10.1 Burrows-Wheeler transform

This exposition has been developed by David Weese. It is based on the following sources, which are all recommended reading:

1. M. Burrows and D. J. Wheeler (1994) *A Block-sorting Lossless Data Compression Algorithm*, SRC Research Report 124
2. G. Navarro and V. Mäkinen (2007) *Compressed Full-Text Indexes*, ACM Computing Surveys 39(1), Article no. 2 (61 pages), Section 5.3
3. D. Adjeroh, T. Bell, and A. Mukherjee (2008) *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*, Springer

10.2 Motivation

We have seen the suffix array to be an efficient data structure for exact string matching in a fixed text. However, for large texts like the human genome of about 3 billion basepairs, the text and the suffix array alone would consume **15 Gb** of memory. To solve the exact string matching in optimal $\mathcal{O}(m + p)$ time (m=pattern length, p=number of occurrences) we would need an enhanced suffix array of $3 \cdot (1 + 4 + 4 + 4)$ Gb = **39 Gb** of memory.

Both cases exceed the amount of physical memory of a typical desktop computer, therefore we need a different data structure with a smaller memory footprint. Burrows and Wheeler proposed in 1994 a lossless compression algorithm (now used in bzip2). The algorithm transforms a given text into the so called *Burrows-Wheeler transform* (BWT), a permutation of the text that can be back transformed. The transformed text can in general be better compressed than the original text as in the BWT equal characters tend to form consecutive runs which can be compressed using run-length encoders.

We will see that it is possible to conduct exact searches using only the compressed BWT and some auxiliary tables.

10.3 Definitions

We consider a string T of length n. For $i, j \in \mathbb{N}$ we define:

- $[i..j] := \{i, i+1, \ldots, j\}$
- $[i..j) := [i..j-1]$
- $T[i]$ is the i-th character of T.
- $T[i..j] := T[i]T[i+1]\ldots T[j]$ is the substring from the i-th to the j-th character
- We start counting from **1**, i. e. $T = T[1..n]$
- $|T|$ denotes the string length, i. e. $|T| = n$
- The concatenation of strings X, Y is denoted as $X \cdot Y$, e. g. $T = T[1..i] \cdot T[i+1..n]$ for $i \in [1..n)$

Definition 1 (cyclic shift)**.** Let $T = T[1..n]$ be a text over the alphabet Σ that ends with unique character $T[n] = \$$, which is the lexicographically smallest character in Σ. The i-th *cyclic shift* of T is $T[i..n] \cdot T[1..i-1]$ for $i \in [1..n]$ and denoted as $T^{(i)}$.

Example 2.

$$
\begin{aligned}
T &= \texttt{mississippi\$} \\
T^{(1)} &= \texttt{mississippi\$} \\
&\vdots \\
T^{(3)} &= \texttt{ssissippi\$mi} \\
&\vdots \\
T^{(n)} &= \texttt{\$mississippi}
\end{aligned}
$$

10.4 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) can be obtained by the following steps:

1. Form a conceptual matrix $\mathcal{M}$ whose rows are the n cyclic shifts of the text T
2. Lexicographically sort the rows of $\mathcal{M}$.
3. Construct the transformed text T^{bwt} by taking the last column of $\mathcal{M}$.

The transformed text T^{bwt} in the last column is also denoted as **L** (last). Notice that every row and every column of $\mathcal{M}$, hence also the transformed text L is a permutation of T. In particular the first column of $\mathcal{M}$, call it **F** (first), is obtained by lexicographically sorting the characters of T (or, equally, the characters of L).

Example 3. Form $\mathcal{M}$ and sort rows lexicographically:

		F	L
`mississippi$`		`$`	`mississipp` *i*
`ississippi$m`		`i`	`$mississip` *p*
`ssissippi$mi`		`i`	`ppi$missis` *s*
`sissippi$mis`		`i`	`ssippi$mis` *s*
`issippi$miss`		`i`	`ssissippi$` *m*
`ssippi$missi`	sort	`m`	`ississippi` *$*
`sippi$missis`	⇒	`p`	`i$mississi` *p*
`ippi$mississ`		`p`	`pi$mississ` *i*
`ppi$mississi`		`s`	`ippi$missi` *s*
`pi$mississip`		`s`	`issippi$mi` *s*
`i$mississipp`		`s`	`sippi$miss` *i*
`$mississippi`		`s`	`sissippi$m` *i*

The transformed string L usually contains long runs of identical symbols and therefore can be efficiently compressed using move-to-front coding, in combination with statistical coders.

10.5 Constructing the BWT

Note that when we sort the rows of $\mathcal{M}$ we are essentially sorting the suffixes of T. Hence, there is a strong relation between the matrix $\mathcal{M}$ and the suffix array A of T.

i	$T^{(i)}$		i	$T^{(i)}$		A	$T[A[j]..n]$
1	`mississippi$`		12	`$mississippi`		12	`$`
2	`ississippi$m`		11	`i$mississipp`		11	`i$`
3	`ssissippi$mi`		8	`ippi$mississ`		8	`ippi$`
4	`sissippi$mis`		5	`issippi$miss`		5	`issippi$`
5	`issippi$miss`		2	`ississippi$m`		2	`ississippi$`
6	`ssippi$missi`	sort	1	`mississippi$`	$\widehat{=}$	1	`mississippi$`
7	`sippi$missis`	⇒	10	`pi$mississip`		10	`pi$`
8	`ippi$mississ`		9	`ppi$mississi`		9	`ppi$`
9	`ppi$mississi`		7	`sippi$missis`		7	`sippi$`
10	`pi$mississip`		4	`sissippi$mis`		4	`sissippi$`
11	`i$mississipp`		6	`ssippi$missi`		6	`ssippi$`
12	`$mississippi`		3	`ssissippi$mi`		3	`ssissippi$`

Lemma 4. *The Burrows-Wheeler transform T^{bwt} can be constructed from the suffix array A of T. It holds:*

$$T^{bwt}[i] = \begin{cases} T[A[i]-1] & \text{if } A[i] > 1 \\ \$ & \text{else} \end{cases}$$

Proof: Since T is terminated with the special character \$, which is lexicographically smaller than any other character and occurs only at the end of T, a comparison of two shifts ends at latest after comparing a \$. Hence the characters right of the \$ do not influence the order of the cyclic shifts and they are sorted exactly like the suffices of T. For each suffix starting at position $A[i]$ the last column contains the preceding character at position $A[i]-1$ (or n resp.).

Corollary 5. *The Burrows-Wheeler transform of a text of length n can be constructed in $O(n)$ time.*

Corollary 6. *The i-th row of $\mathcal{M}$ contains the $A[i]$-th cyclic shift of T, i. e. $\mathcal{M}_i = T^{(A[i])}$.*

10.6 Reverse transform

One interesting property of the Burrows-Wheeler transform T^{bwt} is that the original text T can be reconstructed by a reverse transform without any extra information. Therefore we need the following definition:

Definition 7 (L-to-F mapping)**.** Let $\mathcal{M}$ be the sorted matrix of cyclic shifts of the text T. LF is a function $\mathsf{LF} : [1..n] \to [1..n]$ that maps the rank of a cyclic shift X to the rank of $X^{(n)}$ which is X shifted by one to the right:

$$\mathsf{LF}(l) = f \Leftrightarrow \mathcal{M}_f = \mathcal{M}_l^{(n)}$$

LF represents a one-to-one correspondence between elements of F and elements of L, and $L[i] = F[\mathsf{LF}[i]]$ for all $i \in [1..n]$. Corresponding characters stem from the same position in the text. That can be concluded from the following equivalence:

$$\begin{aligned}
\mathsf{LF}(l) = f \quad &\Leftrightarrow \quad \mathcal{M}_f = \mathcal{M}_l{}^{(n)} \\
&\Leftrightarrow \quad T^{(A[f])} = T^{(A[l])^{(n)}} \\
&\Leftrightarrow \quad T^{(A[f])} = T^{(A[l]+n-1)} \\
&\Leftrightarrow \quad A[f] \equiv A[l] + (n-1) \qquad \text{(mod n)}
\end{aligned}$$

Example 8. T = `mississippi$`. It holds $\mathsf{LF}(1) = 2$ as the cyclic shift in row 1 of $\mathcal{M}$ shifted by one to the right occurs in row 2. $\mathsf{LF}(2) = 7$ as the cyclic shift in row 2 of $\mathcal{M}$ shifted by one to the right occurs in row 7. For the same reason holds $\mathsf{LF}(7) = 8$.

	F		L
$T^{(12)}$	$	mississipp	i
$T^{(11)}$	i	$mississip	p
	i	ppi$missis	s
	i	ssippi$mis	s
	i	ssissippi$	m
	m	ississippi	$
$T^{(10)}$	p	i$mississi	p
$T^{(9)}$	p	pi$mississ	i
	s	ippi$missi	s
	s	issippi$mi	s
	s	sippi$miss	i
	s	sissippi$m	i

i	$\mathsf{LF}(i)$
1	**2**
2	**7**
3	9
4	10
5	6
6	1
7	**8**
8	3
9	11
10	12
11	4
12	5

Thus the first character in row f stems from position $A[f]$ in the text. That is the same position the last character in row l stems from. One important observation is that the relative order of two cyclic shifts that end with the same character is preserved after shifting them one to the right.

Observation 9 (rank preservation)**.** Let $i, j \in [1..n]$ with $L[i] = L[j]$. If $i < j$ then $\mathsf{LF}[i] < \mathsf{LF}[j]$ follows.

Proof: From $L[i] = L[j]$ and $i < j$ follows $\mathcal{M}_i[1..n-1] <_{\text{lex}} \mathcal{M}_j[1..n-1]$. Thus holds:

$$\begin{aligned}
& L[i] \cdot \mathcal{M}_i[1..n-1] && <_{\text{lex}} && L[j] \cdot \mathcal{M}_j[1..n-1] \\
\Leftrightarrow \quad & \mathcal{M}_i^{(n)} && <_{\text{lex}} && \mathcal{M}_j^{(n)} \\
\Leftrightarrow \quad & \mathcal{M}_{\mathsf{LF}[i]} && <_{\text{lex}} && \mathcal{M}_{\mathsf{LF}[j]} \\
\Leftrightarrow \quad & \mathsf{LF}[i] && < && \mathsf{LF}[j]
\end{aligned}$$

Observation 9 allows to compute the LF-mapping without using the suffix array as the i-th occurrence of a character α in L is mapped to the i-th occurrence of α in F.

Example 10. T = `mississippi$`. The L-to-F mapping preserves the relative order of indices of matrix rows that end with the same character. The increasing sequence of *all* indices $3 < 4 < 9 < 10$ of rows that end with `s` is mapped to the increasing and *contiguous* sequence $9 < 10 < 11 < 12$.

F		L
$	mississipp	i
i	$mississip	p
i	ppi$missis	s
i	ssippi$mis	s
i	ssissippi$	m
m	ississippi	$
p	i$mississi	p
p	pi$mississ	i
s	ippi$missi	s
s	issippi$mi	s
s	sippi$miss	i
s	sissippi$m	i

i	LF(i)
1	2
2	7
3	**9**
4	**10**
5	6
6	1
7	8
8	3
9	**11**
10	**12**
11	4
12	5

Definition 11. Let T be a text of length n over an alphabet Σ and L the BWT of T.

- Let $C : \Sigma \to [0..n]$ be a function that maps a character $c \in \Sigma$ to the total number of occurrences in T of the characters which are alphabetically smaller than c.
- Let $\mathsf{Occ} : \Sigma \times [1..n] \to [1..n]$ be a function that maps $(c, k) \in \Sigma \times [1..n]$ to the number of occurrences of c in the prefix $L[1..k]$ of the transformed text L.

Theorem 12. *For the L-to-F mapping* LF *of a text T holds:*

$$LF(i) = C(L[i]) + Occ(L[i], i)$$

Proof: Let $\alpha = L[i]$. Of all occurrences of the character α in L, $\mathsf{Occ}(L[i], i)$ gives index of the occurrence at position i starting counting from 1. $C(L[i]) + j$ is the position of the j-th occurrence of α in F starting counting from 1. With $j = \mathsf{Occ}(L[i], i)$ the j-th occurrence of α in L is mapped to the j-th occurrence of α in F.

How can we back-transform L to T? With the L-to-F mapping we reconstruct T from right to left by cyclic shifting by one to the right beginning with $T^{(n)}$ and extracting the first characters. That can be done using the following properties:

- The last character of T is \$, whose only occurrence in F is $F[1] = \$$, thus $\mathcal{M}_1 = T^{(n)}$ is T shifted by one to the right.
- $\mathcal{M}_{\mathsf{LF}(1)} = T^{(n)^{(n)}} = T^{(n-1)}$ is T shifted by 2 to the right. Therefore $F[\mathsf{LF}(1)] = L[1]$ is the second last character of T.
- The character $T[n-i]$ is $F[\underbrace{\mathsf{LF}(\mathsf{LF}(\ldots\mathsf{LF}(1)\ldots))}_{i \text{ times LF}}] = L[\underbrace{\mathsf{LF}(\mathsf{LF}(\ldots\mathsf{LF}(1)\ldots))}_{i-1 \text{ times LF}}]$.

Hence, we need only L, C and Occ for the reverse transform. C can be obtained by bucket sorting L. LF only uses values $\mathsf{Occ}(L[i], i)$ which can be precomputed in an array of size n by a sequential scan over L.
The pseudo-code for the reverse transform of $L = T^{\text{bwt}}$ is given in algorithm **reverse_transform**.

Example 13. Reverse transform L = `ipssm$pissii` of length $n = 12$ over the alphabet $\Sigma = \{\$, \mathtt{i}, \mathtt{m}, \mathtt{p}, \mathtt{s}\}$. First, we count the number of occurrences n_α of every character $\alpha \in \Sigma$ in L and compute the partial sums $C(\alpha) = \sum_{\beta<\alpha} n_\beta$ of characters smaller than α to obtain C.

$\alpha \in \Sigma$	$	i	m	p	s
n_α	1	4	1	2	4
$C(\alpha)$	0	1	5	6	8

```
// reverse_transform(L,Occ,C)
i = 1, j = n;
T[n] = $;
while (j > 1) do
        j = j − 1;
        T[j] = L[i];
        i = C(L[i]) + Occ(L[i], i);
od
return T;
```

For every i we precompute $\mathsf{Occ}(L[i], i)$ by sequentially scanning L and counting the number of occurrences of $L[i]$ in L up to position i. That can be done during the first run, where we determine the values n_α.

i	1	2	3	4	5	6	7	8	9	10	11	12
$L[i]$	i	p	s	s	m	$	p	i	s	s	i	i
$\mathsf{Occ}(L[i], i)$	1	1	1	2	1	1	2	2	3	4	3	4

First, set the last text character $T[12] =$ *$*

Start in row $i = 1$:
Extract the second-last character $T[11] = L[1] =$ *i*
$T =$ . . . . . i$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[1]) + \mathsf{Occ}(L[1], 1) = 1 + 1 = 2$
Extract the character $T[10] = L[2] =$ *p*
$T =$ . . . . . pi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[2]) + \mathsf{Occ}(L[2], 2) = 6 + 1 = 7$
Extract the character $T[9] = L[7] =$ *p*
$T =$ . . . . . ppi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[7]) + \mathsf{Occ}(L[7], 7) = 6 + 2 = 8$
Extract the character $T[8] = L[8] =$ *i*
$T =$ . . . . . ippi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[8]) + \mathsf{Occ}(L[8], 8) = 1 + 2 = 3$
Extract the character $T[7] = L[3] =$ *s*
$T =$ . . . . . sippi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[3]) + \mathsf{Occ}(L[3], 3) = 8 + 1 = 9$
Extract the character $T[6] = L[9] =$ *s*
$T =$ . . . . . ssippi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[9]) + \mathsf{Occ}(L[9], 9) = 8 + 3 = 11$
Extract the character $T[5] = L[11] =$ *i*
$T =$ . . . . issippi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[11]) + \mathsf{Occ}(L[11], 11) = 1 + 3 = 4$
Extract the character $T[4] = L[4] =$ *s*
$T =$ . . . sissippi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[4]) + \mathsf{Occ}(L[4], 4) = 8 + 2 = 10$
Extract the character $T[3] = L[10] =$ *s*
$T =$ . . ssissippi$

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[10]) + \mathsf{Occ}(L[10], 10) = 8 + 4 = 12$

Extract the character $T[2] = L[12] =$ `i`
$T =$ `.ississippi$`

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[12]) + \mathsf{Occ}(L[12], 12) = 1 + 4 = 5$
Extract the character $T[1] = L[5] =$ `m`
$T =$ `mississippi$`

10.7 Backward search

This exposition has been developed by David Weese. It is based on the following sources, which are all recommended reading:

1. P. Ferragina, G. Manzini (2000) *Opportunistic data structures with applications*, Proceedings of the 41st IEEE Symposium on Foundations of Computer Science
2. P. Ferragina, G. Manzini (2001) *An experimental study of an opportunistic index*, Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, pp. 269-278

For a pattern $P = P[1..m]$ we want to count the number of occurrences in a text $T = T[1..m]$ given its Burrows-Wheeler transform $L = T^{\text{bwt}}$. If we would have access to the conceptual matrix $\mathcal{M}$ we could conduct a binary search like in a suffix array. However, as we have direct access to only F and L we need a different approach.

Ferragina and Manzini proposed a backward search algorithm that searches the pattern from right to left by matching growing suffixes of P. It maintains an interval of matches and transforms the interval of matches of a suffix of P to an interval of the suffix which is one character longer. At the end, the length of the interval of the whole pattern P equals the number of occurrences of P.

Occurrences can be represented as intervals due to the following observation:

Observation 14. For every suffix of $P[j..m]$ of P the matrix rows $\mathcal{M}_i$ with prefix $P[j..m]$ form a contiguous block. Thus there are $a, b \in [1..m]$ such that $i \in [a..b] \Leftrightarrow \mathcal{M}_i[1..m-j+1] = P[j..m]$.

Proof: That is direct consequence of the lexicographical sorting of cyclic shifts in $\mathcal{M}$. Note that $[a..b] = \emptyset$ for $a > b$.

Consider $[a_j..b_j]$ to be the interval of matrix rows beginning with $P_j = P[j..m]$. In that interval we search cyclic shifts whose matching prefix is preceded by $c = P[j-1]$ in the cyclic text, i. e. matrix rows that end with c. If we shift these rows by 1 to the right they begin with $P[j-1..m]$ and determine the next interval $[a_{j-1}..b_{j-1}]$.

Matrix rows that end with c are occurrences of c in $L[a_j..b_j]$. The L-to-F mapping of the first and last occurrence yields a_{j-1} and b_{j-1} (rank preservation, lemma 9).

The L-to-F mapping maps the i-th c in L to the i-th c in F. How to determine i for the first and last c in $L[a_j..b_j]$ without scanning ?

In L are $\mathsf{Occ}(c, a_j - 1)$ occurrences of c before the first occurrence in $L[a_j..b_j]$, hence:

$$i_f = \mathsf{Occ}(c, a_j - 1) + 1$$

i_l the number of the last occurrence of c in $L[a_j..b_j]$ equals the number of occurrences of c in $L[1..b_j]$:

$$i_l = \mathsf{Occ}(c, b_j)$$

Now, we can determine a_{j-1} and b_{j-1} by:

$$\begin{aligned} a_{j-1} &= C(c) + \mathsf{Occ}(c, a_j - 1) + 1 \\ b_{j-1} &= C(c) + \mathsf{Occ}(c, b_j) \end{aligned}$$

Algorithm **count** computes the number of occurrences of $P[1..m]$ in $T[1..n]$:

Example 15. $T =$ `mississippi$`. Search $P =$ `ssi`.

```
// count(P[1..m])
i = m, a = 1, b = n;
while ((a ≤ b) ∧ (i ≥ 1)) do
        c = P[i];
        a = C(c) + Occ(c, a − 1) + 1;
        b = C(c) + Occ(c, b);
        i = i − 1;
od
if (b < a) then return "not found";
           else return "found (b − a + 1) occurrences";
fi
```

i	1	2	3	4	5	6	7	8	9	10	11	12
$L[i]$	i	p	s	s	m	$	p	i	s	s	i	i

Occ(x, i)	1	2	3	4	5	6	7	8	9	10	11	12	$C(x)$
$	0	0	0	0	0	1	1	1	1	1	1	1	0
i	1	1	1	1	1	1	1	2	2	2	3	4	1
m	0	0	0	0	1	1	1	1	1	1	1	1	5
p	0	1	1	1	1	1	2	2	2	2	2	2	6
s	0	0	1	2	2	2	2	2	3	4	4	4	8

Initialization: We begin with the empty suffix ϵ which is a prefix of every suffix, hence we initialize $a_{m+1} = 1$ and $b_{m+1} = n$.

	F		L
a_4 →	$	mississipp	i
	i	$mississip	p
	i	ppi$missis	s
	i	ssippi$mis	s
	i	ssissippi$	m
	m	ississippi	$
	p	i$mississi	p
	p	pi$mississ	i
	s	ippi$missi	s
	s	issippi$mi	s
	s	sippi$miss	i
b_4 →	s	sissippi$m	i

$a_4 = 1$
$b_4 = 12$

Searching P_3: From all matrix rows we search those beginning with the last pattern character $P[3] = \texttt{i}$. From $\mathsf{Occ}(x, 0) = 0$ and $\mathsf{Occ}(x, n) = n_x$ follows $a_m = C(x) + 1$ and $b_m = C(x + 1)$.

	F	L
$a_4 \rightarrow$	$ mississipp	i
	i $mississip	p
	i ppi$missis	s
	i ssippi$mis	s
	i ssissippi$	m
	m ississippi	$
	p i$mississi	p
	p pi$mississ	i
	s ippi$missi	s
	s issippi$mi	s
	s sippi$miss	i
$b_4 \rightarrow$	s sissippi$m	i

$a_4 = 1$
$b_4 = 12$

$\Rightarrow$

F	L
$ mississipp	i
i $mississip	p
i ppi$missis	s
i ssippi$mis	s
i ssissippi$	m
m ississippi	$
p i$mississi	p
p pi$mississ	i
s ippi$missi	s
s issippi$mi	s
s sippi$miss	i
s sissippi$m	i

$a_3 = C(\mathtt{i}) + 1 = 1 + 1$
$b_3 = C(\mathtt{i}) + n_{\mathtt{i}} = 1 + 4$

Searching P_2: From all rows beginning with P_3 we search those beginning with $P[2] = \mathtt{s}$. In L we count the s's in the part before the interval (=0) and including the interval (=2) to L-to-F map the first and last s in the interval.

	F	L
	$ mississipp	i
$a_3 \rightarrow$	i $mississip	p
	i ppi$missis	s
	i ssippi$mis	s
$b_3 \rightarrow$	i ssissippi$	m
	m ississippi	$
	p i$mississi	p
	p pi$mississ	i
	s ippi$missi	s
	s issippi$mi	s
	s sippi$miss	i
	s sissippi$m	i

$a_3 = 2$
$b_3 = 5$

$\Rightarrow$

F	L
$ mississipp	i
i $mississip	p
i ppi$missis	s
i ssippi$mis	s
i ssissippi$	m
m ississippi	$
p i$mississi	p
p pi$mississ	i
s ippi$missi	s
s issippi$mi	s
s sippi$miss	i
s sissippi$m	i

$a_2 = C(\mathtt{s}) + \mathrm{Occ}(\mathtt{s}, 1) + 1 = 8 + 0 + 1$
$b_2 = C(\mathtt{s}) + \mathrm{Occ}(\mathtt{s}, 5) = 8 + 2$

Searching P_1: From all matrix rows beginning with P_2 we search those beginning with $P[1] = \mathtt{s}$.

	F	L
	$ mississipp	i
	i $mississip	p
	i ppi$missis	s
	i ssippi$mis	s
	i ssissippi$	m
	m ississippi	$
	p i$mississi	p
	p pi$mississ	i
$a_2 \rightarrow$	s ippi$missi	s
$b_2 \rightarrow$	s issippi$mi	s
	s sippi$miss	i
	s sissippi$m	i

$a_2 = 9$
$b_2 = 10$

$\Rightarrow$

F	L
$ mississipp	i
i $mississip	p
i ppi$missis	s
i ssippi$mis	s
i ssissippi$	m
m ississippi	$
p i$mississi	p
p pi$mississ	i
s ippi$missi	s
s issippi$mi	s
s sippi$miss	i
s sissippi$m	i

$a_1 = C(\mathtt{s}) + \mathrm{Occ}(\mathtt{s}, 8) + 1 = 8 + 2 + 1$
$b_1 = C(\mathtt{s}) + \mathrm{Occ}(\mathtt{s}, 10) = 8 + 4$

Found the interval for P: $[a_1..b_1]$ is the interval of matrix rows with prefix P, thus P has $b_1 - a_1 + 1 = 2$ occurrences in the text T.

```
      $mississippi
      i$mississipp
      ippi$mississ
      issippi$miss
      ississippi$m
      mississippi$
      pi$mississip
      ppi$mississi
      sippi$missis
      sissippi$mis
a1 →  ssippi$missi
b1 →  ssissippi$mi
```

$a_1 = 11$
$b_1 = 12$

10.8 Locate matches

We have seen how to count occurrences of a pattern P in the text T, but how to obtain their location in the text? Algorithm **count** determines the indexes $a, a+1, \ldots, b$ of matrix rows with prefix P. As cyclic shifts correspond to the suffixes of T, with a suffix array A of T we would be able to get the corresponding text position $pos(i)$ of the suffix in row i. It holds $pos(i) = A[i]$.

We will now see that it is not necessary to have the whole suffix array with $4n$ bytes of memory. It suffices to have a fraction of the suffix array av'ailable to compute $pos(i)$ for every $i \in [1..n]$.

The idea is as following. We logically mark a suitable subset of rows in the matrix. For the marked rows we explicitly store the start positions of the suffixes in the text. If i is marked, row $pos(i)$ is directly available. If i is not marked, the algorithm **locate** uses the L-to-F-mapping to find the row $i_1 = \mathsf{LF}(i)$ corresponding to the suffix $T[pos(i) - 1..n]$. This procedure is iterated v times until we reach a marked row i_v for which $pos(i_v)$ is available; then we set $pos(i) = pos(i_v) + v$.

This is a direct space-time trade-off.

Example 16.

Example:
$pos(1) = 12$
$pos(3) = 8$
$pos(6) = 1$
$pos(10) = 4$

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
First →	s	ippi#missi	s	9
Last →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Preprocessing. The position of every x-th letter in the text is marked and stored for the corresponding row in S.

For a row i, algorithm **locate** determines the location of the corresponding occurrence in $T[1..n]$.

locate(i) is called for every $i = [a..b]$, where $[a..b]$ is interval of occurrences computed by **count**(P). We call the conjunction of both algorithms and their required data structures the *FM Index*.

```
// locate(i)
i' = i
v = 0;
while (row i' is not marked) do
        c = L[i'];
        i' = C(c) + Occ(c, i');
        v = v + 1;
od
return pos(i') + v;
```

In the following we give an example using the BWT of the text `mississippi#`, the thinned out suffix array and the interval $[a..b]$ resulting from the search of the pattern `si`.

For each $i = [9, 10]$ we have to its position in the text. **i = 9**

Step 1

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
$i \rightarrow$	s	ippi#missi	s	9
	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

row 9 is not marked
→ L-to-F(9)=11
→ Look at row 11
v=1

Step 2

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
$i \rightarrow$	s	ippi#missi	s	9
	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

row 11 is not marked
→ L-to-F(11)=4
→ Look at row 4
v=2

Step 3

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
i → s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

row 4 is not marked
→ L-to-F(4)=10
→ Look at row 10
v=3

row 10 is marked
Calculation of the $pos(9)$:
$pos(9) = pos(10) + 3 = 4 + 3 = 7$

We saw how to avoid storing the complete suffix array when locating the text.

However, the table Occ is still quite big. It contains the number of occurrences for each character and each possible prefix of L needing $|\Sigma| \times n$ entries storing the number of occurrences.

One way to reduce the size of Occ is to store only every x-th index. The entries that are omitted can be reconstructed from stored entries at the cost of an increased run time, by simply counting in the BWT from the last stored position on.

Taken together, we can conduct an exact search in a text in time *linear* to the query size.

For example (for DNA) using the the text T (n bytes), the BWT (n bytes resp. $n/4$ bytes), the Occ table (e.g. $4 \cdot 4 \cdot n/32$ bytes when storing only every 32th entry) and a sampled suffix array (e.g. $4 \cdot n/8$ bytes when marking every 8th entry). In our example calculation we would need about $2.25n - 3n$ bytes.

11.9 Compressing the FM Index

This exposition has been developed by David Weese. It is based on the following sources, which are all recommended reading:

1. P. Ferragina, G. Manzini (2000) *Opportunistic data structures with applications*, Proceedings of the 41st IEEE Symposium on Foundations of Computer Science
2. P. Ferragina, G. Manzini (2001) *An experimental study of an opportunistic index*, Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, pp. 269-278
3. Johannes Fischer (2010), *Skriptum VL Text-Indexierung*, SoSe 2010, KIT
4. A. Andersson (1996) *Sorting and searching revisited*, Proceedings of the 5th Scandinavian Workshop on Algorithm Theory, pp. 185-197

11.10 RAM Model

From now on we assume the *RAM model* in which we model a computer with a CPU that has registers of w bits which can be modified with logical and arithmetical operations in $O(1)$ time. The CPU can directly access a memory of at most 2^w words.

In the following we assume $n \leq 2^w$ so that it is possible to address the whole input. To have a more precise measure, we count memory consumptions in bits. The uncompressed suffix array then does not require $O(n)$ memory but $O(n \log n)$ bits, as $\lceil \log_2 n \rceil$ bits are required to represent any number in $[1..n]$.

11.11 Tables of the FM Index

Let T be a text of length n over the alphabet Σ and $\sigma = |\Sigma|$ be the alphabet size. We have seen, that for the algorithms **count** and **locate** we need L and the tables C and Occ. Without compression their memory consumption is as follows:

- $L = T^{\text{bwt}}$ is a string of length n over Σ and requires $O(n \log \sigma)$ bits
- C is an array of length σ over $[0..n]$ and requires $O(\sigma \log n)$ bits
- Occ is an array of length $\sigma \times n$ over $[0..n]$ and requires $O(\sigma \cdot n \log n)$ bits
- *pos* (if every row is marked) is a suffix array of length n over $[1..n]$ and requires $O(n \log n)$ bits

We will present approaches to compress L, Occ and *pos*, but omit to compress C assuming that σ and $\log n$ are tolerably small.

11.12 Compressing L

Burrows and Wheeler proposed a move-to-front coding in combination with Huffman or arithmetic coding. In the context of the move-to-front encoding each character is encoded by its index in a list, which changes over the course of the algorithm. It works as follows:

1. Initialize a list Y of characters to contain each character in Σ exactly once
2. Scan L with $i = 1, \ldots, n$
   (a) Set $R[i]$ to the number of characters preceding character $L[i]$ in the list Y
   (b) Move character $L[i]$ to the front of Y

R is the MTF encoding of L. R can again be decoded to L in a similar way (Exercise).

Algorithm **move_to_front(L)** shows the pseudo-code of the move-to-front encoding. The array M maintains for every alphabet character the number preceding characters in Y instead of using Y directly.

```
// move_to_front(L)
for j = 1 to σ do
    M[j] = j − 1
od
for i = 1 to n do
    // ord maps a character to its rank in the alphabet
    x = ord(L[i])
    R[i] = M[x];
    for j = 1 to σ do
        if M[j] < M[x] then M[j] = M[j] + 1;  fi
    od
    M[x] = 0;
od
return R;
```

Observation 17. The BWT tends to group characters together so that the probability of finding a character close to another instance of the same character is increased substantially:

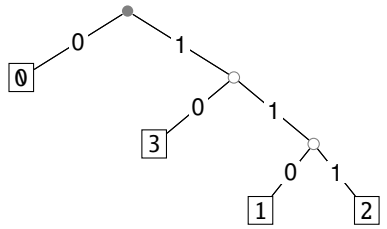
final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Observation 18. The move-to-front encoding replaces equal characters that in L are "close together" by "small values" in R. In practice, the most important effect is that zeroes tend to occur in runs in R. These can be compressed using an order-0 compressor, e.g. the Huffman encoding.

i	$L[i]$	$R[i]$	Y_{next}
			aeio
1	a	0	aeio
2	o	3	oaei
3	o	0	oaei
4	o	0	oaei
5	o	0	oaei
6	a	1	aoei
7	a	0	aoei
8	i	3	iaoe
9	i	0	iaoe
10	o	2	oiae
11	a	2	aoie
12	e	3	eaoi
13	i	3	ieao
14	e	1	eiao
15	e	0	eiao
16	i	1	ieao
17	i	0	ieao
		...	

The Huffman encoding builds a binary tree where leaves are alphabet characters. The tree is balanced such that for every node the leaves in the left and right subtree have a similar sum of occurrences.

character	0	1	2	3
occurrences in R	10	3	2	5



x	bit code of x
0	0
1	110
2	111
3	10

Left and right childs are labeled with 0 and 1. The labels on the paths to each leaf define its bit code. The more frequent a character the shorter its bit code. The final sequence H is the bitwise concatenation of bit codes of characters from left to right in R.

The final sequence of bits H is:

```
L  =  ao oooa  ai i...
R  =  03 0001  03 0...
H  =  0100001100100...
```

One property of the MTF coding is that the whole prefix $R[1..i-1]$ is required to decode character $R[i]$, the same holds for H. For encoding and decoding this is fine (practical assignment).

However, we want to search in the compressed FM index and hence need random accesses to L in algorithm **locate** which would take $O(n)$ time. Manzini and Ferragina achieve this directly on the Huffman encoded R, however their algorithm is not practical, albeit optimal in theory.

We will proceed differently by using a simple trick we can determine $L[i]$ using the Occ function. Clearly, the values $\mathsf{Occ}(c, i)$ and $\mathsf{Occ}(c, i-1)$ differ only for $c = L[i]$.

Thus we can determine both $L[i]$ and $\mathsf{Occ}(L[i], i)$ using σ Occ-queries.

Lets now discuss the possible space-time tradeoffs. The two simplest ideas are:

1. Avoid storing an Occ-table and scan L every time an Occ-query has to be answered. This occupies no space, but needs $O(n)$ time for answering a single Occ-query, leading to a total query time of $O(mn)$ for backwards search.

2. Store all answers to $\mathsf{Occ}(c, i)$ in a two-dimensional table. This table occupies $O(n\sigma \log n)$ bits of space, but allows constant-time Occ-queries and makes the storage of L obsolete. Total time for backwards search is optimal $O(m)$.

For a more practical implementation we can proceed as follows:

11.13 Compressing Occ

We reduce the problem of counting the occurrences of a character in a prefix of L to counting 1's in a prefix of a bitvector. Therefore we construct a bitvector B_c of length n for each $c \in \Sigma$ such that:

$$B_c[i] = \begin{cases} 1 & \text{if } L[i] = c \\ 0 & \text{else} \end{cases} .$$

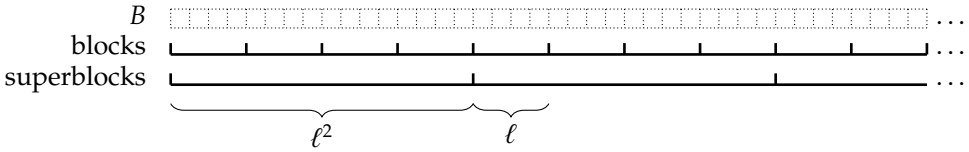
Definition 19. For a bitvector B we define $rank_1(B, i)$ to be the number of 1's in the prefix $B[1..i]$. $rank_0(B, i)$ is defined analogously.

As each 1 in the bitvector B_c indicates an occurrence of c in L, it holds:

$$\mathsf{Occ}(c, i) = rank_1(B_c, i) \quad .$$

We will see that it is possible to answer a rank query of a bitvector of length n in constant time using additional tables of $o(n)$ bits. Hence the σ bitvectors are an implementation of Occ that allows to answer Occ queries in constant time with an overall memory consumption of $O(\sigma n + o(\sigma n))$ bits. Given a bitvector $B = B[1..n]$.

We compute the length $\ell = \left\lfloor \frac{\log n}{2} \right\rfloor$ and divide B into blocks of length ℓ and superblocks of corresponding to ℓ^2 blocks.



1. For the i-th superblock we count the number of 1's from the beginning of B to the end of the superblock in $M'[i] = rank_1(B, i \cdot \ell^2)$. As there are $\left\lfloor \frac{n}{\ell^2} \right\rfloor$ superblocks, M' can be stored in $O\left(\frac{n}{\ell^2} \cdot \log n\right) = O\left(\frac{n}{\log n}\right) = o(n)$ bits.

2. For the i-th block we count the number of 1's from the beginning of the overlapping superblock to the end of the block in $M[i] = rank_1\Big(B[1 + k\ell..n], (i-k)\ell\Big)$ where $k = \left\lfloor \frac{i-1}{\ell} \right\rfloor \ell$ is the number of blocks left of the overlapping superblock. M has $\left\lfloor \frac{n}{\ell} \right\rfloor$ entries and can be stored in $O\left(\frac{n}{\ell} \cdot \log \ell^2\right) = O\left(\frac{n \log\log n}{\log n}\right) = o(n)$ bits.

3. Let P be a precomputed lookup table such that for each possible bitvector V of length ℓ and $i \in [1..\ell]$ holds $P[V][i] = rank_1(V, i)$. V has $2^\ell \times \ell$ entries of values at most ℓ and thus can be stored in

$$O\left(2^\ell \cdot \ell \cdot \log \ell\right) = O\left(2^{\frac{\log n}{2}} \cdot \log n \cdot \log\log n\right) = O\left(\sqrt{n} \log n \log\log n\right) = o(n)$$

bits.

We now decompose a rank-query into 3 subqueries using the precomputed tables. For a position i we determine the index $p = \left\lfloor \frac{i-1}{\ell} \right\rfloor$ of next block left of i and the index $q = \left\lfloor \frac{p-1}{\ell} \right\rfloor$ of the next superblock left of block p. Then it holds:

$$rank_1(B, i) = M'[q] + M[p] + P\Big[B[1 + p\ell..(p+1)\ell]\Big]\Big[i - p\ell\Big] \quad .$$

Note that $B[1 + p\ell..(p+1)\ell]$ fits into a single CPU register and can therefore be determined in $O(1)$ time. Thus a rank-query can be answered in $O(1)$ time.

11.14 Compressing Occ with Wavelet trees

Armed with constant-time rank-queries, we now develop a more space-efficient implementation of the Occ-function, sacrificing the optimal query time. The idea is to use a *wavelet tree* on the BW-transformed text.

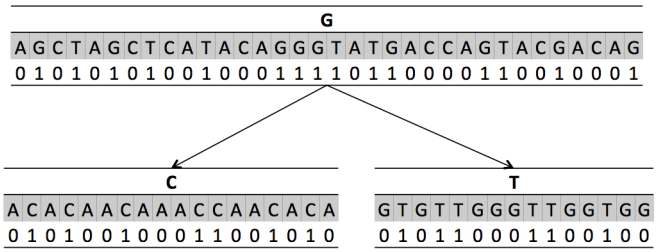
The wavelet tree of a sequence $L[1, n]$ over an alphabet Σ is a balanced binary search tree of height $O(\log \sigma)$. It is obtained as follows.

1. We create a root node v, where we divide Σ into two halves Σ_l and Σ_r of roughly equal size, where the left half contains the lexicographically smaller characters.

2. At v we store a bit-vector B_v of length n (together with data structures for $O(1)$ rank-queries), where a 0 of position i indicates that character $L[i]$ belongs to Σ_l, and a 1 indicates the it belongs to Σ_r.

3. This defines two (virtual) sequences L_v and R_v, where L_v is obtained from L by concatenating all characters $L[i]$ where $B_v[i] = 0$, in the order as they appear in L. Sequence R_v is obtained in a similar manner for positions i with $B_v[i] = 1$.

4. The left child l_v is recursively defined to be the root of the wavelet tree for L_v, and the right child r_v to be the root of the wavelet tree for R_v. This process continues until a sequence consists of only one symbol, in which case we create a leaf.



Note that the sequences themselves are not stored explicitly; node v only stores a bit-vector B_v and structures for $O(1)$ rank-queries.

Theorem 20. *The wavelet tree for a sequence of length n over an alphabet of size σ can be stored in $n \log \sigma \times (1 + o(1))$ bits.*

Proof: We concatenate all bit-vectors at the same depth d into a single bit-vector B_d of length n, and prepare it for $O(1)$-rank-queries. Hence, at any level, the space needed is $n + o(n)$ bits. Because the depth of the tree is $\lceil \log \sigma \rceil$ the claim on the space follows. In order to determine the sub-interval of a particular node v in the concatenated bit-vector B_d at level d, we can store two indices α_v and β_v such that $B_d[\alpha_v, \beta_v]$ is the bit-vector B_v associated to node v. This accounts for additional $O(\sigma \log n)$ bits. Then a rank-query is answered as follows ($b \in \{0, 1\}$):

$$rank_b(B_v, i) = rank_b(B_d, \alpha_v + i - 1) - rank_b(B_d, \alpha_v - 1),$$

where it is assumed that $i \leq \beta_v - \alpha_v + 1$, for otherwise the result is not defined.

How does the wavelet tree help for implementing the Occ-function? Suppose we want to compute $\mathsf{Occ}(c, i)$, i.e., the number of occurrences of $c \in \Sigma$ in $L[1, i]$. We start at the root r of the wavelet tree, and check if c belongs to the first or to the second half of the alphabet.

In the first case, we know that the cs are in the left child of the root, namely L_r. Hence, the number of cs in $L[1, i]$ corresponds to the number of cs in $L_r[1, rank_0(B_r, i)]$. If, on the hand, c belongs to the second half of the alphabet, we know that the cs are in the subsequence R_r that corresponds to the right child of r, and hence compute the number of occurrences of c in $R_r[1, rank_1(B_r, i)]$ as the number of cs in $L[1, i]$.

This leads to the following recursive procedure for computing $\mathsf{Occ}(c, i)$, to be invoked with $WT - occ(c, i, 1, \sigma, r)$, where r is the root of the wavelet tree. (Recall that we assume that the characters in Σ can be accessed as $\Sigma[1], \ldots, \Sigma[\sigma]$.)

```
(1) WT − occ(c, i, σ_l, σ_r, v)
(2) if σ_l = σ_r then return i; fi
(3) σ_m = ⌊(σ_l+σ_r)/2⌋;
(4) if c ≤ Σ[σ_m] then
(5)                 return WT − occ(c, rank_0(B_v, i), σ_l, σ_m, l_v);
(6)             else
(7)                 return WT − occ(c, rank_1(B_v, i), σ_m + 1, σ_r, r_v);
(8) fi
```

Due to the depth of the wavelet tree, the time for $WT - occ(\cdot)$ is $O(\log \sigma)$. This leads to the following theorem.

Theorem 21. *With backward-search and a wavelet-tree on T_bwt, we can answer counting queries in $O(m \log \sigma)$ time. The space (in bits) is*

$$O(\sigma \log n) + n \log \sigma + o(n \log \sigma),$$

where the first term account for $|C|$ + space for the α_v, the second term accounts for the wavelet tree, and the third term accounts for the rank data structure.

11.15 Compressing *pos*

To compress *pos* we mark only a subset of rows in the matrix $\mathcal{M}$ and store their text positions. Therefore we need a data structure that efficiently decides whether a row $\mathcal{M}_i = T[j]$ is marked and that retrieves j for a marked row i.

If we would mark every η-th row in the matrix ($\eta > 1$) we could easily decide whether row i is marked, e. g. iff $i \equiv 1 \pmod{\eta}$. Unfortunately this approach still has worst-cases where a single *pos*-query takes $O\left(\frac{\eta-1}{\eta}n\right)$ time (excercise).

Instead we mark the matrix row for every η-th text position, i. e. for all $j \in \left[0..\lceil\frac{n}{\eta}\rceil\right)$ row i with $\mathcal{M}_i = T^{(1+j\eta)}$ is marked with the text position $pos(i) = 1 + j\eta$. To determine whether a row is marked we could store all marked pairs $(i, 1 + j\eta)$ in a hash map or a binary search tree with key i.

Instead we can again use our $O(1)$ rank-query supported bitvector.

11.16 Compressing *pos*

We can use a rank-query supported bitvector B_{pos} in conjunction with an array Pos of size n/η.

If we still have the suffix array during the construction of the BWT, we can simply scan through the array maintaining an index k which initialize to 0. Whenever $A[i]/\eta \equiv 0 \mod n$, we mark the $i - th$ Bit in the Bitvector, store $Pos[k] = A[i]$ and increment k.

If the suffix array is not given, we use the BWT and L-to-F mapping traverse the BWT as in the reconstruction algorithm. While doing this, we keep counting the number of steps. After η backwards steps, we are at textposition $n - \eta$ and hence we mark the bitvector.

After setting all bits we again traverse the BWT maintaining a counter m which we initialize with 0. Whenever the bitvector is set we increment m, obtain the rank k of the bit and set $Pos[k] = n - m \cdot \eta$