Full-Text Indexing

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Preprocess one text for many queries

An exact matcher preprocesses a pattern and scans a text. A full-text index reverses the investment: preprocess one large text once, then answer many future pattern queries without rescanning it. The goal

is query time governed mainly by pattern length and output size.

All structures in this note organise the same objects, the text suffixes, in different representations. The running example `banana$` connects them from first principles.

2 Learning goals

After studying this note, you should be able to:

- explain why all substrings are prefixes of suffixes;
- construct small suffix tries, suffix trees, and suffix arrays;
- distinguish simple and efficient suffix-array construction;
- find a pattern interval using two binary searches;
- construct and use the longest-common-prefix (LCP) array;
- compute and invert the Burrows–Wheeler transform (BWT);
- trace backward search with half-open intervals; and
- distinguish FM-index counting from locating occurrences.

3 Prerequisites

You should know tries, binary search, lexicographic order, and asymptotic notation. The succinct-data-structures note introduces bit-vector rank/select; the required rank convention is restated here. The exact-matching note explains why preprocessing choices depend on whether the pattern or text is reused.

The order of the note is a representation ladder. A suffix trie stores every character explicitly; path compression gives a suffix tree; a suffix array stores only lexicographic order and LCP restores shared-prefix information; BWT plus rank turns the same order into backward search; sampling then trades FM-index space against locating time.

4 Suffixes and a unique sentinel

Append a sentinel `$` that occurs nowhere else and is ordered before every text symbol. Let the resulting text $T[0..N)$ include the sentinel. There are N suffixes $T[i..N)$, one from every position i.

For `banana$` they are:

position	suffix
0	`banana$`
1	`anana$`
2	`nana$`
3	`ana$`
4	`na$`
5	`a$`
6	`$`

Every substring $T[i..j)$ is a prefix of suffix $T[i..N)$. An index that can navigate suffix prefixes can therefore answer substring queries. The sentinel makes every suffix end at a distinct leaf and gives cyclic-rotation constructions one unambiguous original row.

5 Suffix tries and compressed suffix trees

A **suffix trie** inserts all N suffixes into one character-labelled trie. To search pattern P, follow its characters from the root. If the path exists, every descendant leaf stores one occurrence position. Following the path costs $O(|P|)$ with constant-time outgoing-edge lookup. Reporting in another $O(occ)$ time requires a stored leaf interval or occurrence list; a raw depth-first walk through an uncompressed trie can also traverse long unary paths.

The trie can contain $\Theta(N^2)$ nodes because the total suffix length is quadratic. A **suffix tree** compresses each maximal nonbranching path into one edge. Store an edge label as the interval `[start,end)` of the original text rather than copying its characters.

Search compares pattern characters along edge intervals. If all of P is consumed, descendant leaves give its positions. The tree has N leaves. Every nonroot internal node has at least two children, so there are fewer than N internal nodes; edge intervals and nodes therefore use $O(N)$ words with a sparse or fixed-alphabet child representation.

A simple teaching construction inserts every suffix and may take $O(N^2)$ time. Online linear-time constructions maintain suffix links and implicit positions so work is shared between suffixes; Ukkonen's algorithm is the best-known example. Linear time assumes an alphabet representation supporting the required transition operations. The distinction between **linear-size representation** and **linear-time construction** matters.

Suffix trees support more than lookup. An internal node's string depth is the length of a repeated substring, and its descendant-leaf count is the occurrence count. A generalised suffix tree marks leaves from several texts; the deepest node with leaves from two sources yields a longest common substring.

6 Suffix arrays

The suffix array `SA[0..N)` stores starting positions of suffixes in lexicographic order. For `banana$`:

rank r	`SA[r]`	suffix
0	6	`$`
1	5	`a$`
2	3	`ana$`
3	1	`anana$`
4	0	`banana$`
5	4	`na$`
6	2	`nana$`

Thus `SA = [6,5,3,1,0,4,2]`. Array values are text positions; their array indices are suffix **ranks**.

6.1 Construction choices

Materialising and comparison-sorting all suffix strings is simple but uses $\Theta(N^2)$ characters and can spend $O(N^2 \log N)$ time in worst-case comparisons. Comparing suffixes by references saves copied space but not repeated character work.

Prefix doubling gives a practical improvement. Initially rank suffixes by their first character. At round k, when ranks represent the first 2^k characters, sort each suffix position i by

$$(rank[i], rank[i + 2^k]),$$

using a special rank for positions beyond the sentinel. Rename equal pairs with the same new rank and continue until all ranks differ.

```
suffix_array_doubling(T):
    rank[i] = code(T[i])
    length = 1
    while length < N:
        sort positions i by (rank[i], rank[i+length])
        assign new consecutive ranks to unequal pairs
        length = 2*length
    return positions ordered by final rank
```

Comparison sorting gives $O(N \log^2 N)$ time. Counting/radix sorting integer ranks gives $O(N \log N)$ time and $O(N)$ working space. Direct linear-time suffix-array algorithms exist, but their engineering cost is higher; an implementation report should name the construction actually measured.

6.2 Pattern search with two boundaries

All suffixes beginning with P form one contiguous lexicographic interval. Binary-search the first suffix not less than P, then the first suffix greater than every string with prefix P. The half-open range `[lo,hi)` contains exactly the occurrences.

A straightforward comparison examines up to $|P|$ characters at each of $O(\log N)$ search steps, so query time is $O(|P| \log N + occ)$. Retaining longest-common-prefix information during the searches can avoid repeated comparisons and approach $O(|P| + \log N + occ)$.

7 The LCP array

Define `LCP[0]=0` and, for $r > 0$,

$$LCP[r] = lcp(T[SA[r-1]..N), T[SA[r]..N)).$$

For `banana$`, `LCP = [0,0,1,3,0,0,2]`. The value 3 between `ana$` and `anana$` exposes repeat `ana`; value 2 between `na$` and `nana$` exposes `na`.

Given `SA`, its inverse rank array, and the fact that deleting the first character reduces a known LCP by at most one, Kasai's algorithm constructs `LCP` in $O(N)$ time. Range-minimum queries over LCP give the common-prefix length of any two suffixes. The suffix array plus LCP and suitable navigation information can simulate many suffix-tree operations with better memory locality.

8 Burrows–Wheeler transform

For each suffix-array row, output the character preceding that suffix, wrapping position zero to the sentinel:

$$L[r] = T[(SA[r] - 1) \bmod N].$$

For `banana$`, the rows give `L = annb$aa`. This is the BWT. Equivalent definitions sort all cyclic rotations and take their last column.

The BWT is a reversible permutation of the input, not a compressor by itself. Lexicographically adjacent suffixes have similar following context, so their preceding characters often form runs. Move-to-front, run-length, and entropy coding can exploit those runs; this is why BWT-based compressors can work well.

9 LF mapping and BWT inversion

Let `F` be the first column of sorted rotations, which is simply the sorted text. Stable sorting preserves occurrence order: the rth `c` in `L` is the same text occurrence as the rth `c` in `F`.

Use the exclusive convention

$$Occ(c, i) = |\{j : 0 \le j < i,\ L[j] = c\}|,$$

and let $C[c]$ be the number of text characters strictly smaller than c. Then

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

`LF(i)` moves from the row for suffix starting at s to the row for the suffix starting at $(s-1) \bmod N$. In terms of the inclusive rank convention from the succinct-data-structures note, $Occ(c, 0) = 0$ and $Occ(c, i) = rank_c(i-1)$ for $i > 0$.

The unique sentinel identifies a starting row for inversion:

```
invert_bwt(L):
    build C and Occ/rank support
    row = position of '$' in L
    for k = N-1 down to 0:
        T[k] = L[row]
        row = LF(row)
    return T
```

For `annb$aa`, the visited output characters in reverse positions are `$`, `a`, `n`, `a`, `n`, `a`, `b`, reconstructing `banana$`. With prefix-count tables this is $O(N)$ time; storing a dense `Occ` table costs $O(N|\Sigma|)$, while rank data structures reduce space.

10 Backward search: extend a suffix interval

Suppose suffix-array interval `[l,r)` contains exactly the suffixes beginning with already-processed pattern suffix Q. Prepending character c keeps precisely rows whose preceding BWT character is c. Stable occurrence ranks map them into the `F` block for c:

$$l' = C[c] + Occ(c, l), \qquad r' = C[c] + Occ(c, r).$$

Process the pattern from right to left. Start with every row `[0,N)`.

11 Worked trace: backward search for `ana`

For `ana` in `banana$`:

next character	old interval	new interval
`a`	`[0,7)`	`[1,4)`
`n`	`[1,4)`	`[5,7)`
`a`	`[5,7)`	`[2,4)`

The final suffix-array slice is `SA[2..4) = [3,1]`, exactly the two starts of `ana`. If an interval becomes empty, no occurrence exists.

Correctness follows from the LF principle: rows in the old interval share prefix Q, and selecting their preceding `c` occurrences gives exactly suffixes with prefix cQ. Their stable ranks make the update endpoints contiguous.

12 FM-index: count first, locate when needed

An FM-index stores the BWT with a data structure answering `Occ`/rank queries, plus optional samples. Backward search performs two rank queries per pattern character, so counting takes

$$O(|P|\, t_{rank}).$$

For a small alphabet, per-character bit vectors can provide constant-time rank with lower-order redundancy. A wavelet tree or matrix supports larger alphabets, commonly with logarithmic alphabet dependence.

The final interval `[l,r)` gives count $r - l$ immediately but not text positions. To **locate**, start from each row in the interval. During index construction, sample rows whose suffix-array value is divisible by period d.

Repeated LF steps decrement the represented suffix position modulo N until a sample is reached; at most $d - 1$ steps are required under this sampling rule. If k steps reach sampled value p, the original occurrence starts at $(p + k) \bmod N$.

Samples use about N/d integers, while each reported occurrence costs $O(d\,t_{rank})$ in the worst case. This is an explicit space–locate-time trade-off.

Compressed-index space claims must include the rank structure and samples. "Near entropy" refers to a particular representation and model; BWT alone is still N symbols.

13 Construction and query trade-offs

Structure	Stored space	Basic construction	Exact query
Suffix trie	$\Theta(N^2)$ worst	$O(N^2)$ simple	$O(\|P\| + occ)$ with a leaf interval
Suffix tree	$O(N)$ words	$O(N^2)$ simple; $O(N)$ advanced	$O(\|P\| + occ)$
Suffix array	$O(N)$ words	$O(N \log N)$ prefix doubling with radix sort	$O(\|P\| \log N + occ)$ basic
FM-index	compressed BWT, rank, samples	construction dependent	$O(\|P\| t_{rank})$ count plus locate

14 Connections

- The suffix tree is a path-compressed trie, connecting back to the tree note.
- Prefix doubling reuses sorting and rank compression.
- BWT backward search turns text search into rank queries from the succinct-data-structures note.
- Exact matchers preprocess each pattern; suffix and FM indexes preprocess the shared text.
- LCP links lexicographic adjacency to repeated-substring and longest-common-substring problems.

15 Common mistakes

- Omitting or duplicating the sentinel.
- Copying suffix-tree edge strings and silently losing linear space.
- Confusing `SA[r]` with the rank of suffix r.
- Calling a naive suffix sort linear because the output array has linear size.
- Using one binary search instead of finding both pattern-range boundaries.
- Mixing inclusive rank with exclusive `Occ`.
- Treating BWT as compression without a subsequent coder.
- Claiming FM-index locating has the same cost as counting.

16 Self-check

1. List and sort all suffixes of `miss$`.
2. Why does a sentinel make the suffix-tree leaf-count argument simple?
3. Perform one prefix-doubling rank round for `banana$`.
4. Verify every value of the `banana$` LCP array.
5. Compute `LF(0)` and `LF(4)` for `annb$aa`.
6. Reproduce the three backward-search intervals for `ana`.
7. What changes when the suffix-array sampling period doubles?

17 Revision summary

- Every substring is a prefix of a suffix, so organising suffixes indexes all factors.
- Path compression makes a suffix trie linear-size; construction time is a separate question.
- Suffix arrays store suffix order compactly, and LCP recovers shared-prefix structure.
- BWT records preceding characters in suffix order and is reversible through LF mapping.
- Backward search repeatedly maps a half-open suffix interval with exclusive rank counts.
- An FM-index counts compactly; locating positions requires suffix-array samples and extra LF steps.

18 Implementations and hands-on exploration

- **Python construction:** `pydivsufsort` provides suffix-array search, BWT and inversion, Kasai LCP, and NumPy outputs. It accepts Python strings, bytes, and integer arrays; choose the sequence element that an index position should count instead of assuming byte and Unicode-character offsets coincide. Its example stores adjacent LCP values at the left endpoint rather than using this note's `LCP[r]` previous-suffix convention, and its BWT API returns a primary index instead of requiring the explicit sentinel representation used here.
- **C construction:** libsais constructs suffix arrays, LCP/PLCP arrays, BWT, and inverse BWT, with optional OpenMP. The basic byte API uses 32-bit indices for inputs below 2 GiB; use the separate 64-bit interface for larger inputs and benchmark memory-bandwidth scaling.
- **C++ succinct structures:** SDSL v3 supplies bit vectors with rank/select, wavelet structures, compressed suffix arrays, FM-indexes, and compressed suffix trees. It is a template library: name the concrete support and index types, because representation and serialised data depend on them.
- **C++ indexed search:** SeqAn3 FM-index search builds uni- and bidirectional FM-indexes and supports exact and bounded-error queries over single texts or collections. Bidirectionality costs additional space, and the selected hit policy determines whether all, single-best, all-best, or strata results are returned. Its high-level search reports reference positions because the index includes locating support; a bare backward-search interval would provide only a count.
- **Rust indexing:** `bio::data_structures::fmindex` exposes byte-oriented BWT, `less`, sampled `Occ`, backward search, and FMD-index components. Its `Occ::occ(r,a)` counts through index r inclusively: this note's $Occ(a, 0)$ is zero, and for $i > 0$ its $Occ(a, i)$ is the library call `occ(i-1,a)`. Backward search returns a suffix-array interval; the documented `SAInterval::occ` needs a suffix array to report text positions, or a separate sampled-SA locating layer must be supplied.

18.1 Small experiments

1. Use `pydivsufsort` on `banana$` to reproduce the suffix array, convert its LCP layout to this note's convention, and invert its BWT. For each test pattern, compare suffix-array search positions with a direct scan.
2. Build rank support for the same BWT with sampling periods 1, 3, and 8 using Rust Bio or an equivalent implementation. Verify identical backward-search intervals, then record stored counters and rank-query time to expose the space-time trade-off.

19 Sources and further study

- Full-text indexing lecture deck, Jaak Vilo.
- Dan Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
- Paolo Ferragina and Giovanni Manzini, "Opportunistic data structures with applications," *FOCS*, 2000.