# Text Compression

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

# Properties of the BWT

| *bwt* | *sorted suffixes* |
|---|---|
| d | ie BWT lässt sich ohne |
| d | ie Indizes der sortierten |
| d | ie LMS-Substrings bilden |
| d | ie hinterste freie Stelle |
| d | ie selbe Anordnung |
| d | ie selbe Länge |
| d | ie sortierte Reihenfolge der |
| d | iese ursprüngliche Definition |
| d | ieser Ansatz hat eine |

- In natural language texts, certain combinations of letters appear frequently, e.g. 'die', 'sch', 'tz', or 'ie' (in German).
- Then the BWT contains long runs of the same character, such as . . . dddddddddd. . . , as shown.
- Informally, **repeats** in the original text become **runs** in the BWT.
- Compression techniques have an "easier job" on the BWT.
- Example: **run length encoding**

# Run Length Encoding

`bwt` = `AAAAAAACCCCGGGGGGAAATT`

**Run length encoding (RLE)**

replaces runs of the same character $c$ with a pair (count, $c$).

**Example:**

Run length encoded `bwt`: `7A4C6G3A2T`

# Run Length Encoding

bwt = `AAAAAAACCCCGGGGGGAAATT`

**Run length encoding (RLE)**

replaces runs of the same character $c$ with a pair (count, $c$).

**Example:**

Run length encoded `bwt`: `7A4C6G3A2T`

Variations of RLE exist that avoid wasting space for short runs.

# bzip2

The tool bzip2 by Julian Seward is based on the BWT.
The input is a file (sequence of bytes) and a block size.
It processes each block separately, i.e., repeats across blocks are not exploited.

# bzip2

The tool bzip2 by Julian Seward is based on the BWT.
The input is a file (sequence of bytes) and a block size.
It processes each block separately, i.e., repeats across blocks are not exploited.

## Processing (simplified, abbreviated)

For each block separately:

1. Compute the BWT.
2. Apply move-to-front (MTF) transformation.
   The resulting sequence has many zeros and small values.
3. Apply a variant of run length encoding (RLE)
4. Apply Huffman coding:
   Represent frequent symbols by short bit sequences, rare symbols by longer ones.

All steps are invertible.

# Move-to-Front Encoding (Example `tttt$aaac`)

| i | list | S [i] | R [i] |
|---|---|---|---|
| 0 | $act | t | 3 |
| 1 | t$ac | t | 0 |
| 2 | t$ac | t | 0 |
| 3 | t$ac | t | 0 |
| 4 | t$ac | $ | 1 |
| 5 | $tac | a | 2 |
| 6 | a$tc | a | 0 |
| 7 | a$tc | a | 0 |
| 8 | a$tc | c | 3 |

- Current character *a* is encoded by its index current alphabet list.
- Then *a* is moved to the front of the list.

- Runs of different characters become runs of zeros.
- Small (local) character set in BWT: small numbers
- Encoded sequence has high frequency of small numbers like 0, 1,2, . . .

# Move-to-Front Decoding

Decode (3,0,0,0,1,2,0,0,3) for $\Sigma = \{\$, \mathtt{a}, \mathtt{c}, \mathtt{t}\}$

# Huffman Coding

MTF encoding works best together with frequency-based compression,
like **Huffman Coding (Huffman trees)**, which produce **optimal prefix-free codes**:
Represent frequent symbols by short bit sequences, rare symbols by longer sequences.

# Huffman Coding

MTF encoding works best together with frequency-based compression,
like **Huffman Coding (Huffman trees)**, which produce **optimal prefix-free codes**:
Represent frequent symbols by short bit sequences, rare symbols by longer sequences.

## Prefix-free?

No code word is a prefix of another code word.
This is essential for correct and easy decoding.

## Optimality

Let $T$ be an input string. Let $f_c$ be the relative frequency of character $c$ in $T$.
Let $e_c$ be the chosen bit vector coding for $c$.
Then the average bit length per character is $L(e) := \sum_c f_c \cdot |e_c|$.

Among all prefix-free codes $e : \Sigma \to \{0,1\}^+$, find one with minimal $L(e)$.

# Huffman Coding: Algorithm

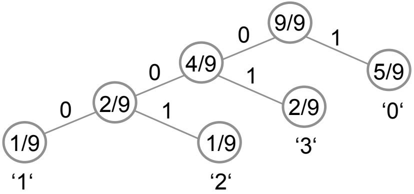
- Initially **each leaf denotes a character** $c$ in $\Sigma$ and has weight $f_c$.
- Repeat until a single node (with weight 1.0) remains:
  - Pick the two nodes with smallest weights.
  - Connect them by a new node, their common parent, whose weight is the sum of the children's weights.
  - The edge to the left child is labeled 0, the edge to the right child 1.
  - Remove the children from the nodes under consideration.
- The code word (bit sequence) for $c$ is spelled by the labels on the path from the root to $c$.

# Example: Huffman Coding

Huffman tree for string $T = 300012003$

# Example: Huffman Coding

$T = 300012003$
code $= 011110000011101$ (15 bits instead of 18 for fixed-length code)




| character | '0' | '1' | '2' | '3' |
|---|---|---|---|---|
| frequency | 5/9 | 1/9 | 1/9 | 2/9 |
| Huffman Code | 1 | 000 | 001 | 01 |
| fixed-length code | 00 | 01 | 10 | 11 |

# Lempel-Ziv Factorizations

# LZ77

## Factorization

A **factorization** of a string $T$ is a non-overlapping partitioning of $T$ into substrings:

$$T = f_1 f_2 \dots f_z$$

The number of factors (substrings) is typically called $z$.

# LZ77

## Factorization

A **factorization** of a string $T$ is a non-overlapping partitioning of $T$ into substrings:

$$T = f_1 f_2 \dots f_z$$

The number of factors (substrings) is typically called $z$.

## Lempel-Ziv 77

The LZ77 factorization is defined as follows: Each factor $f_i$ is either

1. a single character that has not appeared in $f_1 \dots f_{i-1}$, or
2. the longest substring occurring at least twice in $f_1 \dots f_i$.

(The possible "overlap" of $f_i$ with itself is desired!)

# LZ77

## Factorization

A **factorization** of a string $T$ is a non-overlapping partitioning of $T$ into substrings:

$$T = f_1 f_2 \dots f_z$$

The number of factors (substrings) is typically called $z$.

## Lempel-Ziv 77

The LZ77 factorization is defined as follows: Each factor $f_i$ is either

1. a single character that has not appeared in $f_1 \dots f_{i-1}$, or
2. the longest substring occurring at least twice in $f_1 \dots f_i$.

(The possible "overlap" of $f_i$ with itself is desired!)

## Representation

An LZ77 factor $f_i$ is written $(\ell_i, p_i)$ with length $\ell_i$ and starting position $p_i$.
In the single character case $\ell_i = 0$, and $p_i$ is the (ASCII) character code.

# LZ77 Example

Each factor $f_i$ is either

1. a single character that has not appeared in $f_1 \ldots f_{i-1}$, or
2. the longest substring occurring at least twice in $f_1 \ldots f_i$.

```
    0         1
p:  012345678901234
s:  aacaacabcabaaac
```

# LZ77 Example

Each factor $f_i$ is either

1. a single character that has not appeared in $f_1 \ldots f_{i-1}$, or
2. the longest substring occurring at least twice in $f_1 \ldots f_i$.

```
   0         1
p: 012345678901234
s: aacaacabcabaaac

   a|a|c|aaca|b|cab|aa|ac
```

# LZ77 Example

Each factor $f_i$ is either

1. a single character that has not appeared in $f_1 \ldots f_{i-1}$, or
2. the longest substring occurring at least twice in $f_1 \ldots f_i$.

```
   0         1
p: 012345678901234
s: aacaacabcabaaac

   a|a|c|aaca|b|cab|aa|ac

   (0,a)(1,0)(0,c)(4,0)(0,b)(3,5)(2,0)(2,1)
```

# Computation of the LZ77 Factorization

Using a Suffix Tree of $T$, assuming constant alphabet size!

## Preparation

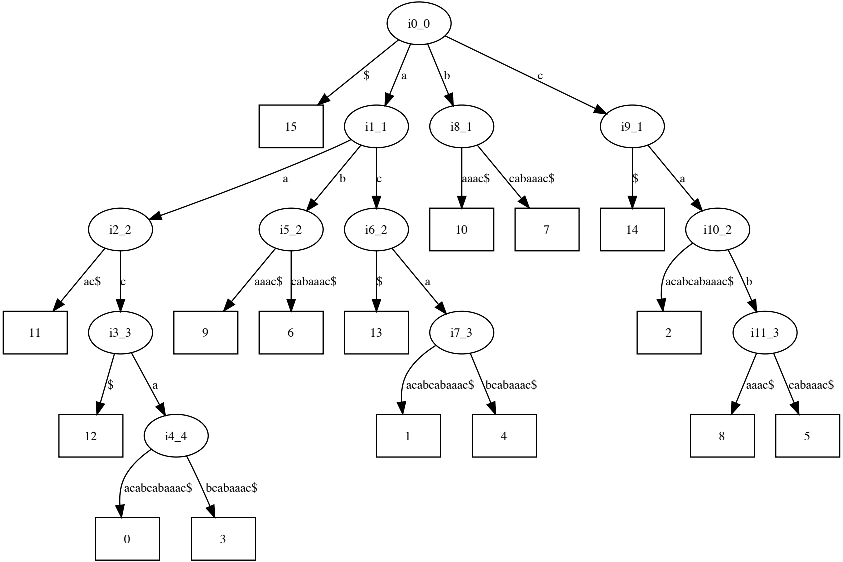
Annotate each internal node with the smallest leaf number (position) below it.
Time: $O(n)$, bottom-up

## Factorization

- When starting a new factor at position $p$, follow a path from the root spelling $T[p\dots]$, as long as leaves with numbers $< p$ are present below.
- When you are still in the root, insert single character $T[p]$
- When you are in an inner node, insert $(d, s)$, where $d$ is the current string depth, and $s$ is the smallest leaf number below you.

Time: $O(n)$, one step down per character (for constant alphabet)

# Example: aacaacabcabaaac$



# LZ78

### Factorization

A **factorization** of a string $T$ is a non-overlapping partitioning of $T$ into substrings:

$$T = f_1 f_2 \ldots f_z$$

# LZ78

### Factorization

A **factorization** of a string $T$ is a non-overlapping partitioning of $T$ into substrings:

$$T = f_1 f_2 \dots f_z$$

### Lempel-Ziv 78 Factorization

Let $f_0 := \varepsilon$ (empty string). For $i \geq 1$, if $f_1 \dots f_{i-1} = T[0 \dots (j-1)]$,
let $f_i$ be the longest prefix of $T[j \dots]$ such that $f_i = f_k a$ for some $k < i$ and $a \in \Sigma$.

# LZ78

## Factorization

A **factorization** of a string $T$ is a non-overlapping partitioning of $T$ into substrings:

$$T = f_1 f_2 \dots f_z$$

## Lempel-Ziv 78 Factorization

Let $f_0 := \varepsilon$ (empty string). For $i \geq 1$, if $f_1 \dots f_{i-1} = T[0 \dots (j-1)]$,
let $f_i$ be the longest prefix of $T[j \dots]$ such that $f_i = f_k a$ for some $k < i$ and $a \in \Sigma$.

## Representation

An LZ78 factor $f_i$ is written $(k_i, a_i)$ with index $0 \leq k_i < i$ and character $a_i \in \Sigma$.
A single character $b$ is thus written $(0, b)$.

# LZ78 Example

Let $f_0 := \varepsilon$ (empty string). For $i \geq 1$, if $f_1 \dots f_{i-1} = T[1 \dots (j-1)]$, let $f_i$ be the longest prefix of $T[j \dots]$ such that $f_i = f_k a$ for some $k < i$ and $a \in \Sigma$.

```
    0         1
p:  012345678901234
s:  aacaacabcabaaac
```

# LZ78 Example

Let $f_0 := \varepsilon$ (empty string). For $i \geq 1$, if $f_1 \dots f_{i-1} = T[1 \dots (j-1)]$, let $f_i$ be the longest prefix of $T[j \dots]$ such that $f_i = f_k a$ for some $k < i$ and $a \in \Sigma$.

```
    0         1
p:  012345678901234
s:  aacaacabcabaaac

     |a|ac|aa|c|ab|ca|b|aaa|c
i:  0 1 2  3  4 5  6  7 8
ka:   0a1c 1a 0c1b 4a 0b3a  4.
```

# Computation of the LZ78 Factorization

Using a trie of existing factors, assuming constant alphabet size!

## Preparation

Initialize a trie consisting only of the root; mark the root as factor $i = 0$ (empty string).

## Factorization

- When starting a new factor $f_i$ at text position $p$, follow a path from the root spelling $T[p \dots]$ as far as possible.
- Attach a new leaf (marked $i$) to the current node (marked with some $k < i$), with an edge labeled $a := T[p + d]$ where $d$ is the current string depth in the trie.
- Output $(k, a)$

Time: $O(n)$, one step down per character (for constant alphabet)

# Example

```
    0         1
p:  012345678901234
s:  aacaacabcabaaac

     |a|ac|aa|c|ab|ca|b|aaa|c
i:  0 1 2  3  4 5  6  7 8
ka:   0a1c 1a 0c1b 4a 0b3a  4.
```

# Remarks on LZ factorizations

- LZ77 and LZ78 invented by Lempel and Ziv in 1977 and 1978 (also LZ1, LZ2)
- Simplified and streamlined description given here!
- Many variations exist:
  In fact, LZ77 uses a sliding window and forgets text to the far left
- We assume constant alphabet (finding correct outgoing edge in $O(1)$ time); more complex algorithms needed for poly($n$) alphabet.
- Still, LZ factorizations are the foundation for many compression methods: ZIP, GIF, PNG, ...
- Patent issues (till 2004) with LZ78, but LZ77 was always patent-free

# Summary

## Compression

- Why the BWT is useful for compression
- Run Length Encoding
- Move-to-Front encoding
- Huffman coding
- LZ77 factorization and computation via suffix tree
- LZ78 factorization and computation via simple trie

## Note

Finding good application-specific compression algorithms is valuable:

`https://www.illumina.com/company/news-center/feature-articles/illumina-acquires-enancio-s-compression-software.html`

# Exam Questions

- Why can the BWT be easier to compress than the input string?
- What is run length encoding?
- Explain Move-to-Front encoding. Apply it to an example.
- Explain Huffman coding.
- Define the LZ77 factorization.
- How do you efficiently compute the LZ77 factorization?
- Define the LZ78 factorization.
- How do you efficiently compute the LZ78 factorization?
- What if the alphabet is not of constant size, but grows as e.g., $\sqrt{n}$:
  How does the time of the LZ77 and LZ78 factorization algorithms change?