Lecture 15: Data Compression III

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: November 21, 2024

Announcements

1. Programming Assignment 2 posted
   - Due 29 November
2. Quiz 6 due Friday
   - Covers Lecture 13 material
   - 1 Question, Short Answer
   - Usual rules apply
3. Attendance Code:

Meeting Goals

1. Give a recap of Lempel-Ziv-Welch (LZW) encoding.
2. Describe LZW decoding procedure
3. Introduce text transformations (non-compressing):
   - Move-to-front (MTF) transform
   - Burrows-Wheeler transform

Lempel-Ziv-Welch Encoding

From Last Time

Lempel-Ziv-Welch encoding idea:

- Fixed length codewords, size k
  - 2^k possible codewords
- Each codeword represents a *string* of text
  - initial codewords correspond to source text alphabet Σ_S (strings of length 1)
- Store a dictionary D that maps strings to codewords
- Encoding scheme:
  - scan source text S sequentially
  - if we see a string xc where string x is in dictionary but xc is not
    - append $D[x]$ to encoded string C
    - add xc to D

Example

Consider $S =$ `N A N A S B A N A N A S`

code	string
0000	A
0001	B
0010	N
0011	S
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	

C =

Example

Consider $S =$ `N A N A S B A N A N A S`

C = `0010`

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	
0110	
0111	
1000	
1001	
1010	
1011	

Example

Consider $S =$ N A N A S B A N A N A S

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	
0111	
1000	
1001	
1010	
1011	

C = 0010 0000

Example

Consider $S =$ N A N A S B A N A N A S

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	
1000	
1001	
1010	
1011	

C = 0010 0000 0100

Example

Consider $S =$ N A N A S B A N A N A S

C = 0010 0000 0100 0011

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	SB
1000	
1001	
1010	
1011	

Example

Consider $S =$ `N A N A S B A N A N A S`

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	SB
1000	BA
1001	
1010	
1011	

C = `0010 0000 0100 0011 0001`

Example

Consider $S =$ `N A N A S B A N A N A S`

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	SB
1000	BA
1001	ANA
1010	
1011	

C = `0010 0000 0100 0011 0001 0101`

Example

Consider $S =$ `N A N A S B A N A N A S`

$C =$ `0010 0000 0100 0011 0001 0101 1001`

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	SB
1000	BA
1001	ANA
1010	ANAS
1011	

Example

Consider $S =$ `N A N A S B A N A N A S`

$C =$ `0010 0000 0100 0011 0001 0101 1001 0011`

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	SB
1000	BA
1001	ANA
1010	ANAS
1011	

LZW in Pseudocode

```
procedure LZWEncode(S[0..n))
    x ← ε                                ▷ previous word, initially empty
    C ← ε                                ▷ output, initially empty
    D ← all c ∈ Σ_S                      ▷ dictionary of codewords
    k ← |Σ_S|                            ▷ next free codeword
    for i = 0, 1, ..., n − 1 do
        c ← S[i]
        if D.ContainsKey(xc) then
            x ← xc
        else
            C ← C D.Get(x)               ▷ append codeword for x
            D.Put(xc, k)
            k ← k + 1, x ← c
        end if
    end for
    C ← C D.Get(x)
end procedure
```

Decoding LZW

Encoding S according to LZW is reasonably simple…

…but how do we **decode** C?

Decoding LZW

Encoding S according to LZW is reasonably simple…

…but how do we **decode** C?

Observations.

- All codewords have length k

Decoding LZW

Encoding S according to LZW is reasonably simple…

…but how do we **decode** C?

Observations.

- All codewords have length k
- *Given* the dictionary D, we can compute the inverse map
  - technically, this depends on the representation of D
  - representing D as a **trie** data structure makes this efficient

Decoding LZW

Encoding S according to LZW is reasonably simple...

...but how do we **decode** C?

Observations.

- All codewords have length k
- *Given* the dictionary D, we can compute the inverse map
  - technically, this depends on the representation of D
  - representing D as a **trie** data structure makes this efficient

Question. How efficient is it to store D?

Decoding LZW

Encoding S according to LZW is reasonably simple…

…but how do we **decode** C?

Observations.

- All codewords have length k
- *Given* the dictionary D, we can compute the inverse map
  - technically, this depends on the representation of D
  - representing D as a **trie** data structure makes this efficient

Question. How efficient is it to store D?

PollEverywhere Question

How many phrases (words) will LZW create on input $S = \texttt{a}^n$ (a run of n as)?

1. $\sim n$
2. $\sim n/2$
3. $\Theta(n/\log n)$
4. $\Theta(\sqrt{n})$
5. $\Theta(\log n)$
6. $\Theta(1)$

pollev.com/comp526

Decoding LZW

Question. How efficient is it to store D?

PollEverywhere Question

How many phrases (words) will LZW create on input $S = \mathtt{a}^n$ (a run of n `as`)?

1. $\sim n$
2. $\sim n/2$
3. $\Theta(n/\log n)$
4. $\Theta(\sqrt{n})$
5. $\Theta(\log n)$
6. $\Theta(1)$

`pollev.com/comp526`

Decoding LZW

Question. How efficient is it to store D?

Important question. Given D could be larger than S for very predictable strings, can we avoid storing D altogether?

Decoding LZW

Question. How efficient is it to store D?

Important question. Given D could be larger than S for very predictable strings, can we avoid storing D altogether?

- **Try:** given C and the start of D (containing only Σ), reconstruct D as we decode

Decoding LZW Example

C = `0010 0000 0100 0011 0001 0101 1001 0011`

code	string
0000	A
0001	B
0010	N
0011	S
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	

$S =$

Decoding LZW Pseudocode

This always works!

procedure LZWDECOMPRESS(C, Σ)
  $D \leftarrow$ dictionary, initialized with codes for $c \in \Sigma_S$ ▷ stored as array
  $k \leftarrow |\Sigma_S|$, $q \leftarrow C[0]$ ▷ first “new” codeword; first codeword
  $y \leftarrow D[q]$ ▷ the first phrase (single character)
  $S \leftarrow y$ ▷ output the first phrase
  for $j = 1, 2, \ldots, |C| - 1$ **do**
    $x \leftarrow y$ ▷ remember decoded phrase
    $q \leftarrow C[j]$ ▷ read next codeword
    if $q = k$ **then**
      $y \leftarrow x x[0]$ ▷ bootstrap case
    else
      $y \leftarrow D[q]$
    end if
    $S \leftarrow Sy$ ▷ append decoded phrase
    $D[k] \leftarrow x y[0]$ ▷ store new phrase
    $k \leftarrow k + 1$
  end for
  return S
end procedure

LZW Discussion

Some Details.

- Implementing the dictionary
  - use a trie data structure (more later…)
- Coded alphabet $\Sigma_C = [0..2^k)$, others are possible (e.g., Huffman)
- What happens when dictionary is full?
  - start using longer codewords?
  - flush dictionary and start from scratch?
- Encoding and decoding both run in linear time! (assuming $|\Sigma_S| = O(1)$)

Appraisal.

- Fast encoding and decoding
- Works in **streaming model**
- Significant compression for many types of data (45% for English text)
- Only captures local repetitions
  - e.g., not maximally helpful for $S = TT$

Compression Summary

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

Text Transforms

Why Settle for What You're Given?

So far we've used the following pipeline:

$$\text{source text} \quad \longrightarrow \quad \text{encoded text}$$

a single algorithm (Huffman, RLE, LZW) to encode the text.

Why Settle for What You're Given?

So far we've used the following pipeline:

source text $\longrightarrow$ encoded text

a single algorithm (Huffman, RLE, LZW) to encode the text.

Inefficiencies:

- Huffman, RLE, and LZW are **limited** in patterns they exploit
  - Huffman: character encoding, changing probabilities
  - RLE: only compresses long runs
  - LZW: only small local repetitions identified

Why Settle for What You're Given?

So far we've used the following pipeline:

source text $\longrightarrow$ encoded text

a single algorithm (Huffman, RLE, LZW) to encode the text.

Inefficiencies:

- Huffman, RLE, and LZW are **limited** in patterns they exploit
  - Huffman: character encoding, changing probabilities
  - RLE: only compresses long runs
  - LZW: only small local repetitions identified

Idea: Text Transformations

- *Reversible* function of text
- does not by itself compress text
- makes text more compressible to algorithms above

Pipeline: source $\longrightarrow$ transformed text $\longrightarrow$ compressed transformed text

Move to Front Transform

Move to Front Heuristic

Simple Observation. In many texts, a recently used character is likely to be reused

- not just globally frequent characters (i.e., Huffman compressible)
- have local runs with repetition

Idea. Start with indexed alphabet Σ (e.g., linked list)

- Whenever a character c is read, move that character to the front of the list

Self Adjusting Lists

Lists

- elements have sequential indexes (like arrays)
- elements can be removed/inserted
  - other elements' indices are shifted

Self-adjustment

- when element c is accessed, move c to the front of list

Self Adjusting Lists

Lists

- elements have sequential indexes (like arrays)
- elements can be removed/inserted
  - other elements' indices are shifted

Self-adjustment

- when element c is accessed, move c to the front of list

PollEverywhere Question

Suppose we start with a list containing `XYZABC` and the next access is to the character `A`. What is the state of the list after the access?

`pollev.com/comp526`

Self Adjusting Lists

Lists

- elements have sequential indexes (like arrays)
- elements can be removed/inserted
  - other elements' indices are shifted

Self-adjustment

- when element *c* is accessed, move *c* to the front of list

PollEverywhere Question

Suppose we start with a list containing `XYZABC` and the next access is to the character `A`. What is the state of the list after the access?

`pollev.com/comp526`

Puzzle. How to implement lists *efficiently*?

Move to Front Simple Example

0	1	2	3	4	5	6	7	8	0	1	2
A	N	A	B	A	N	A	N	A	A	B	N
0									A	B	N

Move to Front Simple Example

0	1	2	3	4	5	6	7	8	0	1	2
A	N	A	B	A	N	A	N	A	A	B	N
0									A	B	N
	2								N	A	B
		1							A	N	B
			2						B	A	N
				1					A	B	N
					2				N	A	B
						1			A	N	B
							1		B	A	N
								1	A	B	N
0	2	1	2	1	2	1	1	1			

Move to Front Simple Example

0	1	2	3	4	5	6	7	8	0	1	2
A	N	A	B	A	N	A	N	A	A	B	N
0									A	B	N
	2								N	A	B
		1							A	N	B
			2						B	A	N
				1					A	B	N
					2				N	A	B
						1			A	N	B
							1		B	A	N
								1	A	B	N
0	2	1	2	1	2	1	1	1			

Observations.

- This process is reasonably efficient
  - store alphabet in a linked list
- This process is **reversible**
  - How?

MTF Heuristic

S	A	N	A	B	A	N	A	N	A
C	0	2	1	2	1	2	1	1	1

Questions.

- What does a *run* in S correspond to in C?

MTF Heuristic

S	A	N	A	B	A	N	A	N	A
C	0	2	1	2	1	2	1	1	1

Questions.

- What does a *run* in S correspond to in C?
  - run in S of length k gives a run of 0s in C of length $k-1$
- What does a run in C correspond to in S?

MTF Code

procedure MTFEncode($S[0..n)$)
  $L \leftarrow$ list containing Σ_S (sorted)
  $C \leftarrow \varepsilon$
  for $i = 0, 1, \ldots, n-1$ **do**
    $c \leftarrow S[i]$
    $p \leftarrow$ position of c in L
    $C \leftarrow C\,p$
    Move c to front of L
  end for
  return C
end procedure

MTF Code

```
procedure MTFEncode(S[0..n))
    L ← list containing Σ_S (sorted)
    C ← ε
    for i = 0, 1, ..., n − 1 do
        c ← S[i]
        p ← position of c in L
        C ← C p
        Move c to front of L
    end for
    return C
end procedure
```

```
procedure MTFDecode(C[0..m))
    L ← list containing Σ_S (sorted)
    S ← ε
    for j = 0, 1, ..., m − 1 do
        p ← S[i]
        c ← char at position p in L
        S ← S c
        Move c to front of L
    end for
    return S
end procedure
```

MTF Code

procedure MTFENCODE($S[0..n)$)
  $L \leftarrow$ list containing Σ_S (sorted)
  $C \leftarrow \varepsilon$
  for $i = 0, 1, \ldots, n-1$ **do**
    $c \leftarrow S[i]$
    $p \leftarrow$ position of c in L
    $C \leftarrow C\,p$
    Move c to front of L
  end for
  return C
end procedure

procedure MTFDECODE($C[0..m)$)
  $L \leftarrow$ list containing Σ_S (sorted)
  $S \leftarrow \varepsilon$
  for $j = 0, 1, \ldots, m-1$ **do**
    $p \leftarrow S[i]$
    $c \leftarrow$ char at position p in L
    $S \leftarrow S\,c$
    Move c to front of L
  end for
  return S
end procedure

Key observation. Both encode and decode procedure perform same sequence of accesses to L

$\Longrightarrow$ the decoded letter is always the same as encoded (induction)

MTF Discussion

- MTF does not compress texts (assuming we use fixed-length codewords)
  - MTF is used as part of a longer pipeline
  - For many texts smaller codeword values are more likely after MTF

MTF Discussion

- MTF does not compress texts (assuming we use fixed-length codewords)
  - MTF is used as part of a longer pipeline
  - For many texts smaller codeword values are more likely after MTF
- Intuitive effect: MTF converts patterns with low *local entropy* to texts with small *global entropy*
  - Huffman is more effective after MTF transform applied

MTF Discussion

- MTF does not compress texts (assuming we use fixed-length codewords)
  - MTF is used as part of a longer pipeline
  - For many texts smaller codeword values are more likely after MTF
- Intuitive effect: MTF converts patterns with low *local entropy* to texts with small *global entropy*
  - Huffman is more effective after MTF transform applied
- Still, many natural texts do not have low local entropy

  … but we can try to transform texts so that they *do* have low local entropy!

Burrows-Wheeler Transform

Burrows-Wheeler Transform

A Sophisticated Transform.

- Coded text has same characters as source, but in a different order
- Coded text is typically more compressible (local character frequencies)

Burrows-Wheeler Transform

A Sophisticated Transform.

- Coded text has same characters as source, but in a different order
- Coded text is typically more compressible (local character frequencies)

A Concession

- Encoding requires access to *all* of S
    - not streaming like LZW, or "two pass" like Huffman

Burrows-Wheeler Transform

A Sophisticated Transform.

- Coded text has same characters as source, but in a different order
- Coded text is typically more compressible (local character frequencies)

A Concession

- Encoding requires access to *all* of S
  - not streaming like LZW, or "two pass" like Huffman

An Effective Pipeline: BWT ⟶ MTF ⟶ RLE ⟶ Huffman

- Used by `bzip2` compression program:

```
5458199 21 Nov 12:06 shakespeare-original.txt
1479261 21 Nov 12:04 shakespeare.txt.bz2
2024091 21 Nov 12:08 shakespeare.txt.gz
2022556 21 Nov 12:06 shakespeare.txt.zip
```

Cyclic Shifts

Definition. A **cyclic shift** of a string S is a re-indexing of S by some fixed offset with wraparound

- add terminating character \$ to show original end of S
- \$ is alphabetically before all other characters

Cyclic Shifts

Definition. A **cyclic shift** of a string *S* is a re-indexing of *S* by some fixed offset with wraparound

- add terminating character $ to show original end of *S*
- $ is alphabetically before all other characters

Example. All cyclic shifts of *S* = `alf␣eats␣alfalfa$`:

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

Burrows Wheeler Transform

A Simple Idea.

1. Form all cyclic shifts of S

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

Burrows Wheeler Transform

A Simple Idea.

1. Form all cyclic shifts of S
2. Sort the shifts alphabetically

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

$\rightsquigarrow$ sort

```
$alf␣eats␣alfalfa
␣alfalfa$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
```

Burrows Wheeler Transform

A Simple Idea.

1. Form all cyclic shifts of S
2. Sort the shifts alphabetically
3. Return the last **column** of the table

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

⇝ sort

BWT ↓

```
$alf␣eats␣alfalfa
␣alfalfa$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
```

Features of BWT

BWT:

1. Form all cyclic shifts of S
2. Sort the shifts alphabetically
3. Return the last **column** of the table

$S =$ `alf␣eats␣alfalfa$`

$B =$ `asff$f␣e␣lllaaata`

Remarkably:

- This procedure can be computed in $O(n)$ time
  - "naive" algorithm/analysis is $O(n^2 \log n)$
- This procedure is reverseable!
  - we will see this soon

```
$alf␣eats␣alfalfa
␣alfalfa$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
```

What does BWT do?

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

Observation 1. BWT contains the same characters as S

- characters are reordered

What does BWT do?

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

Observation 2. BWT groups characters by what follows

- repeated substrings give rise to runs in B
- a always followed by `lf` $\implies$ B contains a run of as

What does BWT do?

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

Observation 3. BWT outputs are typically amenable to compression after applying MTF

Larger BWT Example

```
have␣had␣hadnt␣hasnt␣havent␣has␣what$
ave␣had␣hadnt␣hasnt␣havent␣has␣what$h
ve␣had␣hadnt␣hasnt␣havent␣has␣what$ha
e␣had␣hadnt␣hasnt␣havent␣has␣what$hav
␣had␣hadnt␣hasnt␣havent␣has␣what$have
had␣hadnt␣hasnt␣havent␣has␣what$have␣
ad␣hadnt␣hasnt␣havent␣has␣what$have␣h
d␣hadnt␣hasnt␣havent␣has␣what$have␣ha
␣hadnt␣hasnt␣havent␣has␣what$have␣had
hadnt␣hasnt␣havent␣has␣what$have␣had␣
adnt␣hasnt␣havent␣has␣what$have␣had␣h
dnt␣hasnt␣havent␣has␣what$have␣had␣ha
nt␣hasnt␣havent␣has␣what$have␣had␣had
t␣hasnt␣havent␣has␣what$have␣had␣hadn
␣hasnt␣havent␣has␣what$have␣had␣hadnt
hasnt␣havent␣has␣what$have␣had␣hadnt␣
asnt␣havent␣has␣what$have␣had␣hadnt␣h
snt␣havent␣has␣what$have␣had␣hadnt␣ha
nt␣havent␣has␣what$have␣had␣hadnt␣has
t␣havent␣has␣what$have␣had␣hadnt␣hasn
␣havent␣has␣what$have␣had␣hadnt␣hasnt
havent␣has␣what$have␣had␣hadnt␣hasnt␣
avent␣has␣what$have␣had␣hadnt␣hasnt␣h
vent␣has␣what$have␣had␣hadnt␣hasnt␣ha
ent␣has␣what$have␣had␣hadnt␣hasnt␣hav
nt␣has␣what$have␣had␣hadnt␣hasnt␣have
t␣has␣what$have␣had␣hadnt␣hasnt␣haven
␣has␣what$have␣had␣hadnt␣hasnt␣havent
has␣what$have␣had␣hadnt␣hasnt␣havent␣
as␣what$have␣had␣hadnt␣hasnt␣havent␣h
s␣what$have␣had␣hadnt␣hasnt␣havent␣ha
␣what$have␣had␣hadnt␣hasnt␣havent␣has
what$have␣had␣hadnt␣hasnt␣havent␣has␣
hat$have␣had␣hadnt␣hasnt␣havent␣has␣w
at$have␣had␣hadnt␣hasnt␣havent␣has␣wh
t$have␣had␣hadnt␣hasnt␣havent␣has␣wha
$have␣had␣hadnt␣hasnt␣havent␣has␣what
```

```
$have␣had␣hadnt␣hasnt␣havent␣has␣wha t
␣had␣hadnt␣hasnt␣havent␣has␣what$hav e
␣hadnt␣hasnt␣havent␣has␣what$have␣ha d
␣has␣what$have␣had␣hadnt␣hasnt␣haven t
␣hasnt␣havent␣has␣what$have␣had␣hadn t
␣havent␣has␣what$have␣had␣hadnt␣hasn t
␣what$have␣had␣hadnt␣hasnt␣havent␣ha s
ad␣hadnt␣hasnt␣havent␣has␣what$have␣ h
adnt␣hasnt␣havent␣has␣what$have␣had␣ h
as␣what$have␣had␣hadnt␣hasnt␣havent␣ h
asnt␣havent␣has␣what$have␣had␣hadnt␣ h
at$have␣had␣hadnt␣hasnt␣havent␣has␣w h
ave␣had␣hadnt␣hasnt␣havent␣has␣what$ h
avent␣has␣what$have␣had␣hadnt␣hasnt␣ h
d␣hadnt␣hasnt␣havent␣has␣what$have␣h a
dnt␣hasnt␣havent␣has␣what$have␣had␣h a
e␣had␣hadnt␣hasnt␣havent␣has␣what$ha v
ent␣has␣what$have␣had␣hadnt␣hasnt␣ha v
had␣hadnt␣hasnt␣havent␣has␣what$have ␣
hadnt␣hasnt␣havent␣has␣what$have␣had ␣
has␣what$have␣had␣hadnt␣hasnt␣havent ␣
hasnt␣havent␣has␣what$have␣had␣hadnt ␣
hat$have␣had␣hadnt␣hasnt␣havent␣has␣ w
have␣had␣hadnt␣hasnt␣havent␣has␣what $
havent␣has␣what$have␣had␣hadnt␣hasnt ␣
nt␣has␣what$have␣had␣hadnt␣hasnt␣hav e
nt␣hasnt␣havent␣has␣what$have␣had␣ha d
nt␣havent␣has␣what$have␣had␣hadnt␣ha s
s␣what$have␣had␣hadnt␣hasnt␣havent␣h a
snt␣havent␣has␣what$have␣had␣hadnt␣h a
t$have␣had␣hadnt␣hasnt␣havent␣has␣wh a
t␣has␣what$have␣had␣hadnt␣hasnt␣have n
t␣hasnt␣havent␣has␣what$have␣had␣had n
t␣havent␣has␣what$have␣had␣hadnt␣has n
ve␣had␣hadnt␣hasnt␣havent␣has␣what$h a
vent␣has␣what$have␣had␣hadnt␣hasnt␣h a
what$have␣had␣hadnt␣hasnt␣havent␣has ␣
```

T = have␣had␣hadnt␣hasnt␣havent␣has␣what$

B = tedtttshhhhhhhaavv␣␣␣␣w$␣edsaaannnaa␣

$MTF(B)$ = 8 5 5 2 0 0 8 7 0 0 0 0 0 0 7 0 9 0 8 0 0 0 10 9 2 9 9 8 7 0 0 10 0 0 1 0 5

Inverting BWT

Observation. Compression means nothing unless we can invert the procedure!

- What if we just sorted the characters of S directly an compressed the sorted string?

Inverting BWT

Observation. Compression means nothing unless we can invert the procedure!

A Magic Trick.

1. Create an array D of pairs $(B[r], r)$
2. Sort D *stably* with respect to first entry
3. Interpret D as linked list (follow links!)

Inverting BWT

Observation. Compression means nothing unless we can invert the procedure!

A Magic Trick.

1. Create an array D of pairs $(B[r], r)$
2. Sort D *stably* with respect to first entry
3. Interpret D as linked list (follow links!)

Example.

$B =$ `ard$rcaaaabb`

1: $(a, 1)$
2: $(r, 2)$
3: $(d, 3)$
4: $(\$, 4)$
5: $(r, 5)$
6: $(c, 6)$
7: $(a, 7)$
8: $(a, 8)$
9: $(a, 9)$
10: $(a, 10)$
11: $(b, 11)$
12: $(b, 12)$

Inverting BWT

Observation. Compression means nothing unless we can invert the procedure!

A Magic Trick.

1. Create an array D of pairs $(B[r], r)$
2. Sort D *stably* with respect to first entry
3. Interpret D as linked list (follow links!)

Example.

$B =$ `ard$rcaaaabb`

1:	$(a, 1)$	1:	$(\$, 4)$
2:	$(r, 2)$	2:	$(a, 1)$
3:	$(d, 3)$	3:	$(a, 7)$
4:	$(\$, 4)$	4:	$(a, 8)$
5:	$(r, 5)$	5:	$(a, 9)$
6:	$(c, 6)$	6:	$(a, 10)$
7:	$(a, 7)$	7:	$(b, 11)$
8:	$(a, 8)$	8:	$(b, 12)$
9:	$(a, 9)$	9:	$(c, 6)$
10:	$(a, 10)$	10:	$(d, 3)$
11:	$(b, 11)$	11:	$(r, 2)$
12:	$(b, 12)$	12:	$(r, 5)$

Inverting BWT

Observation. Compression means nothing unless we can invert the procedure!

A Magic Trick.

1. Create an array D of pairs $(B[r], r)$
2. Sort D *stably* with respect to first entry
3. Interpret D as linked list (follow links!)

Example.

$B =$ `ard$rcaaaabb`

1:	$(a, 1)$	1:	$(\$, 4)$
2:	$(r, 2)$	2:	$(a, 1)$
3:	$(d, 3)$	3:	$(a, 7)$
4:	$(\$, 4)$	4:	$(a, 8)$
5:	$(r, 5)$	5:	$(a, 9)$
6:	$(c, 6)$	6:	$(a, 10)$
7:	$(a, 7)$	7:	$(b, 11)$
8:	$(a, 8)$	8:	$(b, 12)$
9:	$(a, 9)$	9:	$(c, 6)$
10:	$(a, 10)$	10:	$(d, 3)$
11:	$(b, 11)$	11:	$(r, 2)$
12:	$(b, 12)$	12:	$(r, 5)$

For Next Time. Convince yourself this inverts the BWT!

BWT Discussion

What do we know about the Burrows-Wheeler Transform?

- Running time $\Theta(n)$
  - encoding uses **suffix sorting** (future reference)
  - decoding can be done in $\Theta(n)$ time with counting sort
  - decoding is simpler/faster!
- Typically slower than other methods
- Needs access to entire text (or apply to smaller blocks)
- WBT → MTF → RLE → Huffman has great compression!

Summary of Compression

Huffman	Variable-width, single-character (optimal in this case)
RLE	Variable-width, multiple-character encoding
LZW	Adaptive, fixed-width, multiple-character encoding Augments dictionary with repeated substrings
MTF	Adaptive, transforms to smaller integers should be followed by variable-width integer encoding
BWT	Block compression method, should be followed by MTF

Next Time

Error Correcting Codes

Scratch Notes