5 Computational paradigms used in read mapping

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

1. Li, H, and Homer, N. (2010) *A survey of sequence alignment algorithms for next-generation sequencing*. Briefings in Bioinformatics 11 (5) (September 21): 473-483.
2. Holtgrewe, M., Emde A.-K., Weese D., Reinert K. (2011) *A Novel And Well-Defined Benchmarking Method For Second Generation Read Mapping.* BMC Bioinformatics 12 (1): 210.

5 Computational paradigms used in read mapping

The term *read mapping* has itself established since a couple of years for another, well studied problem, namely *approximate string matching* with certain application driven constraints.

The constraints are:

- usually DNA (or RNA) is considered.
- we have to map short reads (about 50 to 3000 bases) to a large string (billions of bases).
- there are relatively few errors allowed (usually around 3-4%, some application might go up to 10%).
- the problem sizes are very large (hundreds of millions of reads map to a string of size up to several billion characters).

5.1 Objective functions

Depending on the application, the term read mapping subsumes a number of different objective functions. In the normal case we want to find the approximate occurrences of a complete read, that is, conduct a *semi-global* alignment.

If we have for example RNA reads, then the read may corresponds to several genomic loci that have been spliced together. In this case we speak of *split* alignment to distinguish it from local alignment. In split alignment we want to find the *complete* read, whereas this is not necessary in local alignment.

The problem of split alignment can be further subdivided depending on the decision whether we allow parts of the split read to be reverse complemented (e.g. assembly error), be missing, or out of order (e.g. genomics insertions or deletions).

Usually split read mapping is reduced to several subproblems of normal read mapping.

Finally, a distinction is made whether the approximate string matching supports (weighted) edit distance or only the Hamming distance.

While the edit distance is preferable, it makes the problem computationally harder. Often you will find in read mapping heuristics some ”in-between” formulations (e.g. *supports mismatches and up to 2 insertions*).

Be aware of such limitations.

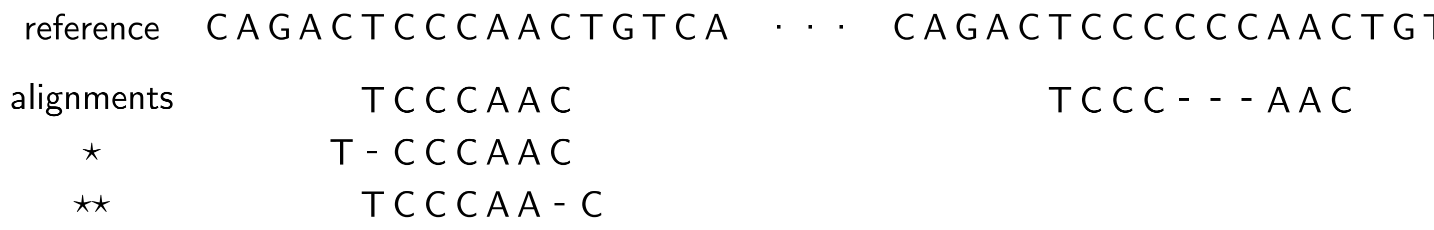
If we have now a fixed objective function for our special approximate string matching problem, we can still make distinctions about the set of matches we want to find. A reasonable distinction could be the tasks of finding:

1. *all* matches with up to k errors.
2. *all* best matches.
3. *any* best match.

Doing this of course implies to have a good definition, what we actually mean with a match?

Have a look at the following situation:

5.2 Benchmarking

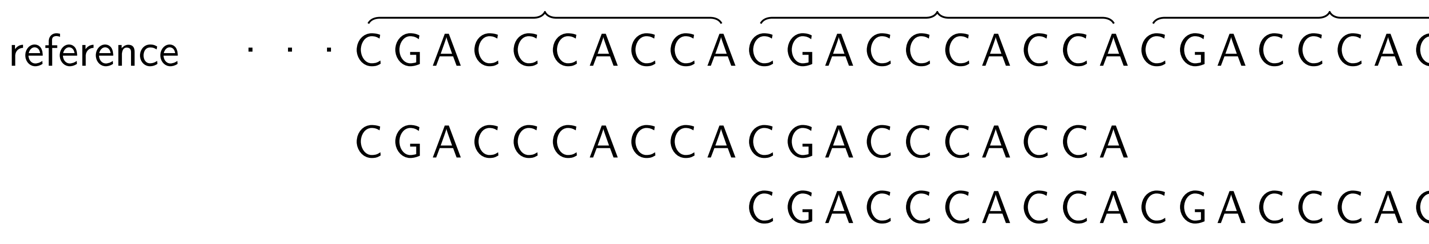


Different kind of approximate matches.

Say, we want to find the best two matches of the read in the reference sequence, with an edit distance of up to 3. Both locations in the reference sequence are shown. The row alignments shows two alignments of the read to the reference sequence that appear to be optimal. However, the alignments in the rows below have a lower edit distance than the right one.

Common sense would tell us that the alignments in the left column are not significantly different, though. Each alignment with distance k induces alignments with distance at most $k + 2$ by aligning the leftmost/rightmost base one more position to the left/right and introducing a gap.

Repeats are another issue. Consider the tandem repeats in the below figure.


reference
· · · C G A C C C A C C A C G A C C C A C C A C G A C C C A
C G A C C C A C C A C G A C C C A C C A
C G A C C C A C C A C G A C C C A

Large period repeat.

Intuitively, we can identify the two distinct alignments in this situation. Now look at a tandem repeat with a shorter period:

$$\text{reference}\quad \cdots\ \overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\,\overbrace{\text{C A A}}\ \cdots$$

```
CAACAACAACAA
   CAACAACAACAA
      CAACAACAACAA
         CAACAACAACAA
            CAACAACAACAA
               CAACAACAACAA
```

Small period repeat.

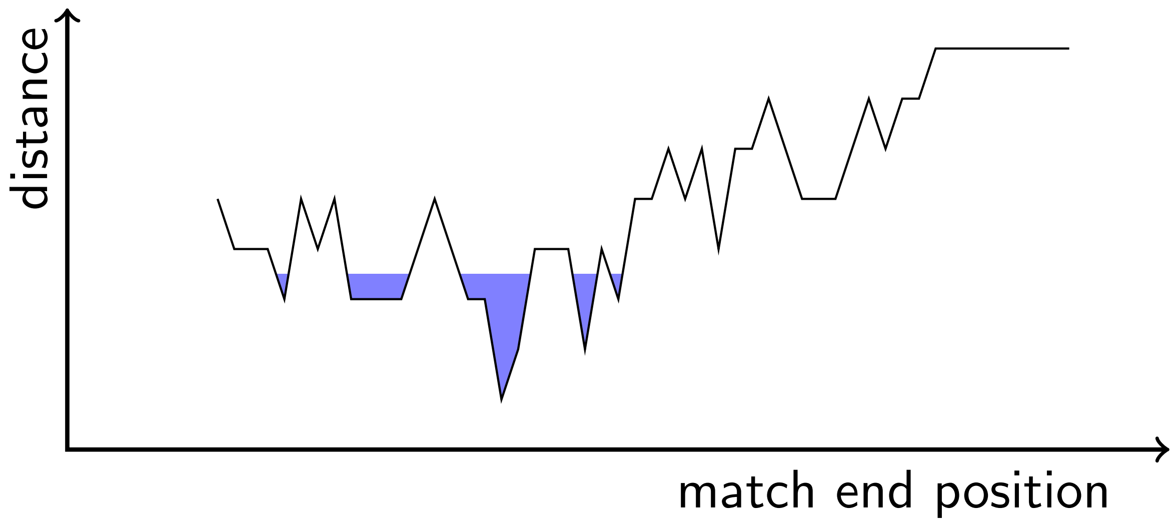
Do we really want to find all those alignments?

Counting alignments in this way would require a read mapper to find lots of positions in repeat regions. This is not desirable since reads from long tandem repeat regions would get a higher weight with this counting scheme than reads from short tandem repeat regions or reads from non-repeat regions.

Only weighting each found match with $1/n$ (where n is the number of positions the read aligns at) is also deficient (why?).

Hence it is more desirable to define when two matches are considered the same and when they should be counted separately.

Without giving the details, one can define an equivalence relation on the set of matches, which can be depicted as follows as an *error landscape*.



Error landscape.

Flooding this landscape to the respective error level gives a number of intervals, which in turn can be used to define the specificity and sensitivity of read mappers. In the benchmark it is sufficient to return one endposition within the interval.

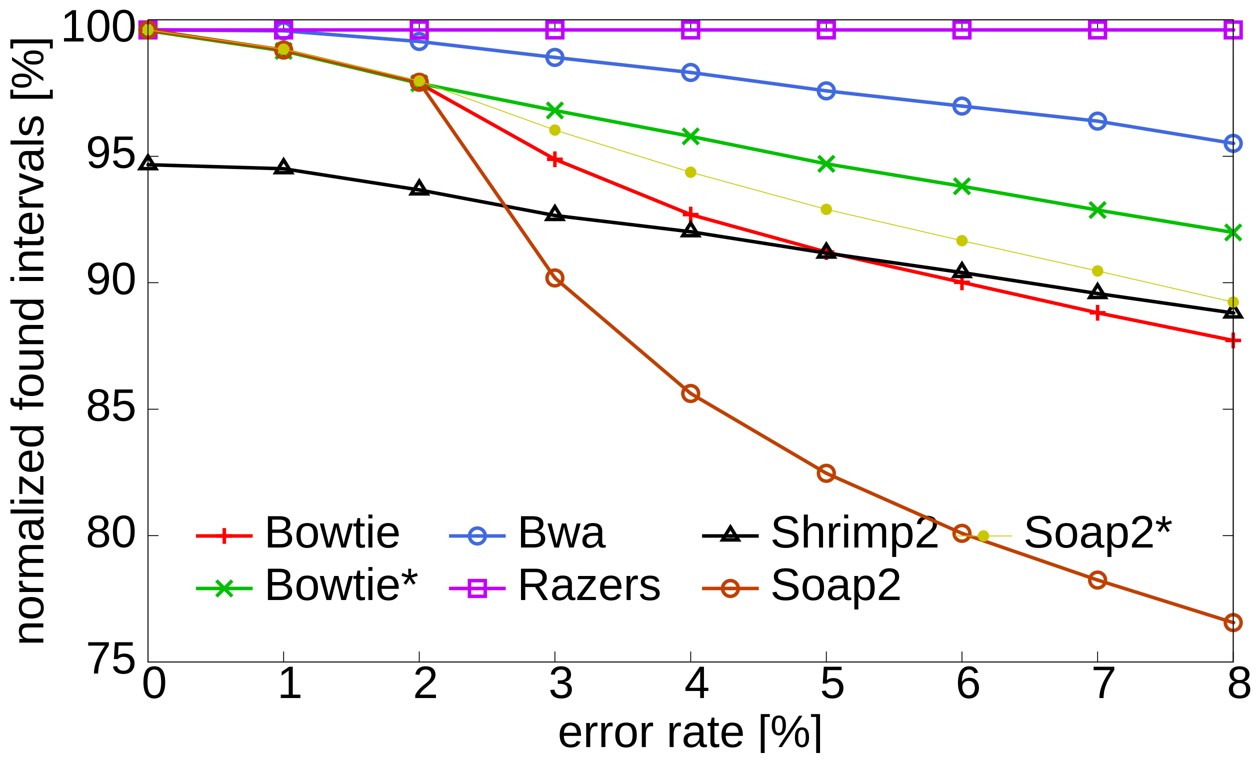
Some of the intervals will be labelled *optimal*, if they contain a matching position with the minimal distance (e.g. edit or hamming). If we benchmark read mapping application with the goal *find any best*, then it should return an alignment ending in one of those intervals. If we have the goal *find all best*, the read mapper should return all optimal intervals, etc.

This can now be used to make comprehensive comparisons between different methods to compare their performance.

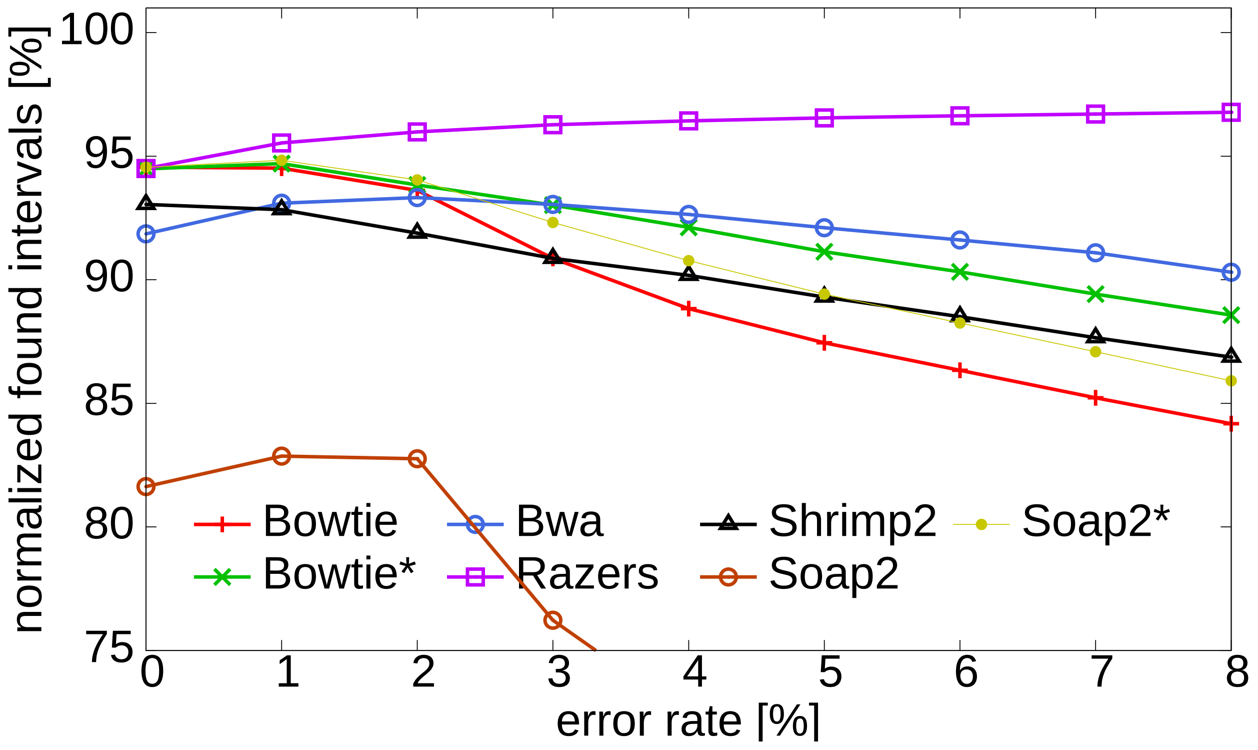
As an evaluation metric we use the number of *normalized found intervals*.

This is defined as follows: Each read gives at most one point. If a read matches at n locations (i.e. intervals), each found location gives $1/n$ point. To get percentages, the number of achieved points is divided by the number of reads and multiplied by 100.

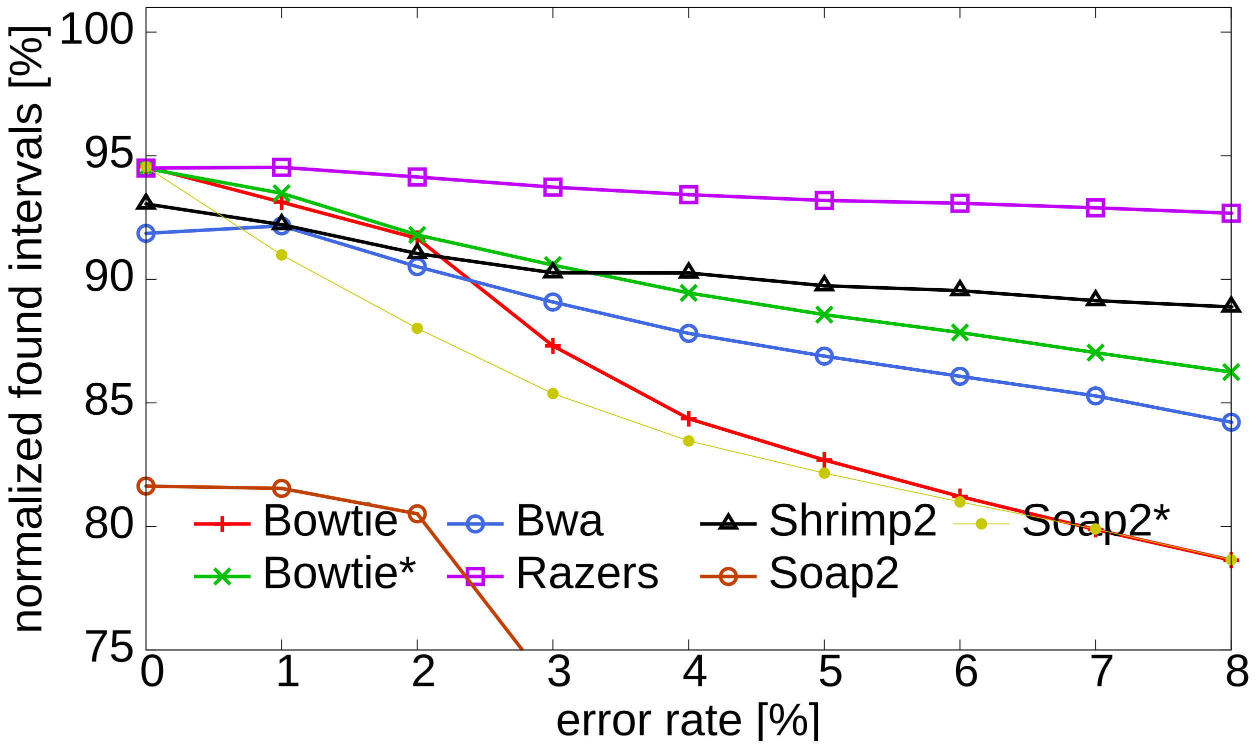
Have a look at results of recent read mappers (2011) for the three different categories (Illumina reads of *Drosophila Melanogaster*, 100 bp length), but mind that those plots do not give the run times.



Benchmark *any best*.



Benchmark *all best*.



Benchmark *all*.

5.3 Computational paradigms

Lets go back to algorithmic paradigms used in read mapping algorithms.

Given the large data, obviously all algorithms use some *string indices* to preprocess the reads, the genome, or both. The indices can be used directly for searching as in the case of the enhanced suffix array or Burrows Wheeler transform, or they are used to filter out regions that do not contain matches (as in the case of (gapped) q-gram indices).

You have already encountered a simple filter that is based on a q-gram index and uses a simple version of the q-gram lemma. This paradigm is called *q-gram counting*.

5.4 Computational paradigms

In the following lectures you will

- learn what the BWT (Burrows-Wheeler transform) is, and what advantages is has compared to an ESA.
- learn how a q-gram filter can be used in a pidgeonhole based filter.

The BWT is the basis of the currently fastest read mapping applications. In the lecture we will talk about how to construct it, and how to search in it exactly.

Inexact searches can be implemented by a backtracking procedure.

5.5 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

5.6 Motivation

You know that a *suffix array* is 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 $O(m + p)$ 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 for example in bzip2).

5.7 Motivation

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. Moreover, one big advantage of the BWT is, that it gives you a *searchable* data structure that need per entry a *character* (e.g. 2 bit in the case of DNA), whereas an ESA needs per entry an integer (e.g. 64 bit in case of a large text).

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

5.8 Definitions

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

- $[i..j] := \{i, i+1, \dots, j\}$
- $[i..j) := [i..j-1]$
- $T[i]$ is the i-th character of T.
- $T[i..j] := T[i]T[i+1]\dots T[j]$ 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.

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

5.9 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:

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

sort $\Rightarrow$

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*

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 (we will not elaborate on that).

5.10 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	$T^{(i)}$
1	`mississippi$`
2	`ississippi$m`
3	`ssissippi$mi`
4	`sissippi$mis`
5	`issippi$miss`
6	`ssippi$missi`
7	`sippi$missis`
8	`ippi$mississ`
9	`ppi$mississi`
10	`pi$mississip`
11	`i$mississipp`
12	`$mississippi`

sort $\Rightarrow$

i	$T^{(i)}$
12	`$mississippi`
11	`i$mississipp`
8	`ippi$mississ`
5	`issippi$miss`
2	`ississippi$m`
1	`mississippi$`
10	`pi$mississip`
9	`ppi$mississi`
7	`sippi$missis`
4	`sissippi$mis`
6	`ssippi$missi`
3	`ssissippi$mi`

$\widehat{=}$

A	$T[A[j]..n]$
12	`$`
11	`i$`
8	`ippi$`
5	`issippi$`
2	`ississippi$`
1	`mississippi$`
10	`pi$`
9	`ppi$`
7	`sippi$`
4	`sissippi$`
6	`ssippi$`
3	`ssissippi$`

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

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

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. $\mathcal{M}_i = T^{(A[i])}$.*

5.11 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:

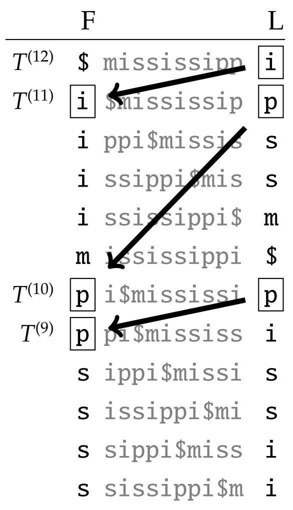
Definition 7 (L-to-F mapping)**.** Let $\mathcal{M}$ be the sorted matrix of cyclic shifts of the text T. LF is a function $\mathsf{LF} : [1..n] \to [1..n]$ that maps the rank of a cyclic shift X to the rank of $X^{(n)}$ which is X shifted by one to the right:

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

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 $\mathsf{LF}(7) = 8$.



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.

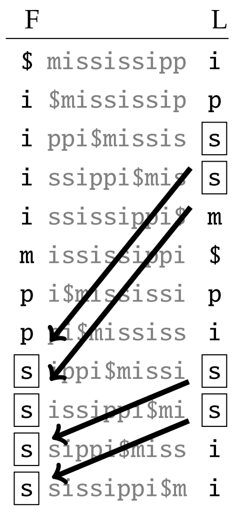
Observation 9 (rank preservation)**.** Let $i, j \in [1..n]$ with $L[i] = L[j]$. If $i < j$ then $\mathsf{LF}[i] < \mathsf{LF}[j]$ follows.

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:

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

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 $9 < 10 < 11 < 12$.



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

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:*

$$\mathsf{LF}(i) = C(L[i]) + \mathsf{Occ}(L[i], i)$$

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 $F[\mathsf{LF}(1)]$ is the character before the last one in T.
- The character $T[n-i]$ is $F[\underbrace{\mathsf{LF}(\mathsf{LF}(\ldots\mathsf{LF}(1)\ldots))}_{i \text{ times LF}}]$.

For the reverse transform we need F and the functions C and Occ. C and F 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 $L = T^{\mathrm{bwt}}$ is given in algorithm **reverse_transform**. We replaced F by L using $F[\mathsf{LF}(i)] = L[i]$ and $F[1] = \$$.

```
(1) // reverse_transform(L,Occ,C)
(2) $i = 1, j = n, c = \$$;
(3) while ($j > 0$) do
(4)         $T[j] = c$;
(5)         $c = L[i]$;
(6)         $i = C(c) + \mathsf{Occ}(c, i)$;
(7)         $j = j - 1$;
(8) od
(9) return $T$;
```

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

F is the concatenated sequence of runs of n_α many characters α in increasing order:

$$\begin{aligned} F &= \$^{n_\$} \cdot \mathtt{i}^{n_\mathtt{i}} \cdot \mathtt{m}^{n_\mathtt{m}} \cdot \mathtt{p}^{n_\mathtt{p}} \cdot \mathtt{s}^{n_\mathtt{s}} \\ &= \mathtt{\$iiiimppssss} \end{aligned}$$

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
$\mathsf{Occ}(L[i], i)$	1	1	1	2	1	1	2	2	3	4	3	4

Begin in row $i = 1$:
Extract the character $T[n] = F[1] = \$$
$T =$ . . . . `$`

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[1]) + \mathsf{Occ}(L[1], 1) = 1 + 1 = 2$
Extract the character $T[n-1] = F[2] =$ `i`
$T =$ . . . . `i$`

Proceed with row $i = \mathsf{LF}(i)$:
$i = C(L[2]) + \mathsf{Occ}(L[2], 2) = 6 + 1 = 7$
Extract the character $T[n-2] = F[7] =$ `p`
$T =$ . . . . `pi$`

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[7]) + \mathsf{Occ}(L[7], 7) = 6 + 2 = \mathit{8}$
Extract the character $T[n-3] = F[8] = \texttt{p}$
$T = \texttt{........ppi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[8]) + \mathsf{Occ}(L[8], 8) = 1 + 2 = \mathit{3}$
Extract the character $T[n-4] = F[3] = \texttt{i}$
$T = \texttt{.......ippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[3]) + \mathsf{Occ}(L[3], 3) = 8 + 1 = \mathit{9}$
Extract the character $T[n-5] = F[9] = \texttt{s}$
$T = \texttt{......sippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[9]) + \mathsf{Occ}(L[9], 9) = 8 + 3 = \mathit{11}$
Extract the character $T[n-6] = F[11] = \texttt{s}$
$T = \texttt{.....ssippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[11]) + \mathsf{Occ}(L[11], 11) = 1 + 3 = \mathit{4}$
Extract the character $T[n-7] = F[4] = \texttt{i}$
$T = \texttt{....issippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[4]) + \mathsf{Occ}(L[4], 4) = 8 + 2 = \mathit{10}$
Extract the character $T[n-8] = F[10] = \texttt{s}$
$T = \texttt{...sissippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[10]) + \mathsf{Occ}(L[10], 10) = 8 + 4 = \mathit{12}$
Extract the character $T[n-9] = F[12] = \texttt{s}$
$T = \texttt{..ssissippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[12]) + \mathsf{Occ}(L[12], 12) = 1 + 4 = \mathit{5}$
Extract the character $T[n-10] = F[5] = \texttt{i}$
$T = \texttt{.ississippi\$}$

Proceed with row $i = \mathbf{LF}(i)$:
$i = C(L[5]) + \mathsf{Occ}(L[5], 5) = 5 + 1 = \mathit{6}$
Extract the character $T[1] = F[6] = \texttt{m}$
$T = \texttt{mississippi\$}$

5.12 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:

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

(Mind that we now need the Occ table for *every* character at every position.)

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 applying the LF mapping to the first and last occurrence of c in the interval:

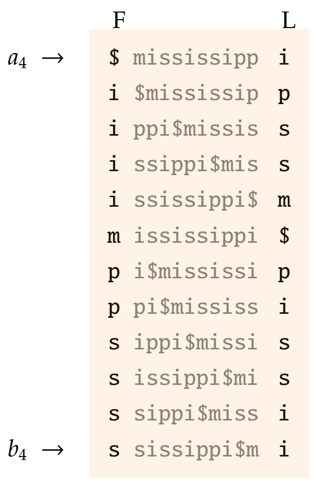
$$\begin{aligned} a_{j-1} &= C(c) + \mathsf{Occ}(c, a_j - 1) + 1 \\ b_{j-1} &= C(c) + \mathsf{Occ}(c, b_j) \end{aligned}$$

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

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

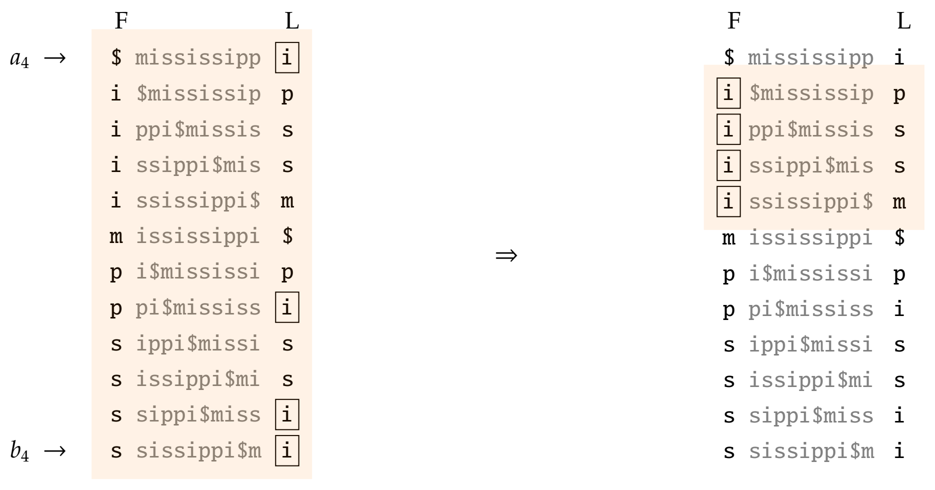
Initialization: We begin with the empty suffix ϵ which is a prefix of every suffix, hence we initialize $a_{m+1} = 1$ and $b_{m+1} = n$.

```
// count(P[1..m])
$i = m, a = 1, b = n;$
while (($a < b$) $\wedge$ ($i \geq 1$)) do
        $c = P[i];$
        $a = C(c) + \mathrm{Occ}(c, a-1) + 1;$
        $b = C(c) + \mathrm{Occ}(c, b);$
        $i = i - 1;$
od
if ($b < a$) then return "not found";
            else return "found ($b - a + 1$) occurrences";
fi
```



$a_4 = 1$
$b_4 = 12$

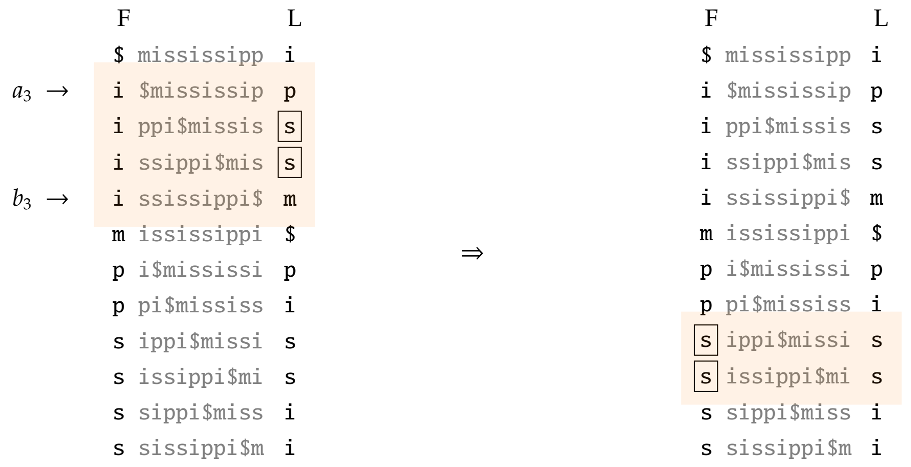
Searching P_3: From all matrix rows we search those beginning with the last pattern character $P[3] = \texttt{i}$. From $\mathrm{Occ}(x, 0) = 0$ and $\mathrm{Occ}(x, n) = n_x$ follows $a_m = C(x) + 1$ and $b_m = C(x + 1)$.



$a_4 = 1$
$b_4 = 12$

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

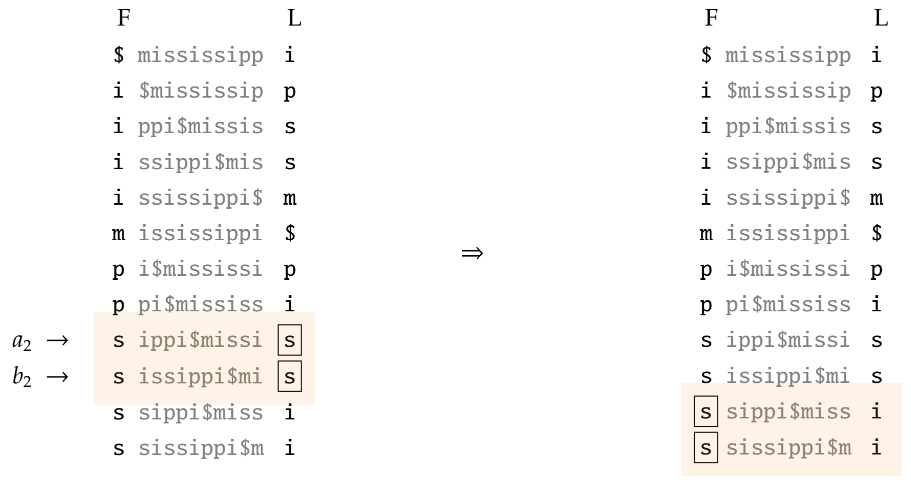
Searching P_2: From all rows beginning with P_3 we search those whose cyclic shift begins with $P[2] = \texttt{s}$. In L we count the s's in the part before the interval (=0) and including the interval (=2) to L-to-F map the first and last s in the interval.



$a_3 = 2$
$b_3 = 5$

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

Searching P_1: From all matrix rows beginning with P_2 we search those whose cyclic shift begins with $P[1] =$ s.



$a_2 = 9$
$b_2 = 10$

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

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
```

$a_1 = 11$
$b_1 = 12$

5.13 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, we would be able to get the corresponding text position $pos(i)$ of the suffix in row i *if* we have a suffix array A of T. Then it holds $pos(i) = A[i]$.

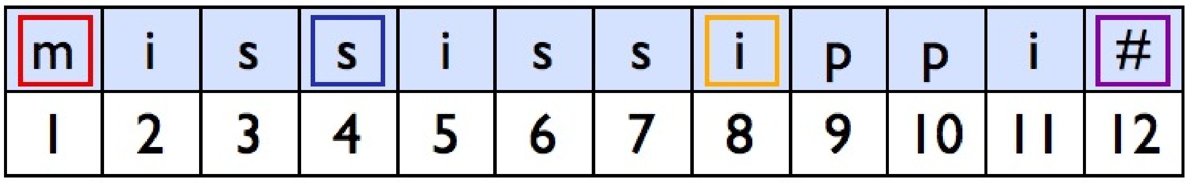
We will now see that it is not necessary to have given the whole suffix array of $4n$ bytes of memory. It suffices to have a fraction of the suffix array available to compute $pos(i)$ for every $i \in [1..n]$.

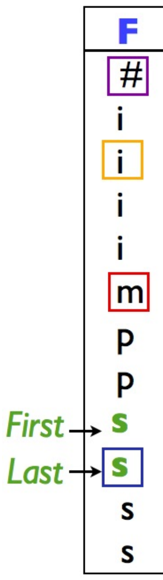
The idea is as following. We logically mark a suitable subset of rows in the matrix. For the marked rows we explicitly store the start positions of the suffixes in the text. If i is marked, it means that row $pos(i)$ is directly available. If i is not marked, the algorithm **locate** uses the L-to-F-mapping 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:
$pos(1) = 12$
$pos(3) = 8$
$pos(6) = 1$
$pos(10) = 4$





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 $T[1..n]$.

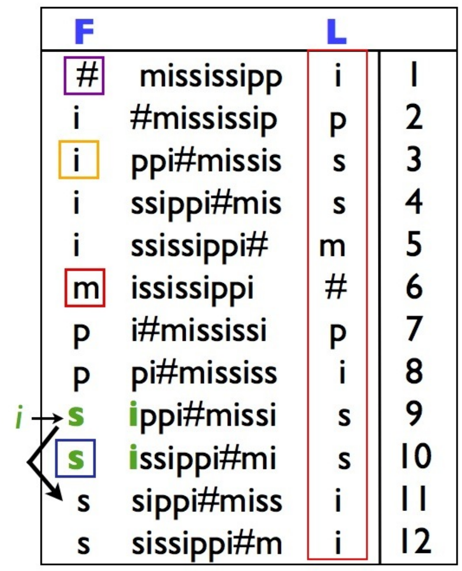
```
(1) // locate(i)
(2) $i' = i$
(3) $v = 0$;
(4) while (row $i'$ is not marked) do
(5)         $c = L[i']$;
(6)         $i' = C(c) + \mathrm{Occ}(c, i')$;
(7)         $v = v + 1$;
(8) od
(9) return $pos(i') + v$;
```

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*.

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**

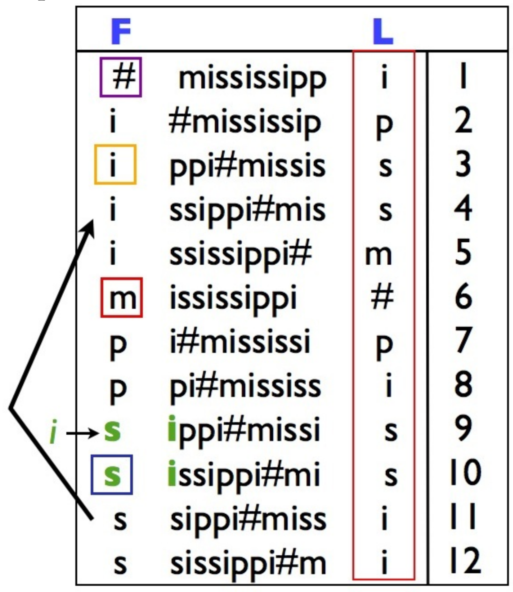
Step 1



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
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

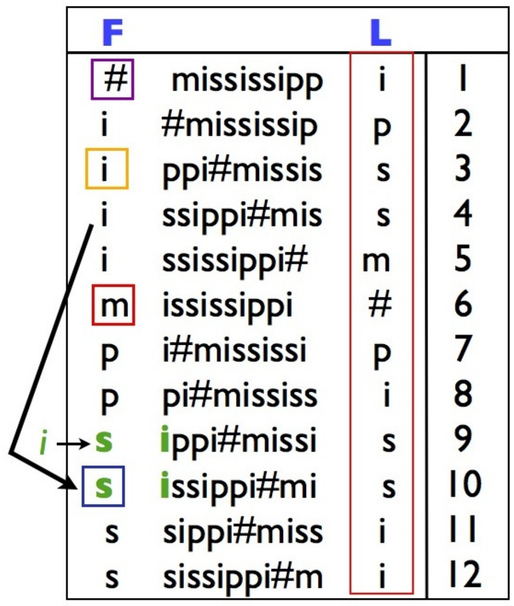
Step 2



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
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
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 running 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.

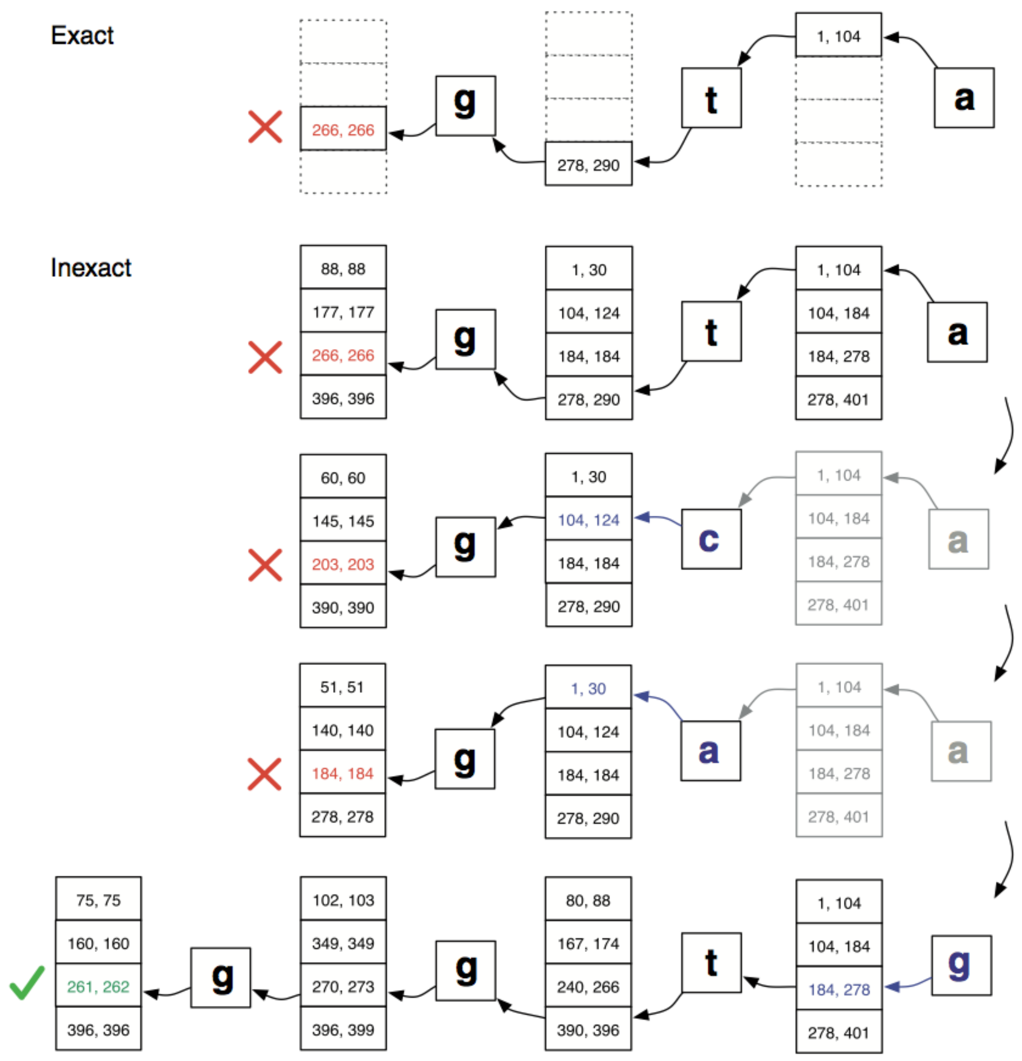
For example (for DNA) using the the text T (n bytes), the BWT (n bytes resp. $n/4$ bytes), the *Occ* table (e.g. $4 \cdot 4 \cdot n/32$ bytes when storing only every 32th entry) and a sampled suffix array (e.g. $4 \cdot n/8$ bytes when marking every 8th entry). In our example calculation we would need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

(The BWT itself and the tables can be further compressed which will be adressed in MSc lectures.)

5.14 Backtracking

The FM index allows us to conduct an exact search in *linear* time in the pattern length. However, how would we conduct an inexact search?

The answer is (as in the case for enhanced suffix arrays), by *backtracking*. The below example from the BowTie paper shows the principle. In the backtracking one can use quality values to heuristically limit the number of possible paths.



Example of inexact search of `ggta` which occurs with one mismatch. (Mind the different interval definition. Here x, x denotes an empty interval.)