Other notable rank/select data structures for sequences include:

- The multiary wavelet tree replace the binary tree with a higher degree tree and supports operations in $\mathcal{O}(1 + (\log\sigma)/\log\log n)$ time.
- The GMR data structure needs $n(\log\sigma + o(\log\sigma))$ bits and supports select in $\mathcal{O}(1)$ time, and rank and access $\mathcal{O}(\log\log\sigma)$ time.
- Combining multiary wavelet trees with GMR using a technique called alphabet partitioning need $nH_0(S) + o(n(H_0(S) + 1))$ bits and supports select in $\mathcal{O}(1)$ time, and rank and access $\mathcal{O}(\log\log\sigma)$ time.

The last two have been implemented and can be better than wavelet trees for really large alphabets.

Wavelet trees support other queries too, including two-dimensional orthogonal range queries:

$$\text{range\_count}_S([b..e),[x..y)) = |\{i \in [b..e) \mid S[i] \in [x..y)\}|$$

Compressed Text Indexes

With a text as the data, we are interested in operations on substrings rather than individual symbols. In particular, text indexes support pattern matching queries.

Let $T[0..n)$ be a text over an alphabet $\Sigma = [0..\sigma)$. Define the following operations:

$$\begin{aligned}
\mathsf{extract}_T(i,j) &= T[i..j) \quad \text{for } 0 \le i \le j \le n \\
\mathsf{count}_T(P) &= |\{i \in [0..n-m) \mid T[i..i+m) = P\}| \quad \text{for } P \in \Sigma^m,\ m \in [1..n) \\
\mathsf{locate}_T(P) &= \{i \in [0..n-m) \mid T[i..i+m) = P\} \quad \text{for } P \in \Sigma^m,\ m \in [1..n)
\end{aligned}$$

Example 3.10: $T =$ `banana$` and $P =$ `ana`:

$$\begin{aligned}
\mathsf{count}_T(P) &= 2 \\
\mathsf{locate}_T(P) &= \{1, 3\}
\end{aligned}$$

FM-Index

FM-index is a text index based on the Burrows–Wheeler transform. Recall:

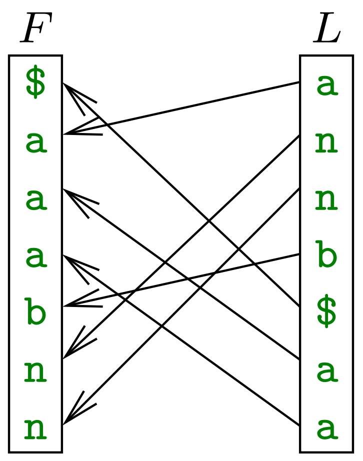
- The BW-matrix $\mathcal{M}$ contains the rotations of the text in lexicographical order. The last column L is the transform. The first column is called F.
- The suffix array SA, maps the rows into text positions.
- A permutation called LF-mapping can be used for recovering the text in reverse order:

$$T[n - i + 1] = L[\mathsf{LF}^i(r_0)]$$

where r_0 is the row containing T.

Example 3.11: $T =$ `banana$`, $r_0 = 4$.

SA	F						L
6	$	b	a	n	a	n	a
5	a	$	b	a	n	a	n
3	a	n	a	$	b	a	n
1	a	n	a	n	a	$	b
0	b	a	n	a	n	a	$
4	n	a	$	b	a	n	a
2	n	a	n	a	$	b	a



Let us first see, how to implement LF-mapping in compressed form.

- LF-mapping maps the jth occurrence of a symbol a in L into the jth occurrence of a in F. Thus

$$\mathsf{LF}(i) = \mathsf{select}_F(a, \mathsf{rank}_L(a, i)) \quad \text{where } a = \mathsf{access}_L(i)$$

- LF-mapping is also the permutation that stably sorts L. Thus

$$\mathsf{LF}(i) = \mathsf{sum}_C(a) + \mathsf{rank}_L(a, i)) \quad \text{where } a = \mathsf{access}_L(i)$$

  where $C[0..\sigma)$ contains the symbol counts, i.e, $C[a]$ is the number of occurrences of a in L (or T or F).

- Since σ is usually fairly small, we can implement sum_C as an array $\mathsf{sum}_C[0..\sigma)$ using $\mathcal{O}(\sigma \log n)$ bits.

- rank_L can be implemented as a wavelet tree. Since L is a permutation of T, $H_0(L) = H_0(T)$. Thus the space is close to the zeroth order entropy of the text.

- The time complexity of computing $\mathsf{LF}(i)$ is $\mathcal{O}(\log \sigma)$.

We can reduce the space for rank_L using compression boosting. Recall that

$$\sum_{w \in \Sigma^k} |L_w| H_0(L_w) = |T| H_k(T)$$

where L_w are the k-context blocks of L.

Three ways of achieving compression boosting have been described:

- Optimal compression boosting uses an algorithm to compute an optimal partitioning of L as described earlier. There is a separate wavelet tree for each block.
- Fixed block compression boosting is the same except the blocks are not chosen according to context but have all the same size, which simplifies implementation.
- When the RRR method is used for implementing the bit vectors, compression boosting is achieved because of the way RRR divides the bit vectors into blocks and compresses them. This is called implicit compression boosting.

All three methods are coarsely optimal, i.e., they use $nH_k(T) + o(n \log \sigma)$ bits for any $k = o(\log_\sigma n)$.

Let us now describe how the operation extract is implemented.

- To recover $T[i..j)$, we use the LF-mapping as when recovering the whole text, but now we start at the row $T[j..n)T[0..j)$ and recover only $j - i$ characters.
- To find the row, we could use the inverse suffix array SA^{-1}, the inverse permutation of SA, that maps text positions to matrix rows.

Example 3.12:

	0	1	2	3	4	5	6
T	b	a	n	a	n	a	$
SA^{-1}	4	3	6	2	5	1	0

- However, SA^{-1} is too large and usually incompressible, so we store only every hth entry of SA^{-1}. This is called sampling. We need $(n/h)\log n$ bits of space.
- To recover $T[i..j)$, we will in fact recover $T[i..k)$, where $k = h\lceil j/h \rceil$ is the nearest sample position to the right of j. The time complexity is $\mathcal{O}((j - i + h)\log\sigma)$.

Now let us consider the pattern matching operations.

- Each occurrence of a pattern P is a prefix of some rotation, and these rotations form a consecutive range in the matrix.

- The pattern matching operations are based on identifying that range.

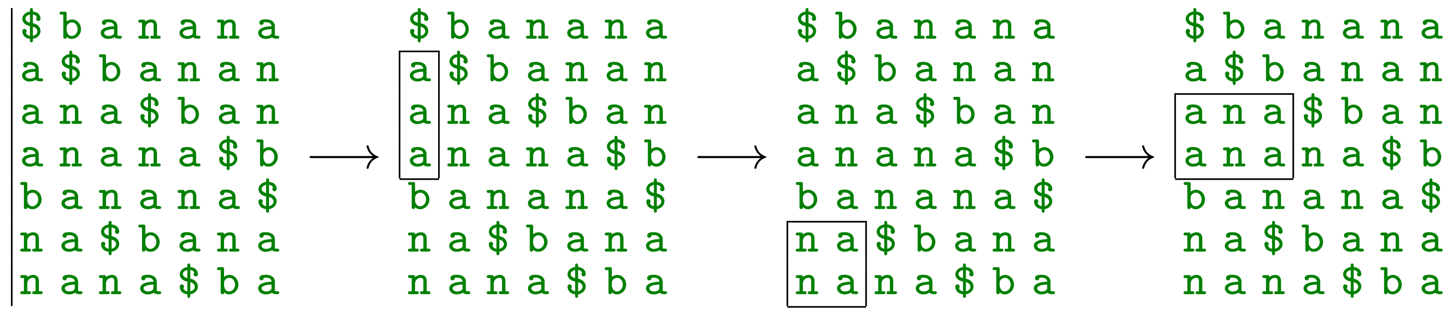
Example 3.13: The BW-matrix of $T =$ `banana$` and the occurrences of patterns `ana` and `na`.

$$\begin{array}{ccccccc}
F & & & & & & L \\
\$ & b & a & n & a & n & a \\
a & \$ & b & a & n & a & n \\
a & n & a & \$ & b & a & n \\
a & n & a & n & a & \$ & b \\
b & a & n & a & n & a & \$ \\
n & a & \$ & b & a & n & a \\
n & a & n & a & \$ & b & a
\end{array}$$

The range of rows for a pattern P can be found using a procedure called backward search.

- For $i \in [0..m)$, let $[b_i..e_i)$ be the range for the pattern suffix $P[m-i..m)$ of length i.
- Backward search starts with $[b_0..e_0) = [0..n)$ and then computes each $[b_{i+1}..e_{i+1})$ from $[b_i..e_i)$.

Example 3.14: Backward search of a pattern $P =$ `ana` in a text $T =$ `banana$`.



- Each row $j \in [b_i..e_i)$ corresponds to an occurrence of $P[m-i..m)$. The symbol preceding that occurrence is $L[j]$. Thus if $L[j] = P[m-i-1]$, we have an occurrence of $P[m-i-1..m)$. The corresponding row is $LF(j)$.

- To compute b_{i+1}, we have to find the first row $j \in [b_i..e_i)$ such that $L[j] = a = P[m-i-1]$. That row is $j = \mathsf{select}_L(a, \mathsf{rank}_L(a, b_i))$. Thus

  $$\begin{aligned} b_{i+1} &= \mathsf{LF}(\mathsf{select}_L(a, \mathsf{rank}_L(a, b_i))) \\ &= \mathsf{sum}_C(a) + \mathsf{rank}_L(a, \mathsf{select}_L(a, \mathsf{rank}_L(a, b_i))) \\ &= \mathsf{sum}_C(a) + \mathsf{rank}_L(a, b_i) \end{aligned}$$

  where we used the fact that $\mathsf{rank}_L(a, \mathsf{select}_L(a, k)) = k$.

- Similarly, $e_{i+1} = \mathsf{sum}_C(a) + \mathsf{rank}_L(a, e_i)$.

Here is the final procedure:

```
BackwardSearch($P[0..m-1]$)
(1) $b \leftarrow 0$; $e \leftarrow n$
(2) for $i \leftarrow m-1$ downto 0 do
(3)     $a \leftarrow P[i]$
(4)     $b \leftarrow \mathsf{sum}_C(a) + \mathsf{rank}_L(a, b)$
(5)     $e \leftarrow \mathsf{sum}_C(a) + \mathsf{rank}_L(a, e)$
(6) return $[b..e)$
```

To implement count, we can simply return the range size $e - b$. Locate is more complicated.

Backward search returns a range of rows. To implement locate, we have to turn the row numbers into text positions.

- The suffix array maps rows into text positions, but it is too large, so we use sampling.

- To find the text position of a row r, we follow LF-mapping until we reach a row that belongs to the sample. If we obtain the text position i after ℓ steps of LF-mapping, the text position for row r is $i + \ell$.

- To ensure that less than h steps of LF-mapping is required, we need to include every hth text position in the sample. Then we need $\mathcal{O}(h \log \sigma)$ time for each pattern occurrence.

- The set of rows in the sample is irregular, so we implement the suffix array as a sparse array with the help of a bit vector as described earlier.

- We need $\mathcal{O}((n/h) \log n)$ bits for the suffix array values. In uncompressed form the bit vector needs $n + o(n)$ bits. A compressed bit vector can be shown to fit in $\mathcal{O}((n/h) \log h) + o(n)$ bits.

Theorem 3.15: A text $T[0..n)$ over an alphabet $[0..\sigma)$, can be stored in

$$nH_k(T) + o(n \log \sigma)$$

bits, for any $k = o(\log_\sigma n)$, so that

- $\mathrm{count}_T(P)$ can be computed in $\mathcal{O}(|P| \log \sigma)$ time.

With an additional $\mathcal{O}((n/h) \log n)$ bits, for any $h \geq 1$,

- $\mathrm{extract}_T(i..j)$ can be computed in $\mathcal{O}((j - i + h) \log \sigma)$ time
- $\mathrm{locate}_T(P)$ can be computed in $\mathcal{O}(|P| + hq) \log \sigma)$ time, where q is the number of occurrences of P in T.

Setting $h = \omega(\log_\sigma n)$ makes this a coarsely optimal compression method.

The $\log \sigma$ factor in the times can be reduced using multiary wavelet trees or alphabet partitioning instead of standard wavelet trees.

Other notable types of compressed text indexes include:

- Compressed suffix arrays (CSAs) are based on a compressed representation of LF^{-1} (often called Ψ), the inverse permutation of LF. CSAs are quite similar to FM indexes, but the search procedures based on LF^{-1} differ from those based on LF.

- Lempel-Ziv (LZ) indexes are based on Lempel-Ziv compression of the text. The data structures and algorithms involved are quite different from FM indexes, as are their properties. LZ indexes are often slow on counting but fast on locating and extracting.

- Compressed suffix trees are often based on CSAs but add more data structures to simulate suffix trees, which are powerful text indexes supporting many operations besides pattern matching (see the course String Processing Algorithms, for example).

There are many variants on these text indexes, both theoretical and practical. The area is still under active research.