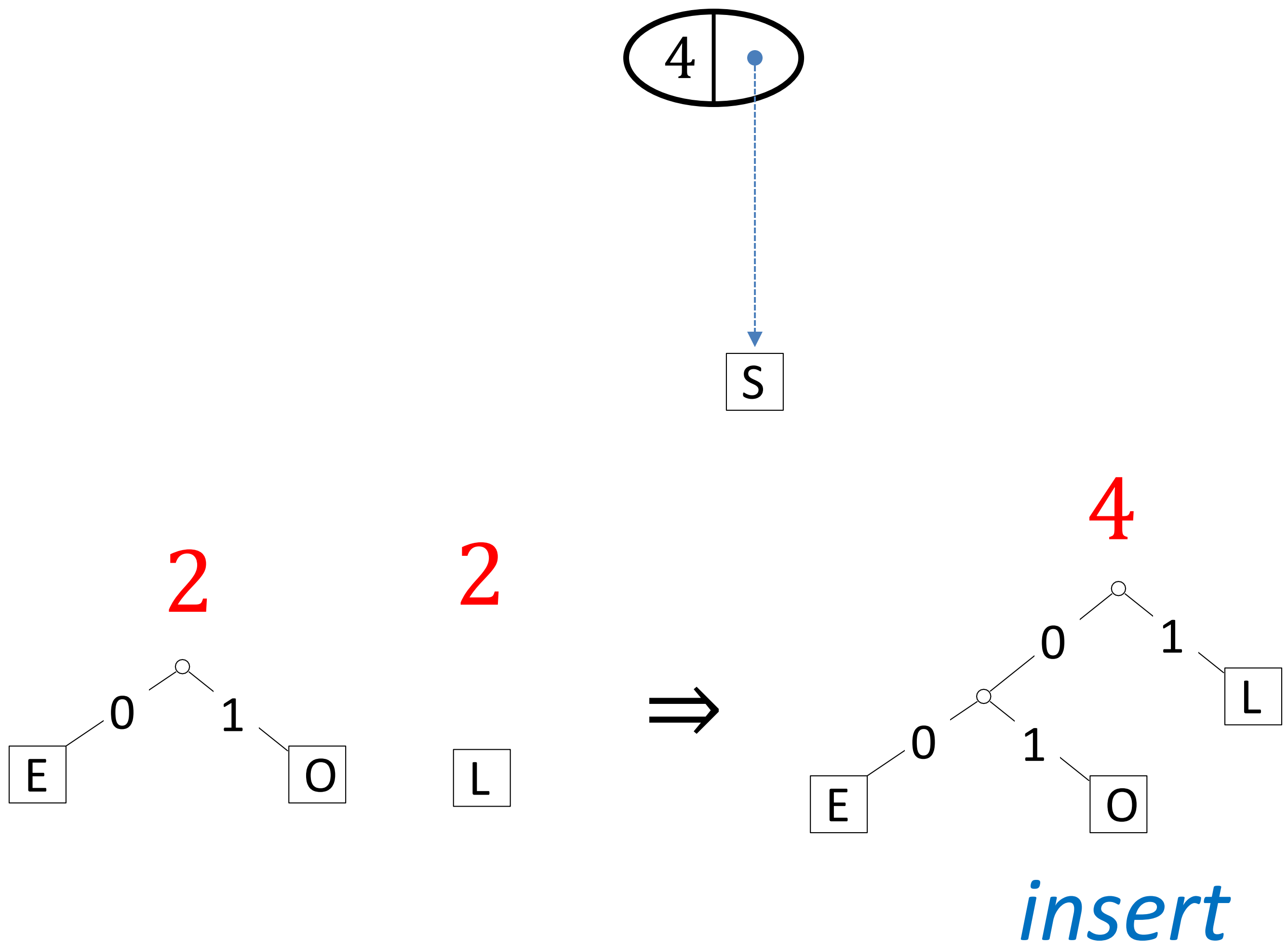
- (key,value) = (trie weight, link to trie)
- step 4 is two *delete-mins*, one *insert*



delete-min

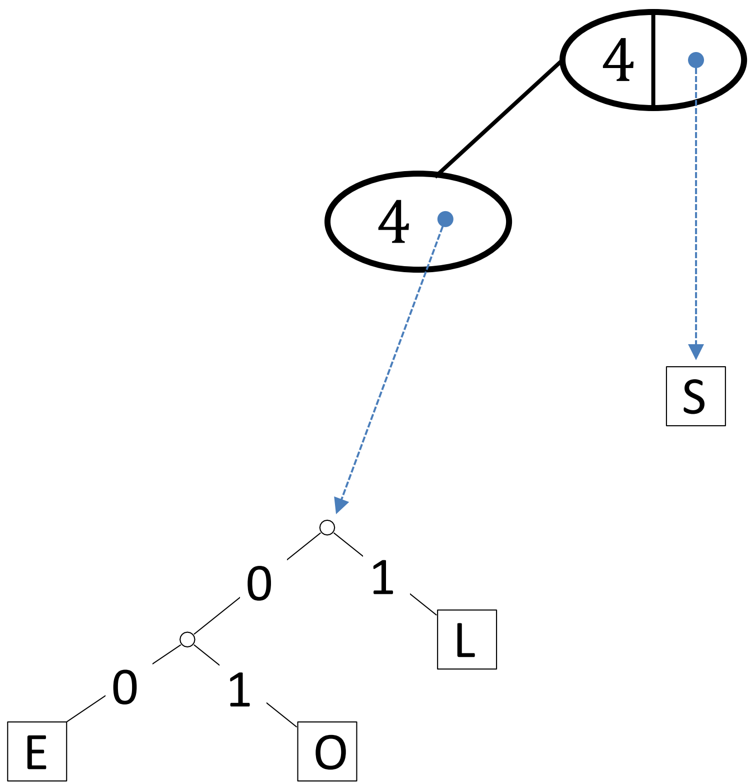
Heap Storing Tries during Huffman Tree Construction

- (key,value) = (trie weight, link to trie)
- step 4 is two *delete-mins*, one *insert*



Heap Storing Tries during Huffman Tree Construction

- (key,value) = (trie weight, link to trie)
- step 4 is two *delete-min*s, one *insert*



Huffman's Algorithm Pseudocode

Huffman::encoding(S, C)
S: input-stream (length n) with characters in Σ_S, C: output-stream, initially empty
$f \leftarrow$ array indexed by Σ_S, initialized to 0
while S is non-empty **do increase** $f[S.pop()]$ by 1 // get frequencies — $O(n)$
$Q \leftarrow$ min-oriented priority queue to store tries
for all $c \in \Sigma_S$ with $f[c] > 0$ — $O(|\Sigma_S| \log |\Sigma_S|)$
    Q.*insert*(single-node trie for c, $f[c]$)
while $Q.size() > 1$ — $O(|\Sigma_S| \log |\Sigma_S|)$
    $(T_1, f_1) \leftarrow Q.deleteMin()$
    $(T_2, f_2) \leftarrow Q.deleteMin()$
    Q.*insert*(trie with T_1, T_2 as subtries, $f_1 + f_2$)
$T \leftarrow Q.deleteMin()$ // trie for decoding
reset input-stream S // read all of S, need to read again for encoding
prefixFree::encoding(T, S, C) // perform actual encoding — $O(|\Sigma_S| + |C|)$

- Total time is $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
    - $n < |C|$

Huffman Coding Discussion

- We require $|\Sigma_S| \geq 2$
- Codes are prefix-free by construction
  - a trie leaf cannot be a prefix of another leaf
- The constructed trie is *optimal* in the sense that the coded text C is shortest among all prefix-free character encodings with $\Sigma_C = \{0, 1\}$
  - proof is in the course notes
- Constructed trie is **not unique**
  - so decoding trie must be transmitted along with the coded text
  - this may make encoding bigger than source text!
- Encoding must pass through stream twice
  1. to compute frequencies and to encode
  2. cannot use stream unless it can be reset
- Encoding runtime: $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
- Decoding run-time: $O(|C|)$
- Good compression if character frequencies are skewed
- Many variations
  - tie-breaking rules, estimate frequencies, adaptively change encoding, etc.

Outline

- Compression
  - Encoding Basics
  - Huffman Codes
  - Lempel-Ziv-Welch
  - bzip2
  - Burrows-Wheeler Transform

Longer Patterns in Input

- Huffman takes 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”, “
”
  - Video
    - repeated background between frames, shifted sub-image
- Could find the most frequent substrings of length up to k and store them in a dictionary (in addition to characters, i.e. strings of length 1)

	null	start of heading	start of text	. . .	A	. . .	delete	er	in	...	ed	the
code	0	1	2	. . .	65	. . .	127	128	129	...		255
code in binary	00000000	00000001	00000010		001000001		01111111	11000001	11000010	...	11111110	11111111

  - however, each text has its own set of most frequently occurring substrings

Lempel-Ziv-Welch Compression

- **Ingredient 1** for Lempel-Ziv-Welch compression
  - encode characters and frequent substrings
    - discover and encode frequent substring as we process text
      - no need to know frequent substrings beforehand

Single-Character vs Multi-Character Encoding

- Single character encoding: each source-text character receives one codeword

$$S = \underbrace{\text{b}}_{01}\;\underbrace{\text{a}}_{1}\;\underbrace{\text{n}}_{11}\;\underbrace{\text{a}}_{1}\;\underbrace{\text{n}}_{11}\;\underbrace{\text{a}}_{1}$$

- Multi-character encoding: multiple source-text characters can receive one codeword

$$S = \underbrace{\text{b}}_{01}\;\underbrace{\text{a n a}}_{11}\;\underbrace{\text{n a}}_{101}$$

- Lempel-Ziv-Welch is a multi-character encoding

Adaptive Dictionaries

- ASCII uses a *fixed* dictionary
    - same dictionary for any text encoded
    - no need to pass dictionary to the decoder
- Huffman's dictionary is not *fixed* but it is *static*
    - dictionary is not fixed: each text has its own dictionary
    - dictionary is *static*: dictionary *does not change* for entire encoding/decoding
    - need to pass dictionary to the decoder
- **Ingredient 2** for LZW: *adaptive dictionary*
    - dictionary constructed during encoding/decoding
    - start with some initial fixed dictionary D_0
        - usually ASCII
    - at iteration $i \geq 0$, D_i is used to determine the ith output
    - after ith output (iteration i), update D_i to D_{i+1}
        - $D_{i+1} \leftarrow D_i$.*insert*(new character combination)
    - decoder knows (i.e. be able to reconstruct from the coded text) how encoder changed the dictionary
        - no need to send dictionary with the encoding,

LZW Encoding: Main Idea

- Iteration i of encoding
- Current D_i = {a:65, b: 66, c: 67 ab:128, bb:129}

S = ab**bb**aad C = 65 66 129

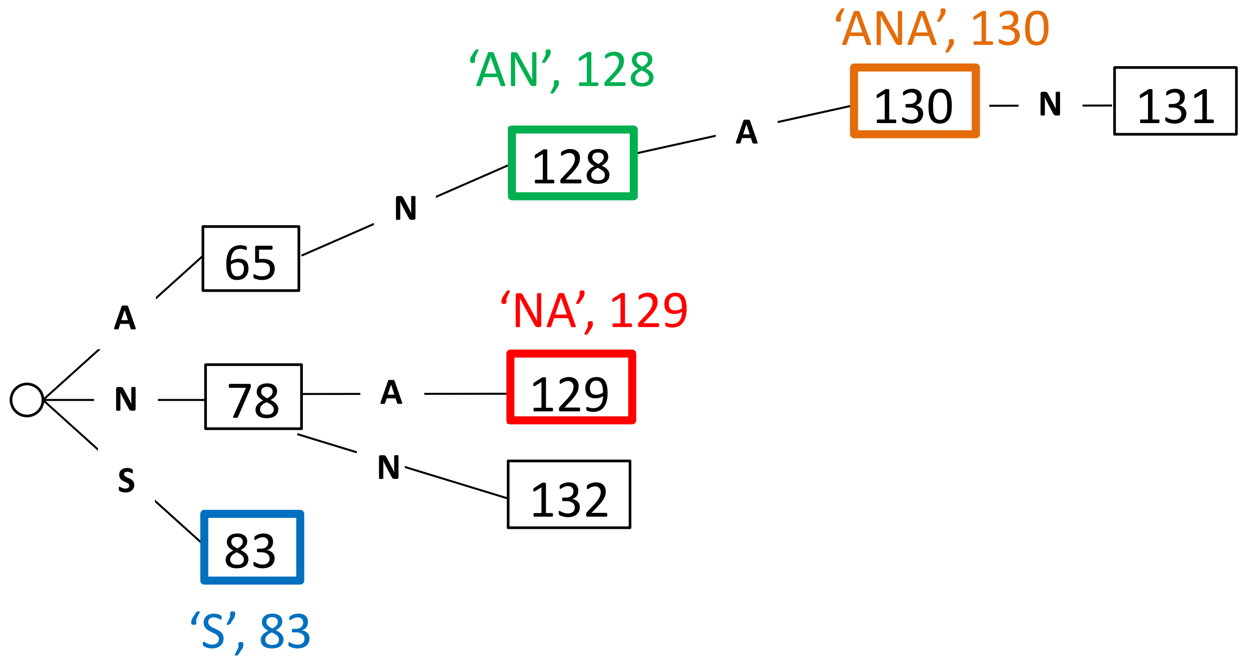
  - find longest substring that starts at current pointer and is in the dictionary
  - encode 'bb' with 129
  - $D_{i+1} = D_i$.*insert*('bba', *nextAvailableCodenumber* = 130)
  - 'bba' would have been useful at iteration i, so likely useful in the future

- After iteration i

$$D_{i+1} = \{a{:}65,\ b{:}\ 66,\ c{:}67,\ ab{:}128,\ bb{:}\ 129,\ bba{:}130\}$$

- *codenumber* = *codeword* = *code*

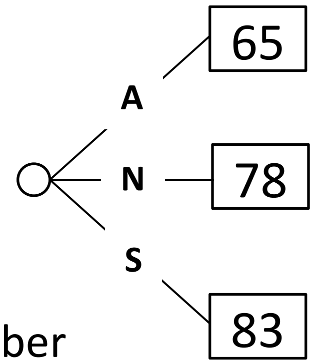
Tries for LZW Encoding



- Store (string, codenumber) pairs, with string being the key
- Variation of tries different from what we have seen before
- Trie stores codenumbers at **all** nodes (external and internal) except the root
  - works because a string is inserted only after all its prefixes are inserted
- Do not store the string key explicitly, store only the codenumber
  - read the string key corresponding to each codenumber from the edges

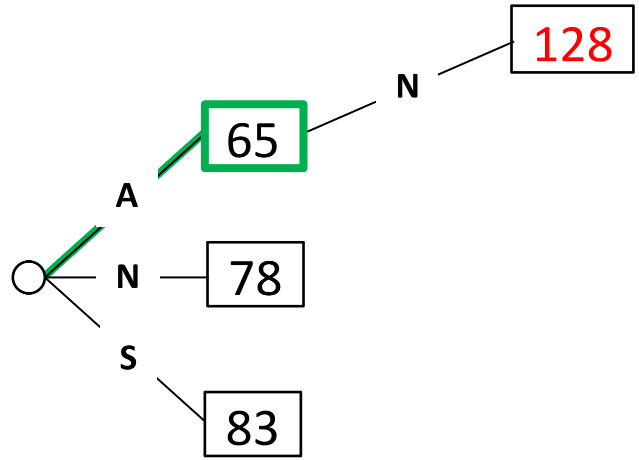
LZW Example

- Start dictionary D
  - ASCII characters
  - codes from 0 to 127
  - next inserted code will be 128
  - variable idx keeps track of next available codenumber
  - initialize $idx = 128$



- Text A N A N A N A N N A

LZW Example



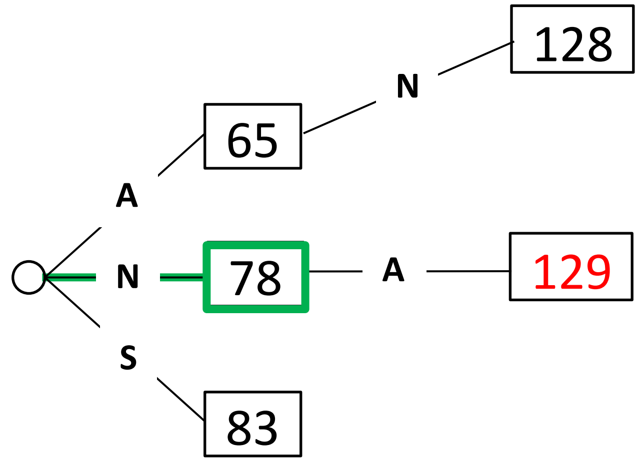
- Dictionary D
  - $idx =$ 129

add to dictionary

- Text A N A N A N A N N A
- Encoding 65

- Add to dictionary “string just encoded” + “first character of next string to be encoded”
- Inserting new item into D is $O(1)$ since we stopped at the right node in the trie when we searched for ‘A’

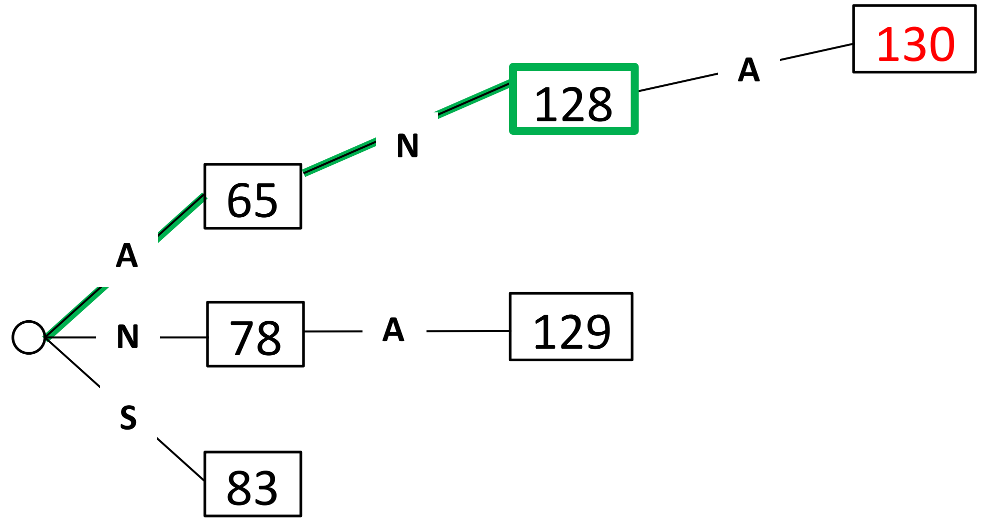
LZW Example



- Dictionary D
  - $idx = 130$

- Text A | N A N A N A N N A
- Encoding 65 | 78

LZW Example



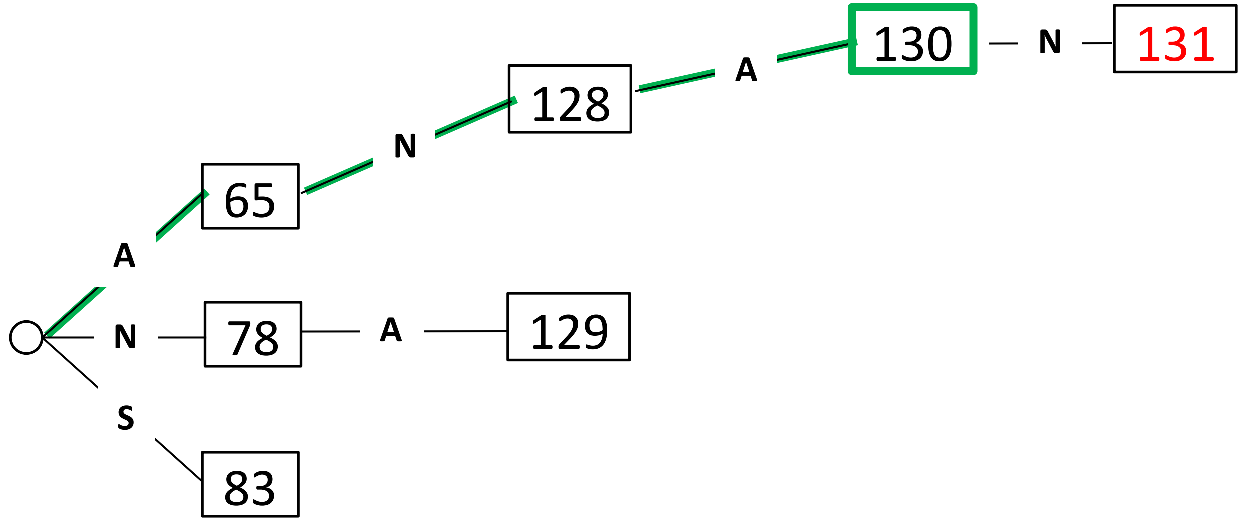
- Dictionary D
  - $idx = 131$

add to dictionary

Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128							

LZW Example

- Dictionary D
  - $idx = 132$

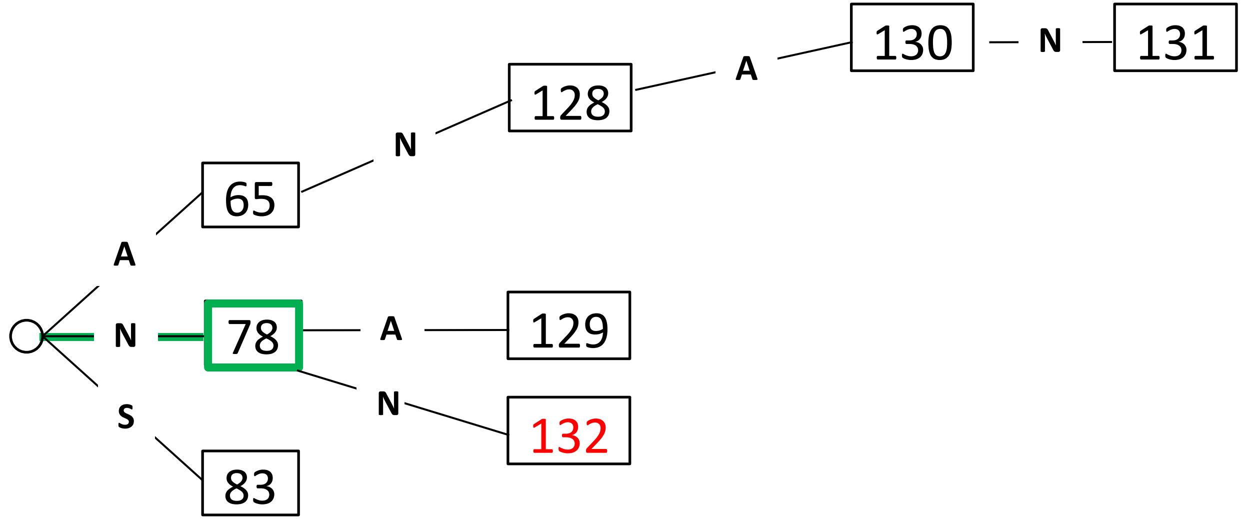


add to dictionary

Text	A	N	A N	A N A	N N A
Encoding	65	78	128	130	

LZW Example

- Dictionary D
  - $idx = 133$



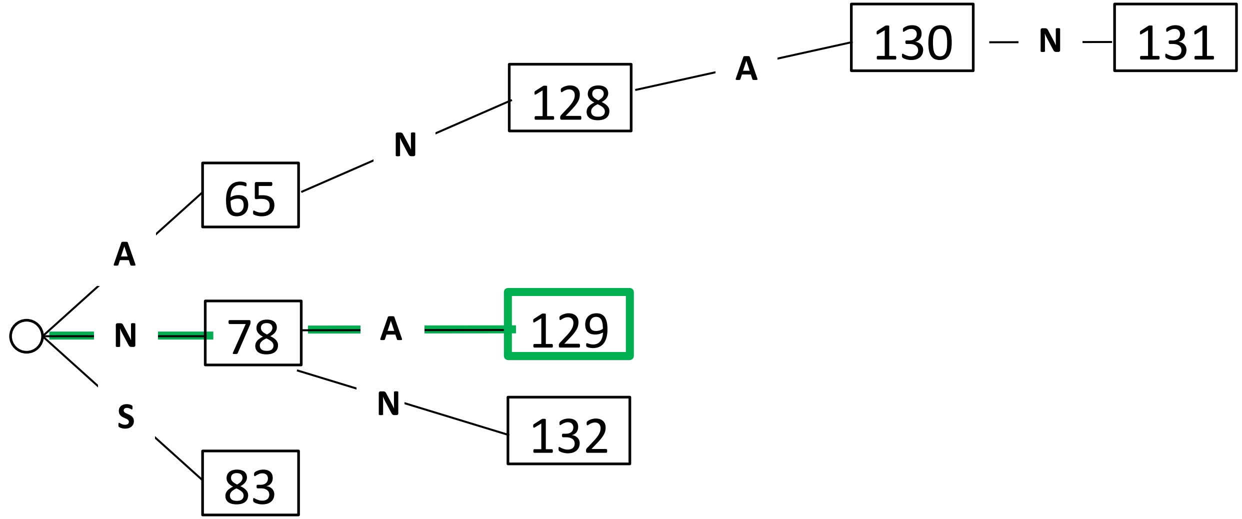
add to dictionary

- Text

A	N	A N	A N A	N	N A
65	78	128	130	78	

- Encoding: 65, 78, 128, 130, 78

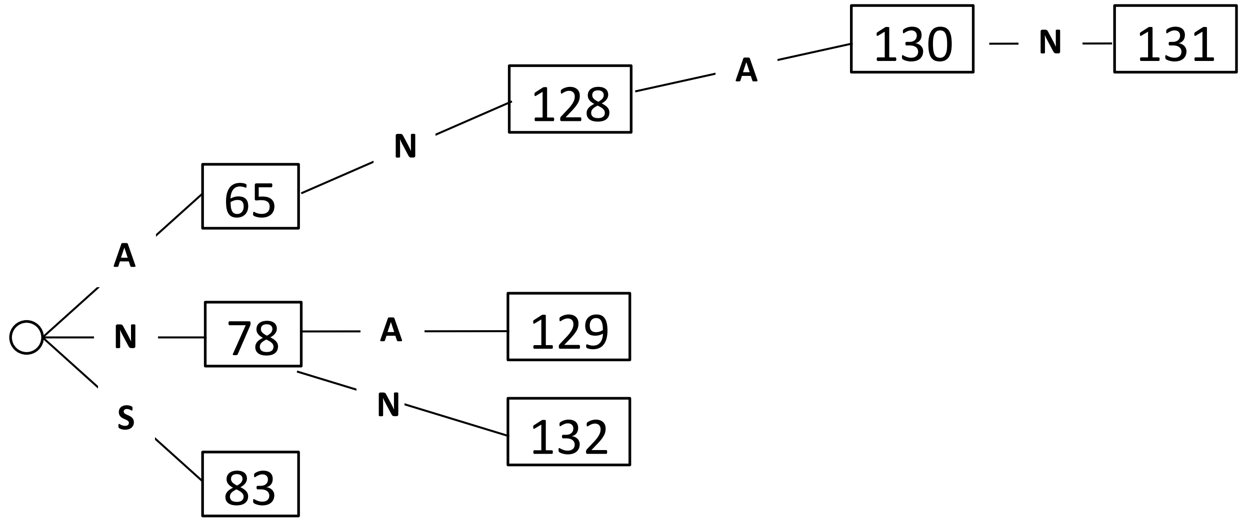
LZW Example



- Dictionary D
  - $idx = 133$

Text	A	N	A N	A N A	N	N A
Encoding	65	78	128	130	78	129

LZW Example



- Dictionary D
  - $idx = 133$

Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130			78	129	

- Final output 000001000001 000001001110 000010000000 000001000010 000001001110 000010000001
- Use fixed length (12 bits) per codenumber
  - 12 bit binary string representation for each code
  - total of $2^{12} = 4096$ codesnumbers available during encoding
    - if you run out of codenumbers, stop inserting new elements in the dictionary

LZW encoding pseudocode

```
LZW::encoding(S, C)
S : input stream of characters, C: output-stream
        initialize dictionary D with ASCII in a trie
        idx ← 128
        while S is not empty do
            v ← root of trie D
            while S is non-empty and v has a child c labelled S.top()   } trie
                v ← c                                                  } search
                S.pop()
            C.append(codenumber stored at v)
            if S is non-empty                                          } new
                create child of v labelled S.top() with code idx       } dictionary
                idx ++                                                 } entry
```

- Running time is $O(|S|)$
  - assuming can look up child labeled with c in $O(1)$ time
    - i.e. trie node stores children in an array

LZW Encoder vs Decoder

- For decoding, need a dictionary
- Construct a dictionary during decoding, imitating what encoder does
- But will be forced to be one step behind
    - at iteration i of decoding can reconstruct substring which encoder inserted into dictionary at iteration $i - 1$
        - delay is due to not having access to the original text

LZW Decoding Example

- Given encoding to decode back to the source text

65	78	128	130	78	129

initial D

65	A
78	N
83	S

$idx = 128$

- Build dictionary adaptively, while decoding
- Decoding starts with the same initial dictionary as encoding
  - use array instead of trie, need D that allows efficient search by code
- We will show the original text during decoding in this example, but just for reference
  - do not need original text to decode

LZW Decoding Example

i=0

- Text: A N A N A N A N N A
- Encoding: 65 78 128 130 78 129
- Decoding: A

iter $i = 0$

$D =$

65	A
78	N
83	S

$idx = 128$

- First step: $s = D(65) =$ A
- Encoding iteration $i = 0$
  - looked ahead in text, saw N, and added AN to D
- Decoding iteration $i = 0$
  - know text starts with A, but cannot look ahead as text is not available
  - no new word added at iteration $i = 0$
  - keep track of s_{prev} = string decoded at previous iteration
    - s_{prev} is also string encoder encoded at previous iteration

LZW Decoding Example

	i=0										
Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130			78	129	
Decoding	A	N (i=1)								

iter $i = 1$

$D =$

65	A
78	**N**
83	S
128	AN

$idx = 129$

- $s_{prev} = \text{A}$
  - string encoded/decoded at previous iteration
- First step: $s = D(78) = \text{N}$
- The first letter of s is the letter the encoder looked ahead at during previous iteration!
- So at previous iteration, encoder added to the dictionary $s_{prev} + s[0]$ (A, N)
- Starting at iteration $i = 1$ of decoding
  - add $s_{prev} + s[0]$ to dictionary

LZW Decoding Example Continued

Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130		78	129		
Decoding	A	N	AN							

i=1, i=2

iter $i = 2$

$D =$

65	A
78	N
83	S
128	**AN**
129	NA

$idx = 130$

- $s_{prev} = \text{N}$
  - string encoded/decoded at previous iteration
- First step: $s = D(128) = \text{AN}$
- Next step: add to dictionary $s_{prev} + s[0]$

  $\text{N} + \text{A} = \text{NA}$

  - encoder added this string at previous iteration

LZW Decoding Example

▪ Text	A	N	A	N	A	N	A	N	N	A
▪ Encoding	65	78	128		130			78	129	
▪ Decoding	A	N	AN		s =???					

iter $i = 3$

$D =$

65	A
78	N
83	S
128	AN
129	NA

$idx = 130$

- $s_{prev} = \text{AN}$
- First step: $s = D(130) =$??? (code 130 is not in D)
  - string encoded/decoded at previous iteration
- Dictionary is exactly one step behind at decoding
- Encoder added (s,130) to D at previous iteration
- Encoder added

$$\underset{\text{known}}{s_{prev}} + \underset{\text{unknown}}{s[0]} = \underset{\text{unknown}}{s}$$

$$\text{AN} + s[0] = s$$

$$s[0] = \textbf{A} = s_{prev}[0]$$

$$\text{AN}\textbf{A} = s$$

$$s = s_{prev} + s_{prev}[0]$$

LZW Decoding Example

			i=2								
▪ Text	A	N	A	N	A	N	A	N	N	A
▪ Encoding	65	78	128		130			78	129	
▪ Decoding	A	N	AN		ANA					

iter $i = 3$

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA

$idx = 131$

- General rule: if code C is not in D
  - $s = s_{prev} + s_{prev}[0]$
- in our example, $s_{prev} = \text{AN}$
  - $s = \text{AN} + \text{A} = \text{ANA}$
- Now that we recovered s, continue as usual
- Add to dictionary $s_{prev} + s[0]$

LZW Decoding Example

▪ Text	A	N	A	N	A	N	A	N	N	A
▪ Encoding	65	78	128		130			78	129	
▪ Decoding	A	N	AN		ANA			N		

iter $i = 4$

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA
131	ANAN

$idx =$ 132

- s_{prev} = ANA
- If code C is not in D

$$s = s_{prev} + s_{prev}[0]$$

- Add to dictionary $s_{prev} + s[0]$

LZW Decoding Example

▪ Text	A	N	A N	A N A	N	N A
▪ Encoding	65	78	128	130	78	129
▪ Decoding	A	N	AN	ANA	N	NA

iter $i = 5$

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA
131	ANAN

$idx = 132$

- $s_{prev} = \text{N}$
- If code C is not in D

  $$s = s_{prev} + s_{prev}[0]$$
- Add to dictionary $s_{prev} + s[0]$

LZW decoding

- To save space, store new codes using its prefix code + one character
  - given a codenumber, can find corresponding string s in $O(|s|)$ time

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA
131	ANAN

wasteful storage

65	A
78	N
83	S
128	65, N
129	78, A
130	128, A
131	130, N

A,N,A,N

65, N,A,N

128, A,N

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128		

next available code

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128		

$s =$ B

nothing added to dictionary at iteration 0

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A

$s_{prev} =$ B, $code_{prev} = 98$

$s =$ A

add to dictionary $s_{prev} + s[0] =$ BA

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R

$s_{prev} =$ A, $code_{prev} = 97$

$s =$ R

add to dictionary $s_{prev} + s[0] =$ AR

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114, B

$s_{prev} =$ R, $code_{prev} = 114$

$s =$ BA

add to dictionary $s_{prev} + s[0] =$ RB

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128, R

s_{prev} = BA , $code_{prev} = 128$

s = R

add to dictionary $s_{prev} + s[0]$ = BAR

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114, A

s_{prev} = R, $code_{prev} = 114$

s = A

add to dictionary $s_{prev} + s[0]$ = RA

LZW decoding, Another Example

Encoding:	98	97	114	128	114	97	131	134	129	101	135
Decoding:	B	A	R	BA	R	A	BAR				

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97, B

$s_{prev} =$ A, $code_{prev} = 97$

$s =$ BAR

add to dictionary $s_{prev} + s[0] =$ AB

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97,B
134	BARB	131, B

$s_{prev} = \text{BAR}, code_{prev} = 131$

$s = ?$

if code is not in dictionary

$s = s_{prev} + s_{prev}[0]$

$s = \text{BAR} + \text{B} = \text{BARB}$

add to dictionary $s_{prev} + s[0] = \text{BARB}$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB AR

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97,B
134	BARB	131,B
135	BARBA	134, A

s_{prev} = BARB, $code_{prev} = 134$

s = AR

add to dictionary $s_{prev} + s[0]$ = BARBA

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB AR E

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97,B
134	BARB	131,B
135	BARBA	134,A
136	ARE	129, E

$s_{prev} =$ AR, $code_{prev} = 129$

$s =$ E

add to dictionary $s_{prev} + s[0] =$ ARE

LZW decoding, Another Example

Encoding:	98	97	114	128	114	97	131	134	129	101	135
Decoding:	B	A	R	BA	R	A	BAR	BARB	AR	E	BARBA

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97,B
134	BARB	131,B
135	BARBA	134,A
136	ARE	129,E

$s_{prev} = \text{E}$

$s = \text{BARBA}$

LZW Decoding Pseudocode

LZW::decoding(C, S)

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

$D \leftarrow$ dictionary that maps $\{0, \ldots, 127\}$ to ASCII

$idx \leftarrow 128$ *// next available code*

$code \leftarrow C.pop(); s \leftarrow D.$*search*$(code); S.append(s)$

while there are more codes in C **do**

$s_{prev} \leftarrow s; code \leftarrow C.pop()$

if *code* $<$ *idx* **then**

$s \leftarrow D.$*search*$(code)$ *//code in D, look up string s*

if *code* $=$ *idx* *// code not in D yet, reconstruct string*

$s \leftarrow s_{prev} + s_{prev}[0]$

else Fail *// invalid encoding*

append each character of s to S

$D.$*insert*$(idx, s_{prev} + s[0])$

$idx{++}$

- Running time is $O(|S|)$

LZW Discussion

- Encoding is $O(|S|)$ time, uses a trie of encoded substrings to store the dictionary
- Decoding is $O(|S|)$ time, uses an array indexed by code numbers to store the dictionary
- Encoding and decoding need to go through the string only one time and do not need to see the whole string
    - can do compression while streaming the text
- Works badly if no repeated substrings
    - dictionary gets bigger, but no new useful substrings inserted
- In practice, compression rate is around 45% on English text

Lempel-Ziv Family

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

- Data Compression
  - Background
  - Single-Character Encodings
  - Huffman Codes
  - Lempel-Ziv-Welch
  - Combining Compression Schemes: bzip2
  - Burrows-Wheeler Transform

Overview of bzip2

- Idea: Combine multiple compression schemes and *text transforms*
  - *text transform*: change input text into a *different text*
    - ouput is not shorter, but likely to compresses better

$T_0 =$ a l f e a t s a l f a l f a

Burrows-Wheeler transform — if T_0 has repeated longer substrings, then T_1 has long runs of characters

$T_1 =$ a f f s $ e f l l l a a a t a

Move-to-front transform — if T_1 has long runs of characters, then T_2 has long runs of 0 and skewed frequencies

text T_2 = 1 3 0 5 3 4 3 5 0 0 6 0 0 6

0-runs encoding — if T_2 has long runs of zeroes, then T_3 is shorter; skewed frequencies remain

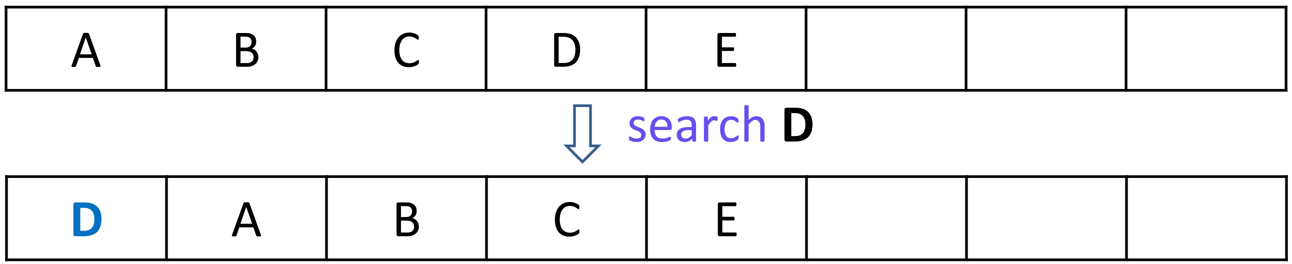
text T_3

Huffman encoding — compresses well since frequencies are skewed

text T_4

Move-to-Front transform

- Recall the MTF heuristic
  - after an element is accessed, move it to array front



- Use this idea for MTF (move to front) text transformation
  - transformed text is likely to have text with repeated zeros and skewed frequencies

MTF Encoding Example

- Source alphabet Σ_S with size $|\Sigma_S| = m$
- Put alphabet in array L, initially in sorted order, but allow L to get unsorted

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

- This gives encoding dictionary L
  - single character encoding E
- Code of any character = index of array where character stored in dictionary L
  - $E(\text{'}B\text{'}) = 1$
  - $E(\text{'}H\text{'}) = 7$
- After each encoding, update L with Move-To-Front heuristic
- Coded alphabet is $\Sigma_C = \{0,1 \ldots, m - 1\}$
- Change dictionary D dynamically (like LZW)
  - unlike LZW
    - no new items added to dictionary
    - codeword for one or more letters can change at each iteration

MTF Encoding 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 = **M**ISSISSIPPI

C =

MTF Encoding 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 = **M**ISSISSIPPI

C = **12**

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

S = ~~M~~ISSISSIPPI

C = 12 **9**

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

S = ~~MI~~**S**SISSIPPI

C = 12 9 **18**

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

S = ~~MISS~~**S**ISSIPPI

C = 12 9 18 **0**

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

S = ~~MISSI~~ISSIPPI

C = 12 9 18 0 **1**

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

S = ~~MISSI~~**S**SIPPI

C = 12 9 18 0 1 **1**

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

S = ~~MISSIS~~**S**IPPI

C = 12 9 18 0 1 1 **0**

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

S = ~~MISSISSIPPI~~

C = 12 9 18 0 1 1 0 1 16 0 1

- What does a run in C mean about the source S?
  - zeros tell us about consecutive character runs

MTF Decoding 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 =

C = 12 9 18 0 1 1 0 1 16 0 1

- Decoding is similar
- Start with the same dictionary D as encoding
- Apply the same MTF transformation at each iteration
    - dictionary D undergoes exactly the transformations when decoding
    - no delays, identical dictionary at encoding and decoding iteration i
    - can always decode original letter

MTF Decoding 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 = M

C = 12 9 18 0 1 1 0 1 16 0 1

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

S = M I

C = 12 9 18 0 1 1 0 1 16 0 1

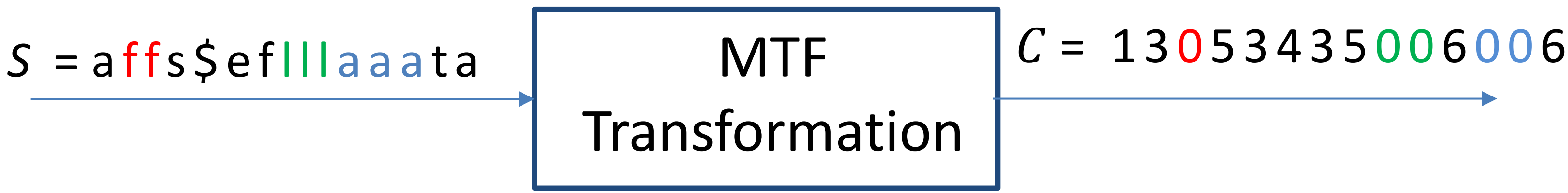
MTF Decoding Example

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

S = M I S

C = 12 9 18 0 1 1 0 1 16 0 1

Move-to-Front Transform: Properties



- If a character in S repeats k times, then C has a run of $k-1$ zeros
- C contains a lot of small numbers and a few big ones
  - skewed frequencies
- C has the same length as S, but better properties for encoding

0-runs Encoding

- Input consists of ‘characters’ in $\{0, \ldots, 127\}$ with long runs of zeros
- Replace k consecutive zeros by $(k)_2$ (takes approximately $\log k$ bits) bits using two new characters A,B
- Example
  - 65, 0, 0, 0, 0 67, 0, 0, 72 becomes 65, A B, 67, B 72
    - actually use bijective binary encoding to save some space

Outline

- Data Compression
  - Background
  - Single-Character Encodings
  - Huffman Codes
  - Lempel-Ziv-Welch
  - Combining Compression Schemes: bzip2
  - Burrows-Wheeler Transform

Burrows-Wheeler Transform

- Transformation (not compression) algorithm
  - transforms source text to coded text with same letters but in different order
    - source and coded alphabets are the same
  - if original text had frequently occurring substrings, transformed text should have many runs of the same character
    - *more suitable for MTF transformation*

$S =$ a l f e a t s a l f a l f a → **Burrows-Wheeler Transform** → C = a f f s $ e f l l l a a a t a

- Required: the source text S ends with *end-of-word character* $
  - $ occurs nowhere else in S
  - count $ towards length of S
- Based on *cyclic* shifts for a string

  - example

string	**cyclic shift**
abcde	cdeab

- Formal definition
  - a *cyclic shift* of string X of length n is the concatenation of $X[i+1 \ldots n-1]$ and $X[0 \ldots i]$, for $0 \leq i < n$

BWT Algorithm and Example

S = `alfeatsalfalfa$`

- Write all consecutive cyclic shifts
  - forms *an array of shifts*
  - last letter in any row is the first letter of the previous row

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Algorithm and Example

$S =$ `alfeatsalfalfa$`

- Array of cyclic shifts
  - first column is the original S
  - each column has same letters as S

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Algorithm and Example

S = a|l f e a t s a|l f a|l f a $

- Array of cyclic shifts
- S has **alf** repeated 3 times
  - 3 different shifts start with **lf** and end with **a**

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Algorithm and Example

S = a|l f e a t s a|l f a|l f a $

- Array of cyclic shifts
- Sort (lexographically) cyclic shifts
  - strict sorting order due to $
- First column (of course) has many consecutive character runs
- But also the last column has many consecutive character runs
  - 3 different shifts start with **lf** and end with **a**
  - sort groups **lf** lines together, and they all end with **a**

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Algorithm and Example

S = **a**|**lf**eatsa**|lfa|lf**a$

- Array of cyclic shifts
- Sort (lexographically) cyclic shifts
  - strict sorting order due to '$'
- First column (of course) has many consecutive character runs
- But also the last column has many consecutive character runs
  - 3 different shifts start with **lf** and end with **a**
  - sort groups **lf** lines together, and they all end with **a**
  - could happen that another pattern will interfere
    - hlfd broken into h and lfd
  - chance of interference is small

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfd      ………     h
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Algorithm and Example

S = `alfeatsalfalfa$`

- Sorted array of cyclic shifts
- First column is useless for encoding
  - cannot decode it
- Last column can be decoded
- BWT Encoding
  - last characters from sorted shifts
    - i.e. the last column

C = **`affs$efllllaaata`**

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Fast Encoding: Efficient Sorting

S = `alfeatsalfalfa$`

i	cyclic shift
0	alfeatsalfalfa$
1	lfeatsalfalfa$a
2	featsalfalfa$al
3	eatsalfalfa$alf
4	atsalfalfa$alfe
5	tsalfalfa$alfea
6	salfalfa$alfeat
7	alfalfa$alfeats
8	lfalfa$alfeatsa
9	falfa$alfeatsal
10	alfa$alfeatsalf
11	lfa$alfeatsalfa
12	fa$alfeatsalfal
13	a$alfeatsalfalf
14	$alfeatsalfalfa

- Refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after $ do not matter

`alfalfa$alfeats`

<

`lfa$alfeatsalfa`

BWT Fast Encoding: Efficient Sorting

S = alfeatsalfalfa$

i	cyclic shift
0	alfeatsalfalfa$
1	lfeatsalfalfa$a
2	featsalfalfa$al
3	eatsalfalfa$alf
4	atsalfalfa$alfe
5	tsalfalfa$alfea
6	salfalfa$alfeat
7	alfalfa$alfeats
8	lfalfa$alfeatsa
9	falfa$alfeatsal
10	alfa$alfeatsalf
11	lfa$alfeatsalfa
12	fa$alfeatsalfal
13	a$alfeatsalfalf
14	$alfeatsalfalfa

- Refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after $ do not matter

lfa$alfeatsalfa

<

lfalfa$alfeatsa

BWT Fast Encoding: Efficient Sorting

S =alfeatsalfalfa$

i	cyclic shift
0	alfeatsalfalfa$
1	lfeatsalfalfa$a
2	featsalfalfa$al
3	eatsalfalfa$alf
4	atsalfalfa$alfe
5	tsalfalfa$alfea
6	salfalfa$alfeat
7	alfalfa$alfeats
8	lfalfa$alfeatsa
9	falfa$alfeatsal
10	alfa$alfeatsalf
11	lfa$alfeatsalfa
12	fa$alfeatsalfal
13	a$alfeatsalfalf
14	$alfeatsalfalfa

- Refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after $ do not matter
- This is the same as sorting suffixes of S
- We already know how to do it
  - exactly as for suffix arrays, with MSD-Radix-Sort
  - $O(n \log n)$ running time

BWT Fast Encoding: Efficient Sorting

S = alfeatsalfalfa$

i	cyclic shift
0	**alfeatsalfalfa$**
1	**lfeatsalfalfa$**a
2	**featsalfalfa$**al
3	**eatsalfalfa$**alf
4	**atsalfalfa$**alfe
5	**tsalfalfa$**alfea
6	**salfalfa$**alfeat
7	**alfalfa$**alfeats
8	**lfalfa$**alfeatsa
9	**falfa$**alfeatsal
10	**alfa$**alfeatsalf
11	**lfa$**alfeatsalfa
12	**fa$**alfeatsalfal
13	**a$**alfeatsalfalf
14	**$**alfeatsalfalfa

j	$A^s[j]$	sorted cyclic shifts
0	14	**$**alfeatsalfalfa
1	13	**a$**alfeatsalfalf
2	10	**alfa$**alfeatsalf
3	7	**alfalfa$**alfeats
4	0	**alfeatsalfalfa$**
5	4	**atsalfalfa$**alfe
6	3	**eatsalfalfa$**alf
7	12	**fa$**alfeatsalfal
8	9	**falfa$**alfeatsal
9	2	**featsalfalfa$**al
10	11	**lfa$**alfeatsalfa
11	8	**lfalfa$**alfeatsa
12	1	**lfeatsalfalfa$**a
13	6	**salfalfa$**alfeat
14	5	**tsalfalfa$**alfea

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

$S =$

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

$A^s =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	10	7	0	4	3	12	9	2	11	8	1	6	5

cyclic shift starts at $S[14]$

need last letter of that cyclic shift, it is at $S[13]$

a

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

$S =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	l	f	e	a	t	s	a	l	f	a	l	**f**	a	$

$A^s =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	10	7	0	4	3	12	9	2	11	8	1	6	5

cyclic shift starts at $S[13]$

need last letter of that cyclic shift, it is at $S[12]$

a f

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

$S =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	l	f	e	a	t	s	a	l	**f**	a	l	f	a	$

$A^s =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	10	7	0	4	3	12	9	2	11	8	1	6	5

cyclic shift starts at $S[10]$

need last letter of that cyclic shift, it is at $S[9]$

a f f

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

$S =$

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

$A^s =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	10	7	0	4	3	12	9	2	11	8	1	6	5

a f f s $ e f l l l a a a t a

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in O

$S =$

0	1	2	3	4	5	6	7	8	9
a	l	f	e	a	t	s	a	l	f

$A^s =$

0	1	2	3	4	5	6	7	8	9
14	13	10	7	0	4	3	12	9	2

j	$A^s[j]$	
0	14	$alfeatsalfalfa
1	13	a$alfeatsalfalf
2	10	alfa$alfeatsalf
3	7	alfalfa$alfeats
4	0	alfeatsalfalfa$
5	4	atsalfalfa$alfe
6	3	eatsalfalfa$alf
7	12	fa$alfeatsalfal
8	9	falfa$alfeatsal
9	2	featsalfalfa$al
10	11	lfa$alfeatsalfa
11	8	lfalfa$alfeatsa
12	1	lfeatsalfalfa$a
13	6	salfalfa$alfeat
14	5	tsalfalfa$alfea

a f f s $ e f l l l a a a t a

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

$S =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	l	f	e	**a**	t	s	a	l	f	a	l	f	a	$

$A^s =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	10	7	0	4	3	12	9	2	11	8	1	6	5

a f f s $ e f l l l a a a t a

- Formula: $C[i] = S[A^s[i] - 1]$
  - array is circular, i.e. $S[-1] = S[n-1]$

BWT Decoding

$C =$`affs$eflllaaata`

- In unsorted shifts array, first column is S
- So decoding = determining the first letter of each row in unsorted shifts array
  - when decoding, do not have unsorted shifts array

unsorted shifts array

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Decoding

$C =$ `affs$eflllaaata`

- Given C, last column of *sorted* shifts array
- Can reconstruct the first column of sorted shifts array by sorting
  - first column has exactly the same characters as the last column
  - *and* they must be sorted

sorted shifts array

```
. . . . . . . . a
. . . . . . . . f
. . . . . . . . f
. . . . . . . . s
. . . . . . . . $
. . . . . . . . e
. . . . . . . . f
. . . . . . . . l
. . . . . . . . l
. . . . . . . . l
. . . . . . . . a
. . . . . . . . a
. . . . . . . . a
. . . . . . . . t
. . . . . . . . a
```

BWT Decoding

$C =$ affs$eflllaaata

- Now have first and last columns of sorted shifts array
- Need the first column of *unsorted* shifts array

unsorted shifts array

$S[0]$	**a** l f e a t s a l f a l f a $
$S[1]$	**l** f e a t s a l f a l f a $ a
$S[2]$	**f** e a t s a l f a l f a $ a l
$S[3]$	**e** a t s a l f a l f a $ a l f
	… … … … … … … … … … … … …

- Where in sorted shifts array are rows $0, 1, \ldots, n-1$ of unsorted shifts array?
- Where is row 0 of unsorted shifts array?

sorted shifts array

$	a
a	f
a	f
a	s
a	$
a	e
e	f
f	l
f	l
f	l
l	a
l	a
l	a
s	t
t	a

BWT Decoding

$C =$ `affs$eflllaaata`

- Now have first and last columns of sorted shifts array
- Need the first column of *unsorted* shifts array

unsorted shifts array

$S[0]$	**a** l f e a t s a l f a l f a $
$S[1]$	**l** f e a t s a l f a l f a $ a
$S[2]$	**f** e a t s a l f a l f a $ a l
$S[3]$	**e** a t s a l f a l f a $ a l f

… … … … … … … … … … … … …

- Where in sorted shifts array are rows $0, 1, \dots, n-1$ of unsorted shifts array?
- Where is row 0 of unsorted shifts array?
  - **must end with $**

sorted shifts array

```
$ . . . . . . . a
a . . . . . . . f
a . . . . . . . f
a . . . . . . . s
a . . . . . . . $   row 0
a . . . . . . . e
e . . . . . . . f
f . . . . . . . l
f . . . . . . . l
f . . . . . . . l
l . . . . . . . a
l . . . . . . . a
l . . . . . . . a
s . . . . . . . t
t . . . . . . . a
```

BWT Decoding

$C =$ `affs$eflllaaata`

$S =$ **a**

- Row 0 of unsorted shifts starts with **a**
- Therefore string S starts with **a**
- Where is row 1 of unsorted shifts array?

unsorted shifts array

row ends with first letter of previous row

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
… … … … … … … … … … … … …
```

- Row 1 ends with the first letter of row 0
  - with **a** in our example

sorted shifts array

```
$ . . . . . . . a
a . . . . . . . f
a . . . . . . . f
a . . . . . . . s
a . . . . . . . $   row 0
a . . . . . . . e
e . . . . . . . f
f . . . . . . . l
f . . . . . . . l
f . . . . . . . l
l . . . . . . . a
l . . . . . . . a
l . . . . . . . a
s . . . . . . . t
t . . . . . . . a
```

BWT Decoding

- Row 1 of unsorted shifts array ends with **a**

sorted shifts array

```
$ . . . . . . . a
a . . . . . . . f
a . . . . . . . f
a . . . . . . . s
a . . . . . . . $   row 0
a . . . . . . . e
e . . . . . . . f
f . . . . . . . l
f . . . . . . . l
f . . . . . . . l
l . . . . . . . a
l . . . . . . . a
l . . . . . . . a
s . . . . . . . t
t . . . . . . . a
```

BWT Decoding

- Row 1 of unsorted shifts array ends with **a**
- Multiple rows end with **a**, which one is row 1 of unsorted shifts?
  - row 1 is a cyclic shift by one of row 0

sorted shifts array

$ . . . . **a**	?
a . . . . f	
a . . . . f	
a . . . . s	
a . . . . $	**row 0**
a . . . . e	
e . . . . f	
f l	
f l	
f l	
l . . . . **a**	?
l . . . . **a**	?
l . . . . **a**	?
s . . . . t	
t . . . . **a**	?

BWT Algorithm and Example

- Multiple rows end with **a**, which one is row 1 of unsorted shifts?
  - row 1 is a cyclic shift by one of row 0
- Rows that end with **a** are cyclic shifts by one of rows that start with **a**
- Rows that start with **a** appear in exactly the same order as their cyclic shifts by 1 (i.e. rows that end with **a**)

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$   row 0
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Algorithm and Example

- for both group of patterns, sorting does not depend on **a**, and all other letters are the same between these two groups

rows starting with **a**	their cyclic shifts by 1
a$alfeatsalfalf	$alfeatsalfalfa
alfa$alfeatsalf	lfa$alfeatsalfa
alfalfa$alfeats	lfalfa$alfeatsa
alfeatsalfalfa$	lfeatsalfalfa$a
atsalfalfa$alfe	tsalfalfa$alfea
row 0 of unsorted shifts is #4	its cyclic shift by one is **also** #4

BWT Algorithm and Example

- Multiple rows end with **a**, which one is row 1 of unsorted shifts?
  - row 1 is a cyclic shift by one of row 0
- Rows that end with **a** are cyclic shifts by one of rows that start with **a**
- Rows that start with **a** appear in exactly the same order as their cyclic shifts by 1 (i.e. rows that end with **a**)
- Direct 'counting' to find row 1 is $O(n)$ time

sorted shifts array

```
  $alfeatsalfalfa 1
1 a$alfeatsalfalf
2 alfa$alfeatsalf
3 alfalfa$alfeats
4 alfeatsalfalfa$ row 0
  atsalfalfa$alfe
  eatsalfalfa$alf
  fa$alfeatsalfal
  falfa$alfeatsal
  featsalfalfa$al
  lfa$alfeatsalfa 2
  lfalfa$alfeatsa 3
  lfeatsalfalfa$a 4 row 1
  salfalfa$alfeat
  tsalfalfa$alfea
```

BWT Algorithm and Example

sorted shifts array

- Form KVP=(letter, row) in the last column, and sort KVPs using stable sort
  - bucket sort, $O(n + |\Sigma_S|)$

```
. . . . . . . . . . a , 0
. . . . . . . . . . f , 1
. . . . . . . . . . f , 2
. . . . . . . . . . s , 3
. . . . . . . . . . $ , 4
. . . . . . . . . . e , 5
. . . . . . . . . . f , 6
. . . . . . . . . . l , 7
. . . . . . . . . . l , 8
. . . . . . . . . . l , 9
. . . . . . . . . . a , 1 0
. . . . . . . . . . a , 1 1
. . . . . . . . . . a , 1 2
. . . . . . . . . . t , 1 3
. . . . . . . . . . a , 1 4
```

BWT Algorithm and Example

- Form KVP=(letter, row) in the last column, and sort KVPs using stable sort
  - bucket sort, $O(n + |\Sigma_S|)$
- Rows for equal letters stay in the same relative order because we used stable sort
- **Row number read in constant time!**

sorted shifts array

$, 4	. . . .	a , 0	
a , 0	. . . .	f , 1	
a , 10	. . . . . .	f , 2	
a , 11	. . . . . .	s , 3	
a , 12	. . . . . .	$, 4	row 0
a , 14	. . . . . .	e , 5	
e , 5	. . . .	f , 6	
f , 1	. . . .	l , 7	
f , 2	. . . .	l , 8	
f , 6	. . . .	l , 9	
l , 7	. . . .	a , 10	
l , 8	. . . .	a , 11	
l , 9	. . . .	a , 12	row 1
s , 3	. . . .	t , 13	
t , 13	. . . . . .	a , 14	

#4 among all rows starting with **a**

#4 among all rows ending with **a**

BWT Decoding Continued

$C =$ affs$eflllaaata

$S =$ a **1**

- Multiple rows end with a, which one is row 1 of unsorted shifts?

sorted shifts array

	$, 4	. . . .	a , 0	
	a , 0	. . . .	f , 1	
	a , 1 0	. . . .	f , 2	
	a , 1 1	. . . .	s , 3	
	a , 1 2	. . . .	$, 4	**row 0**
	a , 1 4	. . . .	e , 5	
	e , 5	. . . .	f , 6	
	f , 1	. . . .	l , 7	
	f , 2	. . . .	l , 8	
	f , 6	. . . .	l , 9	
	l , 7	. . . .	a , 1 0	
	l , 8	. . . .	a , 1 1	
$S[1] =$ **1** $\leftarrow$	**l** , 9	. . . .	a , 1 2	**row 1**
	s , 3	. . . .	t , 1 3	
	t , 1 3	. . . .	a , 1 4	

BWT Decoding Continued

$C =$ affs$efllllaaata

$S =$ al**f**

$S[2] =$ **f** ←

sorted shifts array

$,4	....	a,0	
a,0	....	f,1	
a,10	....	f,2	
a,11	....	s,3	
a,12	....	$,4	row 0
a,14	....	e,5	
e,5	....	f,6	
f,1	....	l,7	
f,2	....	l,8	
f,6	....	l,9	row 2
l,7	....	a,10	
l,8	....	a,11	
l,9	....	a,12	row 1
s,3	....	t,13	
t,13	....	a,14	

BWT Decoding Continued

$C =$ affs$eflllaaata

$S =$ alf**e**

sorted shifts array

	$, 4	. . . .	a , 0	
	a , 0	. . . .	f , 1	
	a , 10	. . . .	f , 2	
	a , 11	. . . .	s , 3	
	a , 12	. . . .	$, 4	**row 0**
	a , 14	. . . .	e , 5	
$S[3]$ = **e** ←	**e** , 5	. . . .	f , 6	**row 3**
	f , 1	. . . .	l , 7	
	f , 2	. . . .	l , 8	
	f , 6	. . . .	l , 9	**row 2**
	l , 7	. . . .	a , 10	
	l , 8	. . . .	a , 11	
	l , 9	. . . .	a , 12	**row 1**
	s , 3	. . . .	t , 13	
	t , 13	. . . .	a , 14	

BWT Decoding Continued

$C =$ affs$eflllaaata

$S =$ alfe**a**

sorted shifts array

	$, 4	. . . .	a , 0	
	a , 0	. . . .	f , 1	
	a , 10	. . . .	f , 2	
	a , 11	. . . .	s , 3	
	a , 12	. . . .	$, 4	**row 0**
$S[4] =$ **a** ←	**a** , 14	. . . .	e , 5	**row 4**
	e , 5	. . . .	f , 6	**row 3**
	f , 1	. . . .	l , 7	
	f , 2	. . . .	l , 8	
	f , 6	. . . .	l , 9	**row 2**
	l , 7	. . . .	a , 10	
	l , 8	. . . .	a , 11	
	l , 9	. . . .	a , 12	**row 1**
	s , 3	. . . .	t , 13	
	t , 13	. . . .	a , 14	

BWT Decoding Continued

$C =$ affs$efllllaaata

$S =$ alfeatsalfalfa$

sorted shifts array

$, 4	. . . .	a , 0	**row 14**
a , 0	. . . .	f , 1	**row 13**
a , 10	. . . .	f , 2	**row 10**
a , 11	. . . .	s , 3	**row 7**
a , 12	. . . .	$, 4	**row 0**
a , 14	. . . .	e , 5	**row 4**
e , 5	. . . .	f , 6	**row 3**
f , 1	. . . .	l , 7	**row 12**
f , 2	. . . .	l , 8	**row 9**
f , 6	. . . .	l , 9	**row 2**
l , 7	. . . .	a , 10	**row 11**
l , 8	. . . .	a , 11	**row 8**
l , 9	. . . .	a , 12	**row 1**
s , 3	. . . .	t , 13	**row 6**
t , 13	. . . .	a , 14	**row 5**

BWT Decoding Pseudocode

```
BWT::decoding(C, S)
C: string of characters over alphabet Σ_C, one of which 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 character (the first aspect)
    for all indices j of C  // find $
        if C[j] = $ break
    do
        S.append(character stored in A[j])
        j ← index stored in A[j]
    while appended character is not $
```

BWT and bzip2 Discussion

- **BWT**
  - encoding cost
    - $O(n \log n)$ with special sorting algorithm
    - read encoding from the suffix array
  - 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)
  - can use on blocks of text (block compression method)
- **bzip2**
  - encoding cost: $O(n\,[\log n + |\Sigma|])$ with a big multiplicative constant
  - decoding cost: $O(n|\Sigma|)$ with a big multiplicative constant
  - 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