

# Text Indexing

## Lecture 05: Burrows-Wheeler Transform

Florian Kurpicz

The slides are licensed under a Creative Commons Attribution-ShareAlike 4.0 International License © ⓘ ⓘ: [www.creativecommons.org/licenses/by-sa/4.0](https://www.creativecommons.org/licenses/by-sa/4.0) | commit 224e27c compiled at 2022-11-28-12:28



<https://pingo.scc.kit.edu/886630>

# Recap: Text-Compression

## Definition: LZ77 Factorization [ZL77]

Given a text  $T$  of length  $n$  over an alphabet  $\Sigma$ , the **LZ77 factorization** is

- a set of  $z$  factors  $f_1, f_2, \dots, f_z \in \Sigma^+$ , such that
- $T = f_1 f_2 \dots f_z$  and for all  $i \in [1, z]$   $f_i$  is
- single character not occurring in  $f_1 \dots f_{i-1}$  or
- longest substring occurring  $\geq 2$  times in  $f_1 \dots f_i$

$T = \text{abababbbbaba\$}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = abab$
- $f_4 = bbb$
- $f_5 = aba$
- $f_6 = \$$

## Definition: LZ78 Factorization [ZL78]

Given a text  $T$  of length  $n$  over an alphabet  $\Sigma$ , the **LZ78 factorization** is

- a set of  $z$  factors  $f_1, f_2, \dots, f_z \in \Sigma^+$ , such that
- $T = f_1 f_2 \dots f_z$ ,  $f_0 = \epsilon$  and for all  $i \in [1, z]$
- if  $f_1 \dots f_{i-1} = T[1..j-1]$ , then  $f_i$  is the longest prefix of  $T[j..n]$ , such that

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\}$$
:  $f_k = f_i \alpha$

$T = \text{abababbbbaba\$}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = ab$
- $f_4 = abb$
- $f_5 = bb$
- $f_6 = aba$
- $f_7 = \$$

# Burrows-Wheeler Transform [BW94] (1/2)

## Definition: Burrows-Wheeler Transform

Given a text  $T$  of length  $n$  and its suffix array  $SA$ , for  $i \in [1, n]$  the **Burrows-Wheeler transform** is

$$BWT[i] = \begin{cases} T[SA[i] - 1] & SA[i] > 1 \\ \$ & SA[i] = 1 \end{cases}$$


	1	2	3	4	5	6	7	8	9	10	11	12	13
$T$	a	b	a	b	c	a	b	c	a	b	b	a	\$
$SA$	13	12	1	9	6	3	11	2	10	7	4	8	5
$LCP$	0	0	1	2	2	5	0	2	1	1	4	0	3
$BWT$	a	b	\$	c	c	b	b	a	a	a	a	b	b

- character before the suffix in  $SA$ -order
- choose characters cyclic ⓘ \$ for first suffix

- can compute  $BWT$  in  $O(n)$  time
- for binary alphabet  $O(n/\sqrt{\lg n})$  time and  $O(n/\lg n)$  words space is possible [KK19]



- definition is not very descriptive
- easy way to compute  $BWT$
- what can we do with the  $BWT$

-  **PINGO** can the  $BWT$  be reversed?

# Burrows-Wheeler Transform (2/2)

## Definition: Cyclic Rotation

Given a text  $T$  of length  $n$ , the  $i$ -th **cyclic rotation** is

$$T^{(i)} = T[j..n]T[1..i]$$

- $i$ -th cyclic rotation is concatenation of  $i$ -th suffix and  $(i - 1)$ -th prefix

## Definition: Burrows-Wheeler Transform (alt.)

Given a text  $T$  and a matrix containing all its cyclic rotations in lexicographical order as **columns**, then the **Burrows-Wheeler transform** of the text is the last **row** of the matrix

$T = \text{ababcabcabba\$}$

$T^{(1)} T^{(2)} T^{(3)} T^{(4)} T^{(5)} T^{(6)} T^{(7)} T^{(8)} T^{(9)} T^{(10)} T^{(11)} T^{(12)} T^{(13)}$

a	b	a	b	c	a	b	c	a	b	b	a	\$
b	a	b	c	a	b	c	a	b	b	a	\$	a
a	b	c	a	b	c	a	b	b	a	\$	a	b
b	c	a	b	c	a	b	b	a	\$	a	b	a
c	a	b	c	a	b	b	a	\$	a	b	a	b
a	b	c	a	b	b	a	\$	a	b	a	b	c
b	c	a	b	b	a	\$	a	b	a	b	c	a
c	a	b	b	a	\$	a	b	a	b	c	a	b
a	b	b	a	\$	a	b	a	b	c	a	b	c
b	b	a	\$	a	b	a	b	c	a	b	c	a
b	a	\$	a	b	a	b	c	a	b	c	a	b
a	\$	a	b	a	b	c	a	b	c	a	b	b
\$	a	b	a	b	c	a	b	c	a	b	b	a

# First and Last Row

- two important rows in the matrix
- other rows are not needed at all
- there is a special relation between the two rows
  - ⓘ later this lecture

## First Row F

- contains all characters of the text in sorted order

## Last Row L

- is the *BWT* itself

$T = ababcabcabba\$$

	$T^{(13)}$	$T^{(12)}$	$T^{(1)}$	$T^{(9)}$	$T^{(6)}$	$T^{(3)}$	$T^{(11)}$	$T^{(2)}$	$T^{(10)}$	$T^{(7)}$	$T^{(4)}$	$T^{(8)}$	$T^{(5)}$
F	\$	a	a	a	a	a	b	b	b	b	b	c	c
	a	\$	b	b	b	b	a	a	b	c	c	a	a
	b	a	a	b	c	c	\$	b	a	a	a	b	b
	a	b	b	a	a	a	a	c	\$	b	b	b	c
	b	a	c	\$	b	b	b	a	a	b	c	a	a
	c	b	a	a	b	c	a	b	b	a	a	\$	b
	a	c	b	b	a	a	b	c	a	\$	b	a	b
	b	a	c	a	\$	b	c	a	b	a	b	b	a
	c	b	a	b	a	b	a	b	c	b	a	a	\$
	a	c	b	c	b	a	b	b	a	a	\$	b	a
	b	a	b	a	a	\$	c	a	b	b	a	c	b
	b	b	a	b	b	a	a	\$	c	c	b	a	a
L	a	b	\$	c	c	b	b	a	a	a	a	b	b

# Properties of the BWT: Rank of Characters

## Definition: Rank

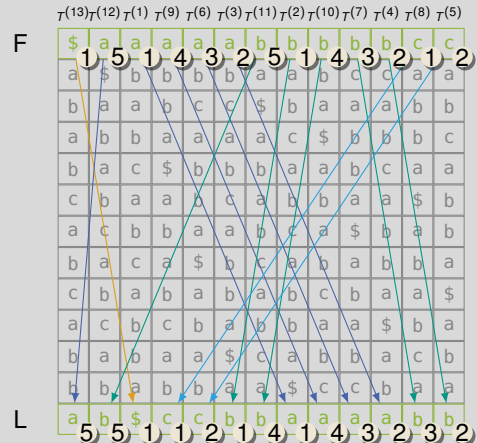
Given a text  $T$  over an alphabet  $\Sigma$ , the **rank** of a character at position  $i \in [1, n]$  is

$$\text{rank}(i) = |\{j \in [1, i]: T[j] = T[i]\}|$$

- rank is number of same characters that occur before in the text
- mark ranks of characters w.r.t. text not *BWT*
- order of ranks is the same in first and last row

$T$	a	b	a	b	c	a	b	c	a	b	b	a	\$
rank	1	1	2	2	1	3	3	2	4	4	5	5	1

$T = \text{ababcabcabba\$}$



# LF-Mapping (1/2)

- want to map characters from last to first row
- why do we want this?
  - helps with pattern matching
  - transform *BWT* back to *T*

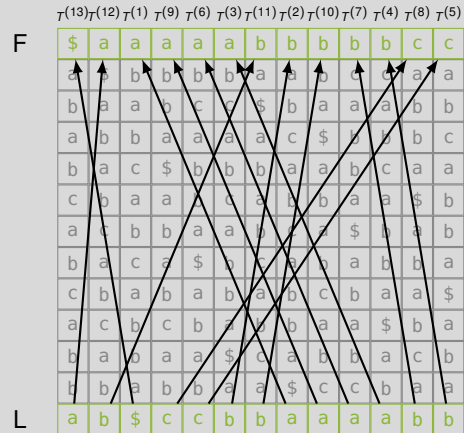
## Definition: *LF*-mapping

Given a text *T* of length *n* and its suffix array *SA*, then the *LF*-mapping is a permutation of  $[1, n]$ , such that

$$LF(i) = j \iff SA[j] = SA[i] - 1$$

- similar to definition of *BWT*
- requires *SA* or explicitly saving *LF*-mapping

*T* = ababcabcabba\$





## LF-Mapping (2/2)

### Definition: *C*-Array and *Rank*-Function

Given a text  $T$  of length  $n$  over an alphabet  $\Sigma$ ,  $\alpha \in \Sigma$ , and  $i \in [1, n]$  then


$$C[\alpha] = |\{i \in [1, n] : T[i] < \alpha\}|$$

and

$$\text{rank}_\alpha(i) = |\{j \in [1, i] : T[j] = \alpha\}|$$

- $C$  contains total number of smaller characters
- $\text{rank}_\alpha$  contains number of  $\alpha$ 's in prefix  $T[1..i]$
- $\text{rank}_\alpha$  can be computed in  $O(1)$  time [FM00]

$T$	a	b	a	b	c	a	b	c	a	b	b	a	\$
$\text{rank}$	1	1	2	2	1	3	3	2	4	4	5	5	1

- rank now on *BWT*
- $C$  is exclusive prefix sum over histogram 

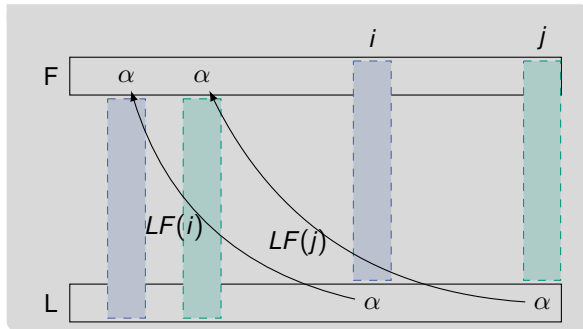
### Definition: *LF*-Mapping (alt.)

Given a *BWT*, its *C*-array, and its *rank*-Function, then

$$LF(i) = C[BWT[i]] + \text{rank}_{BWT[i]}(i)$$

# Reversing the BWT (1/2)

- characters (w.r.t. text) preserve order in  $L$  and  $F$
- $LF$ -mapping returns previous character in text



$T = \text{ababcabcabba}\$$

	\$	a	a	a	a	a	b	b	b	b	b	c	c
F	a	b	b	b	b	a	a	b	b	c	a	a	a
	b	a	a	b	c	\$	b	a	a	a	b	b	b
	a	b	b	a	a	a	c	\$	b	b	b	c	c
	b	a	c	\$	b	b	a	a	b	c	a	a	a
	c	b	a	a	c	a	b	b	a	a	\$	b	b
	a	c	b	b	a	a	b	a	\$	b	a	b	b
	b	a	c	a	\$	b	c	b	a	b	b	a	a
	c	b	a	b	a	b	a	b	c	b	a	a	\$
	a	c	b	c	b	b	b	a	a	\$	b	a	b
	b	a	b	a	a	\$	c	a	b	b	a	c	b
	b	b	a	b	b	a	a	\$	c	c	b	a	a
L	a	b	\$	c	c	b	b	a	a	a	a	b	b
LF	2	7	1	12	13	8	9	3	4	5	6	10	11

## Reversing the BWT (2/2)

- characters (w.r.t. text) preserve order in  $L$  and  $F$
- $LF$ -mapping returns previous character in text

- $T[n] = \$$  and  $T^{(n)}$  in first row
- apply  $LF$ -mapping on result to obtain any character

$$T[n - i] = L[\underbrace{LF(LF(\dots(LF(1))\dots))}_{i-1 \text{ times}}]$$

	1	2	3	4	5	6	7	8	9	0	11	12	13
L	a	b	\$	c	c	b	b	a	a	a	a	b	b
LF	2	7	1	12	13	8	9	3	4	5	6	10	11

- $T[13] = \$$  and  $k = 1$
- $T[12] = L[1] = a$  and  $k = LF(1) = 2$
- $T[11] = L[2] = b$  and  $k = LF(2) = 7$
- $T[10] = L[7] = b$  and  $k = LF(7) = 9$
- $T[9] = L[9] = a$  and  $k = LF(9) = 4$
- $T[9] = L[4] = c$  and  $k = LF(4) = 12$
- ...

# Properties of the BWT: Runs

- *BWT* is reversible
- can be used for lossless compression

## Definition: Run (simplified)


Given a text  $T$  of length  $n$ , we call its substring  $T[i..j]$  a **run**, if

- $T[k] = T[\ell]$  for all  $k, \ell \in [i, j]$  and
- $T[j - 1] \neq T[j]$  and  $T[j + 1] \neq T[j]$

❗ (To be more precise, this is a definition for a run of a periodic substring with smallest period 1, but this is not important for this lecture )



	1	2	3	4	5	6	7	8	9	0	11	12	13
L	a	b	\$	c	c	b	b	a	a	a	a	b	b

- *BWT* contains lots of runs
- same context is often grouped together 

# Compressing the BWT: Run-Length Compression

## Definition: Run-Length Encoding

Given a text  $T$ , represent each run  $T[i..i + \ell)$  as tuple

$$(T[i], \ell)$$

$T = \text{ababcabcabba\$}$

	1	2	3	4	5	6	7	8	9	10	11	12	13
BWT	a	b	\$	c	c	b	b	a	a	a	a	b	b



- (a, 1)
- (b, 1)
- (\$, 1)
- (c, 2)
- (b, 2)
- (a, 4)
- (b, 2)

# Compressing the BWT: Move-to-Front

## Definition: Move-To-Front Encoding

Given a text  $T$  over an alphabet  $\Sigma = [1, \sigma]$ , the **MTF encoding**  $MTF(T)$  of the text is computed as follows

- start with a list  $X = \Sigma[1], \Sigma[2], \dots, \Sigma[\sigma]$
- scan text from left to right, for character  $T[i]$ 
  - append position of  $T[i]$  in  $X$  to  $MTF(T)$  and
  - move  $T[i]$  to front of  $X$

- MTF encoding can easily be reverted 
- consists of many small numbers
- runs are preserved
- use Huffman on encoding  no theoretical improvement but good in practice

$T = ababcabcabba\$$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	\$	c	c	b	b	a	a	a	a	b	b

- $X = \$, a, b, c$
- $MTF = 2$  and  $X = a, \$, b, c$
- $MTF = 23$  and  $X = b, a, \$, c$
- $MTF = 233$  and  $X = \$, b, a, c$
- $MTF = 2334$  and  $X = c, \$, b, a$
- $MTF = 23341$  and  $X = c, \$, b, a$
- $MTF = 233411$  and  $X = c, \$, b, a$
- ...
- $MTF = 23341131411121$

# Pattern Matching using the BWT

## Recap


Given a text  $T$  of length  $n$  over an alphabet  $\Sigma$ ,  $\alpha \in \Sigma$ , and  $i \in [1, n]$  then


$$C[\alpha] = |\{i \in [1, n] : T[i] < \alpha\}|$$

and

$$\text{rank}_\alpha(i) = |\{j \in [1, i] : T[j] = \alpha\}|$$

- interval for  $\alpha$  is  $[C[\alpha - 1], C[\alpha + 1]]$
- find sub-interval using  $\text{rank}_\alpha$

- example on the board 


- find interval of occurrences in *SA* using *BWT*
- *SA* is divided into intervals based on first character of suffix  as seen during SAIS
- text from *BWT* is backwards
- search pattern backwards

# Backwards Search in the BWT

**Function** *BackwardsSearch*( $P[1..n]$ ,  $C$ ,  $rank$ ):


```

1  |  $s = 1, e = n$ 
2  | for  $i = m, \dots, 1$  do
3  |   |  $s = C[P[i]] + rank_{P[i]}(s - 1) + 1$ 
4  |   |  $e = C[P[i]] + rank_{P[i]}(e)$ 
5  |   | if  $s > e$  then
6  |   |   | return  $\emptyset$ 
7  | return  $[s, e]$ 
  
```

- no access to text or SA required
- no binary search
- existential queries are easy
- counting queries are easy
- reporting queries require additional information
- example on the board 



# Sampling the Suffix Array

- reporting queries require  $SA$
- storing whole  $SA$  requires too much space
- better: sample every  $s$ -th  $SA$  position in  $SA'$  

- how to find sampled position?
- mark sampled positions in bit vector of size  $n$
- if match occurs check if position is sampled
- otherwise find sample using  $LF$
- if  $SA[i] = j$  then  $SA[LF(i)] = j - 1$

- $rank_1(i)$  in bit vector is number of sample
- $SA'[rank_1(i)]$  is sampled value
- $SA'[rank_1(i)] + \#steps$  till sample found is correct  $SA$  value

- finding a sample requires  $O(s \cdot t_{rank})$  time

# Efficient Bit Vectors in Practice (1/3)

`std::vector<char/int/...>`

- easy access
- very big: 1, 4, ... bytes per bit

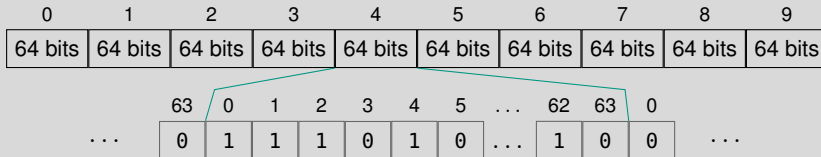
`std::vector<bool>`

- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

`std::vector<uint64_t>`

- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

- $i/64$  is position in 64-bit word
- $i\%64$  is position in word



# Efficient Bit Vectors in Practice (2/3)

```

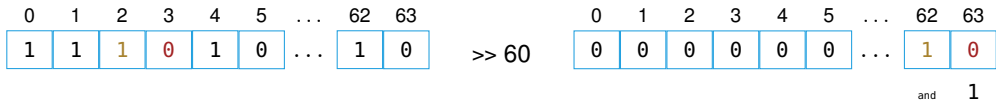
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> ( 63 - (i % 64)) ) & 1ULL;
  
```

shift bits right

# bits

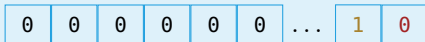
logical and 1



# Efficient Bit Vectors in Practice (3/3)

`(block >> (63-(i%64))) & 1ULL;`

- fill bit vector from left to right



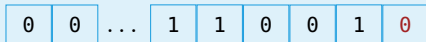
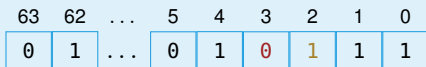
- assembler code:
 

```

mov ecx, edi
not ecx
shr rsi, cl
mov eax, esi
and eax, 1
      
```

`(block >> (i%64)) & 1ULL;`

- fill bit vector right to left



- assembler code:
 

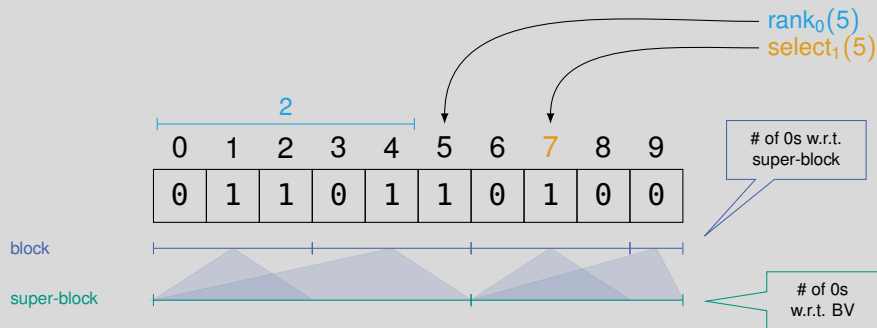
```

mov ecx, edi
shr rsi, cl
mov eax, esi
and eax, 1
      
```

# Rank Queries in Bit Vectors (1/2)

$\text{rank}_\alpha(i)$  # of  $\alpha$ s before  $i$


$\text{select}_\alpha(j)$  position of  $j$ -th  $\alpha$



## Rank Queries in Bit Vectors (2/2)


- for a bit vector of size  $n$
- blocks of size  $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size  $s' = s^2 = \Theta(\lg^2 n)$

- for all  $\lfloor \frac{n}{s'} \rfloor$  super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$  bits of space

-  **PINGO** how fast can rank queries be answered?

- for all  $\lfloor \frac{n}{s} \rfloor$  blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$  bits of space

- for all length- $s$  bit vectors, for every position  $i$  store number of 0s up to  $i$
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$  bits of space

- query in  $O(1)$  time 
- $rank_0(i) = i - rank_1(i)$

# The FM-Index (First Look) [FM00]

## Building Blocks of FM-Index

- wavelet tree on BWT providing *rank*-function
  - ⓘ wavelet trees are topic of next lecture!
- *C*-array
- sampled suffix array with sample rate  $s$
- bit vector marking sampled suffix array positions

## Lemma: FM-Index Space Requirements

Given a text  $T$  of length  $n$  over an alphabet of size  $\sigma$ , the FM-index requires  $O(n \lg \sigma)$  bits of space

## Space Requirements

- wavelet tree:  $n \lceil \lg \sigma \rceil (1 + o(1))$  bits
- *C*-array:  $\sigma \lceil \lg n \rceil$  bits ⓘ  $n(1 + o(1))$  bits if  $\sigma \geq \frac{n}{\lg n}$
- sampled suffix array:  $\frac{n}{s} \lceil \lg n \rceil$  bits
- bit vector:  $n(1 + o(1))$  bits

- space and time bounds can be achieved with  $s = \lg_{\sigma} n$

# Conclusion and Outlook

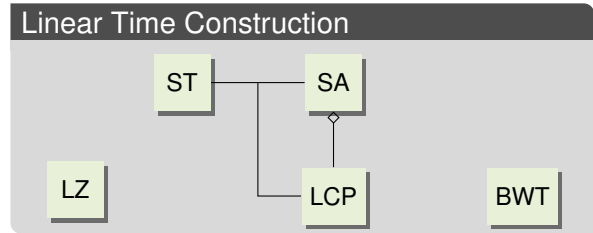
## This Lecture

- Burrows-Wheeler transform
- introduction to FM-index
- efficient bit vectors
- rank queries on bit vectors

## Next Lecture

- wavelet trees
- more on FM-index

## Linear Time Construction





# Bibliography I

- [BW94] Michael Burrows and David J. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Technical report. 1994.
- [FM00] Paolo Ferragina and Giovanni Manzini. “Opportunistic Data Structures with Applications”. In: *FOCS*. IEEE Computer Society, 2000, pages 390–398. DOI: [10.1109/SFCS.2000.892127](https://doi.org/10.1109/SFCS.2000.892127).
- [KK19] Dominik Kempa and Tomasz Kociumaka. “String Synchronizing Sets: Sublinear-Time BWT Construction and Optimal LCE Data Structure”. In: *STOC*. ACM, 2019, pages 756–767.
- [ZL77] Jacob Ziv and Abraham Lempel. “A Universal Algorithm for Sequential Data Compression”. In: *IEEE Trans. Inf. Theory* 23.3 (1977), pages 337–343. DOI: [10.1109/TIT.1977.1055714](https://doi.org/10.1109/TIT.1977.1055714).
- [ZL78] Jacob Ziv and Abraham Lempel. “Compression of Individual Sequences via Variable-Rate Coding”. In: *IEEE Trans. Inf. Theory* 24.5 (1978), pages 530–536. DOI: [10.1109/TIT.1978.1055934](https://doi.org/10.1109/TIT.1978.1055934).