UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

The FM Index

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Reminder about the FM Index: Backward Search with the BWT

Ferragina and Manzini, Opportunistic Data Structures with Applications, 2000

Backward Search with BWT, Occ and C

$s = $ ctatatat$ and bwt=tttt$aaac:

C [$]	C [a]	C [c]	C [t]
0	1	4	5

	t	t	t	t	$	a	a	a	c
	0	1	2	3	4	5	6	7	8
Occ [$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

	index	*bwt*	*S[pos[i]]*
	0	**t**	*$*
	1	**t**	*at$*
	2	**t**	*atat$*
	3	**t**	*atatat$*
	4	**$**	*ctatatat$*
→	*5*	**a**	*t$*
	6	**a**	*tat$*
	7	**a**	*tatat$*
→	*8*	**c**	*tatatat$*

$\texttt{Occ}(c, r)$ returns the number of occurrences of $c \in \Sigma$ in the prefix $\texttt{bwt}[0 \dots r]$.
The k-th c in the BWT is the k-th c in the first characters of the sorted suffixes.
$LF(r) = \texttt{C}[c] + \texttt{Occ}(c, r) - 1$

Backward Search with BWT, Occ and C

$s =$ ctatatat\$ and bwt=tttt\$aaac:

C [\$]	C [a]	C [c]	C [t]
0	1	4	5

	t	t	t	t	\$	a	a	a	c
	0	1	2	3	4	5	6	7	8
Occ [\$]	0	0	0	0	1	1	1	1	1
Occ [a]	0	0	0	0	0	1	2	3	3
Occ [c]	0	0	0	0	0	0	0	0	1
Occ [t]	1	2	3	4	4	4	4	4	4

index	*bwt*	*S[pos[i]]*
0	**t**	*\$*
1	**t**	*at\$*
2	**t**	*atat\$*
3	**t**	*atatat\$*
4	**\$**	*ctatatat\$*
→ *5*	**a**	*t\$*
6	**a**	*tat\$*
7	**a**	*tatat\$*
→ *8*	**c**	*tatatat\$*

Let $[i, j]$ be the interval for μ; let $[i', j']$ be the interval for $c\mu$. Then

$$\begin{aligned} i' &= \mathtt{C}[c] + \mathtt{Occ}(c, i-1) \\ j' &= \mathtt{C}[c] + \mathtt{Occ}(c, j) - 1 \end{aligned}$$

Implementation of `Occ`

As shown, `Occ` uses $O(|\Sigma| n)$ words of space – worse than the suffix tree!
(A word is a number of size $O(\text{poly}(n))$, i.e., using $O(\log n)$ bits.)

Implementation of `Occ`

As shown, `Occ` uses $O(|\Sigma|n)$ words of space – worse than the suffix tree!
(A word is a number of size $O(\text{poly}(n))$, i.e., using $O(\log n)$ bits.)

Simple Idea

Store the entries of `Occ` only for every k-th position (typically $k \in \{32, 64, 128\}$).
To obtain $\texttt{Occ}(a, r)$, look up $\texttt{Occ}(a, \lfloor r/k \rfloor)$,
and count the as in the remaining $r - \lfloor r/k \rfloor$ characters in `bwt`.

Implementation of `Occ`

As shown, `Occ` uses $O(|\Sigma|n)$ words of space – worse than the suffix tree!
(A word is a number of size $O(\text{poly}(n))$, i.e., using $O(\log n)$ bits.)

Simple Idea

Store the entries of `Occ` only for every k-th position (typically $k \in \{32, 64, 128\}$).
To obtain $\texttt{Occ}(a, r)$, look up $\texttt{Occ}(a, \lfloor r/k \rfloor)$,
and count the as in the remaining $r - \lfloor r/k \rfloor$ characters in `bwt`.

	t	**t**	**t**	**t**	**$**	**a**	**a**	**a**	**c**
	0	1	2	3	4	5	6	7	8
Occ [$]	0				1				1
Occ [a]	0				0				3
Occ [c]	0				0				1
Occ [t]	1				4				4

`Occ` Queries in Constant Time and in Small Space

Implementation of `Occ`: Advanced Ideas

Data Structures

- Succinct data structure for rank queries (binary alphabet)
- Wavelet tree (for converting alphabet to sequence of binary alphabets)
- Wavelet matrix (alternative to wavelet tree)

Implementation of `Occ`: Advanced Ideas

Data Structures

- Succinct data structure for rank queries (binary alphabet)
- Wavelet tree (for converting alphabet to sequence of binary alphabets)
- Wavelet matrix (alternative to wavelet tree)

Rank queries

Let $rank_{s,a}(i)$ be the number of as in $s[0 \ldots i]$, i.e., $\texttt{Occ}(a, i)$ for s.

Next goal

Rank queries in $O(1)$ time for s over binary alphabet $\{0, 1\}$ with $o(n)$ additional bits, i.e., for binary s, compute $rank_s(i) := rank_{s,1}(i)$ in constant time for all i.

Implementation of `Occ`: Advanced Ideas

Data Structures

- Succinct data structure for rank queries (binary alphabet)
- Wavelet tree (for converting alphabet to sequence of binary alphabets)
- Wavelet matrix (alternative to wavelet tree)

Rank queries

Let $rank_{s,a}(i)$ be the number of as in $s[0 \dots i]$, i.e., $\texttt{Occ}(a, i)$ for s.

Next goal

Rank queries in $O(1)$ time for s over binary alphabet $\{0, 1\}$ with $o(n)$ additional bits, i.e., for binary s, compute $rank_s(i) := rank_{s,1}(i)$ in constant time for all i.
Note: $rank_{s,0}(i) = (i + 1) - rank_{s,1}(i)$.

First Observations

$rank_s(i)$: number of ones in $s[\dots i]$ for $i = 0, \dots, n-1$, where $n = |s|$

Trivial ideas

- Slow, but lightweight: $O(n)$ time, $O(1)$ additional memory
- Slightly faster: loop with popcount: $O(n/W)$ time, $O(1)$ additional memory (popcount: number of 1-bits in a machine word, elementary instruction)
- Fast but heavy-weight: full `Occ` table: $O(1)$ time, but $O(n \log n)$ bits (n words)
- Slightly slower, but lighter: sparse table (every k-th entry): $O(k)$ time and $O(n/k \cdot \log n)$ bits (n/k words)

First Observations

$rank_s(i)$: number of ones in $s[\dots i]$ for $i = 0, \dots, n-1$, where $n = |s|$

Trivial ideas

- Slow, but lightweight: $O(n)$ time, $O(1)$ additional memory
- Slightly faster: loop with popcount: $O(n/W)$ time, $O(1)$ additional memory (popcount: number of 1-bits in a machine word, elementary instruction)
- Fast but heavy-weight: full `Occ` table: $O(1)$ time, but $O(n \log n)$ bits (n words)
- Slightly slower, but lighter: sparse table (every k-th entry): $O(k)$ time and $O(n/k \cdot \log n)$ bits (n/k words)

Desired

Data structure that supports $O(1)$ time, but needs only $o(n)$ bits, i.e., if $x(n)$ is the additional number of bits (plus n for s), we want $x(n)/n \to 0$ for $n \to \infty$.

A small rank data structure for (binary) Occ

Basic Idea

- Store every S-th entry, so S bits form a **superblock**:
  $O(\log n \cdot n/S)$ bits for superblock table

A small rank data structure for (binary) `Occ`

Basic Idea

- Store every S-th entry, so S bits form a **superblock**:
  $O(\log n \cdot n/S)$ bits for superblock table
- Choose $S := \Theta((\log n)^2)$; so need $\mathcal{O}(n/\log n) = o(n)$ bits
- Remaining problem: Count ones in superblocks of size S

A small rank data structure for (binary) Occ

Basic Idea

- Store every S-th entry, so S bits form a **superblock**:
  $O(\log n \cdot n/S)$ bits for superblock table
- Choose $S := \Theta((\log n)^2)$; so need $\mathcal{O}(n/\log n) = o(n)$ bits
- Remaining problem: Count ones in superblocks of size S
- So far: time $O(\log^2 n)$; memory $o(n)$ bits

Refinement

- Partition each superblock into $\Theta(\log n)$ **blocks** of size $B := \Theta(\log n)$
- Each superblock has a table with rank differences for each block start.
- Values up to $\Theta(\log^2 n)$ need $O(\log\log n)$ Bits.
- Number of blocks is $\Theta(\log n \cdot n/S) = \Theta(n/\log n)$.
- Total size: $\mathcal{O}(n \log\log n/\log n) = o(n)$ Bits.

Answering Rank Queries in Constant Time

Query $rank_s(i)$

- Given i, compute index s of superblock and index b of block inside superblock, such that $i = s \cdot S + b \cdot B + j$ with $0 \leq j < B$.
- Look up rank R_s for superblock s in first table
- Look up rank difference $r_{s,b}$ for block b in second table
- Compute number of ones $r'_{s,b,j}$ in remaining j bits; constant time with bitmask and popcount, because $j < B = \Theta(\log n)$
- Answer is $R_s + r_{s,b} + r'_{s,b,j}$: sum of three terms, each in constant time.

Practical Implementation

- Theory (RAM model): popcount of $O(\log n)$ bits in constant time
- Practical popcount of up to 64 Bits in constant time
- Choose $B := 64 = \Theta(\log n)$, assume $n \leq 2^{64}$
- Choose $S := 16 \cdot (64)^2 = 65536 = 2^{16} = \Theta((\log n)^2)$
- 64-bit ints for superblock ranks, 16-bit ints for block ranks
- Values can be adjusted for different n, but these choices are convenient.

Practical Implementation

- Theory (RAM model): popcount of $O(\log n)$ bits in constant time
- Practical popcount of up to 64 Bits in constant time
- Choose $B := 64 = \Theta(\log n)$, assume $n \leq 2^{64}$
- Choose $S := 16 \cdot (64)^2 = 65536 = 2^{16} = \Theta((\log n)^2)$
- 64-bit ints for superblock ranks, 16-bit ints for block ranks
- Values can be adjusted for different n, but these choices are convenient.

- We have $n/2^{16}$ superblocks with 64-bit rank values
- Each superblock has 1024 blocks (64 bits) with 16-bit rank values
- Total: $n/65536 \cdot (64 + 1024 \cdot 16) \approx 0.25 \cdot n$ bits

From Binary to General Alphabet

Simple but Wasteful Idea

Use one bit vector per letter to represent BWT;
compute a separate succinct rank data structure for each letter.

Simple but Wasteful Idea

Use one bit vector per letter to represent BWT;
compute a separate succinct rank data structure for each letter.

Example: $T =$ `banana$`, $bwt =$ `annb$aa`

Bits	0	1	2	3	4	5	6
$	0	0	0	0	1	0	0
a	1	0	0	0	0	1	1
b	0	0	0	1	0	0	0
n	0	1	1	0	0	0	0

Simple but Wasteful Idea

Use one bit vector per letter to represent BWT;
compute a separate succinct rank data structure for each letter.

Example: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Bits	0	1	2	3	4	5	6
\$	0	0	0	0	1	0	0
a	1	0	0	0	0	1	1
b	0	0	0	1	0	0	0
n	0	1	1	0	0	0	0

Wasteful: Needs $O(n|\Sigma|) + o(n|\Sigma|)$ bits.
But BWT itself only needs $O(n \log |\Sigma|)$ bits!

Wavelet Tree

Definition (Wavele tree)

Let T be a text with $|T| = n$; let $\Sigma = \{0, \ldots, s-1\}$ be an alphabet with $|\Sigma| = s$.
The wavelet tree for T is a balanced binary tree with s leaves.

Wavelet Tree

Definition (Wavele tree)

Let T be a text with $|T| = n$; let $\Sigma = \{0, \ldots, s-1\}$ be an alphabet with $|\Sigma| = s$.
The wavelet tree for T is a balanced binary tree with s leaves.
Each node corresponds to a sub-interval of the alphabet.
The root corresponds to Σ; each leaf corresponds to a single character.

Wavelet Tree

Definition (Wavele tree)

Let T be a text with $|T| = n$; let $\Sigma = \{0, \ldots, s-1\}$ be an alphabet with $|\Sigma| = s$.
The wavelet tree for T is a balanced binary tree with s leaves.
Each node corresponds to a sub-interval of the alphabet.
The root corresponds to Σ; each leaf corresponds to a single character.

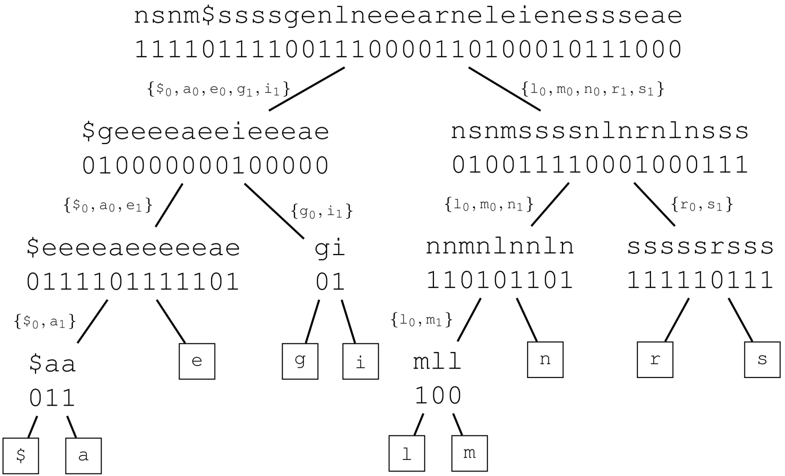
Each non-leaf node represents the sub-sequence of T whose characters are in a sub-alphabet $\{a, \ldots, b\}$. It partitions the sub-alphabet into two parts of equal size:

- lower alphabet $a, \ldots, \lfloor (a+b)/2 \rfloor$,
- upper alphabet $\lfloor (a+b)/2 \rfloor + 1, \ldots, b$.

A bit vector indicates which letter belongs to which sub-sub-sequence.

Wavelet Tree: Example

$$\Sigma = \{\$_0, a_0, e_0, g_0, i_0, l_1, m_1, n_1, r_1, s_1\}$$



Properties of the Wavelet Tree

- There are $\log |\Sigma|$ levels in the wavelet tree.
- In each level, we have n bits (summed over all nodes in the level).
- The wavelet tree thus needs $n \cdot \lceil \log |\Sigma| \rceil$ bits, as T does.
- An additional $O(|\Sigma| \log n)$ bits are required for representing the tree structure.

Properties of the Wavelet Tree

- There are $\log|\Sigma|$ levels in the wavelet tree.
- In each level, we have n bits (summed over all nodes in the level).
- The wavelet tree thus needs $n \cdot \lceil \log|\Sigma| \rceil$ bits, as T does.
- An additional $O(|\Sigma| \log n)$ bits are required for representing the tree structure.
- With a rank data structure for each node, we need an additional $o(n \log|\Sigma|)$ bits.
- With these, we can answer character and rank queries in $O(\log|\Sigma|)$ time.
- We can replace T by the wavelet tree (and delete T).

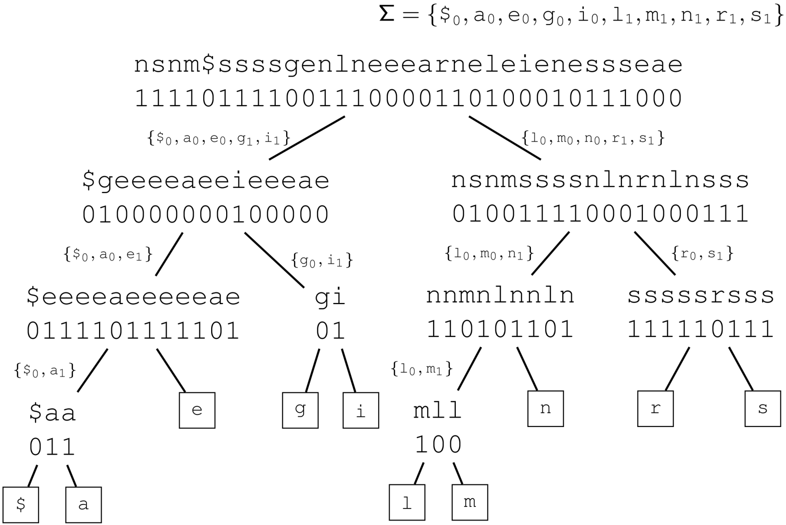
Rank Queries on the Wavelet Tree

- Root query: $rank_\sigma(i)$
- In the root, is σ in the lower or upper alphabet (0-bit or 1-bit) ?
  - 0-bit: Compute $k := i - rank_1(i) + 1$, go to left child.
  - 1-bit: Compute $k := rank_1(i)$, go to right child.

Rank Queries on the Wavelet Tree

- Root query: $rank_\sigma(i)$
- In the root, is σ in the lower or upper alphabet (0-bit or 1-bit) ?
  - 0-bit: Compute $k := i - rank_1(i) + 1$, go to left child.
  - 1-bit: Compute $k := rank_1(i)$, go to right child.
- With the child as new root, query for $rank_\sigma(k)$.
- Result is found when no child exists for σ.

Wavelet Tree: Example Query $rank_{\mathtt{n}}(15)$



Saving Space for the suffix array `pos`

Storing the Suffix Array pos

Problem

So far, we can answer the pattern search **decision** problem and the **counting** problem.
How do we obtain the positions of the suffixes in the BWT interval?

Storing the Suffix Array pos

Problem

So far, we can answer the pattern search **decision** problem and the **counting** problem.
How do we obtain the positions of the suffixes in the BWT interval?

Answer

Easy: Enumerate the interval of the suffix array pos. However...

Storing the Suffix Array $\mathtt{pos}$

Problem

So far, we can answer the pattern search **decision** problem and the **counting** problem.
How do we obtain the positions of the suffixes in the BWT interval?

Answer

Easy: Enumerate the interval of the suffix array $\mathtt{pos}$. However...
Storing the complete suffix array $\mathtt{pos}$ takes space (less than suffix tree, but still...).
We are looking for a more space-efficient solution.

Sparse suffix array?

For the Occ table, we store only every k-th entry and recompute the rest on demand.
Can we do the same for the suffix array?

Successor Array Ψ and Predecessor Array Ψ^{-1}

Definition: successor array Ψ

$\Psi[r]$ is the index in the suffix array pos, such that

$$\mathrm{pos}[\Psi[r]] = \mathrm{pos}[r] + 1$$

Successor Array Ψ and Predecessor Array Ψ^{-1}

Definition: **successor array** Ψ

$\Psi[r]$ is the index in the suffix array `pos`, such that

$$\texttt{pos}[\Psi[r]] = \texttt{pos}[r] + 1$$

We used this in principle for Kasai's linear-time `lcp` computation:

$$\Psi[r] = \texttt{rank}[\texttt{pos}[r] + 1]$$

Successor Array Ψ and Predecessor Array Ψ^{-1}

Definition: successor array Ψ

$\Psi[r]$ is the index in the suffix array pos, such that

$$\texttt{pos}[\Psi[r]] = \texttt{pos}[r] + 1$$

We used this in principle for Kasai's linear-time `lcp` computation:

$$\Psi[r] = \texttt{rank}[\texttt{pos}[r] + 1]$$

We call Ψ^{-1}, the inverse of Ψ, the **predecessor array**. This is the *LF* mapping.

$$\texttt{pos}[\Psi^{-1}[r]] = \texttt{pos}[r] - 1$$

$$\Psi^{-1}[r] = \texttt{rank}[\texttt{pos}[r] - 1]$$

Example: The Predecessor Array Ψ^{-1} or LF

r	LF $\Psi^{-1}[r]$	$\texttt{pos}[r]$	L $\texttt{bwt}[r]$	F $T[\texttt{pos}[r]:]$
0	1	**13**	`i`	`$`
1	2		`i`	`i$`
2	8		`p`	`ii$`
3	7	1	`m`	`iissisissippii$`
4	10		`s`	`ippii$`
5	11		`s`	`issippii$`
6	3	2	`i`	`ississippii$`
7	0		`$`	`miississippii$`
8	9		`p`	`pii$`
9	4	9	`i`	`ppii$`
10	12		`s`	`sippii$`
11	13		`s`	`sissippii$`
12	5	6	`i`	`ssippii$`
13	6		`i`	`ssissippii$`

$LF[r] = \texttt{C}[a] + \texttt{Occ}(a, r) - 1$,
where $a = \texttt{bwt}[r]$:

- Reconstruct T from `bwt`
- Reconstruct `pos`

Using LF to (Partially) Reconstruct the Suffix Array

Approach

$$\begin{aligned} & \mathrm{pos}[\Psi^{-1}[r]] = \mathrm{pos}[r] - 1 \\ \Leftrightarrow \quad & \mathrm{pos}[r] = \mathrm{pos}[\Psi^{-1}[r]] + 1 \end{aligned}$$

Applying that relationship **recursively** yields

$$\mathrm{pos}[r] = \mathrm{pos}[(\Psi^{-1})^k[r]] + k$$

Data structure

- We compute $\Psi^{-1} = LF$ from `Occ` and `C`
- Store every t-th entry of suffix array pos (e.g. $t = 32$: BWA read mapper)
- Reconstruct the rest of the suffix array (pos) on-the-fly:
  Apply Ψ^{-1} and increase k until we hit a stored value

Summary

FM Index

- BWT, `C`, `Occ`
- implementation: succinct rank data structure on wavelet tree
- sampled `pos`, every t-th entry
- original text is not required!

Backward Search

- Compute interval for $c\mu$ from interval for μ
- Constant time per character, $O(m)$ for pattern P with $|P| = m$
- Enumeration of text positions from sampled `pos`,
  expected $O(t)$ time, but worst-case $O(n)$ time per position

Possible Exam Questions

- Why and how is the FM index compressed?
- How can rank (`Occ`) queries be implemented in constant time with succinct space?
- What is a wavelet tree? How does it support character and rank queries?
- What are the successor / predecessor arrays? Construct an example.
- Explain sparse suffix arrays.
- How long does a query on a sparse suffix array take in the worst case?
- How can one determine the position of pattern matches for a BWT interval?