CS 7800/4810: Advanced Algorithms Spring 2026

Lecture 7 — Feb 02, 2026

Prof. Prashant Pandey *Scribe: Sudhanva Devanathan*

1 Overview

In the previous lecture, we explored how static data structures, specifically Suffix Trees, facilitate efficient string matching. To recap, the core objective is:

> **Goal:** Given a text $\underline{T}$ and a pattern $\underline{P}$ (both strings over a fixed alphabet), locate some or all occurrences of P within T.

In this session, we will shift our focus to alternative structures for solving this problem: **Suffix Arrays** and the **Burrows-Wheeler Transform (BWT)**.

2 Suffix Arrays

2.1 Motivation: The Human Genome Problem

Consider the scale of the Human Genome as a string T. It possesses the following properties:

- **Alphabet Size:** $|\Sigma| = 4$
- **String Length:** $|T| \approx 3 \times 10^9$ base pairs.
- **Space:** Each base pair requires $\log(4) = 2$ bits. Total raw size $\approx 6 \times 10^9$ bits.

> **Goal:** Efficient substring matching over the Human Genome.

Memory footprint comparison for various data structures on $|T| = 3 \times 10^9$:

- **Suffix Tree:** ≈ 47 GB (High overhead due to pointers/nodes).
- **Suffix Array:** ≈ 12 GB ($\approx 4\times$ more space-efficient).
- **FM-Index:** < 1.5 GB.

2.2 Suffix Array Construction

Goal: Reduce the space overhead of Suffix Trees while maintaining search time complexities.

A Suffix Array (SA) is a sorted permutation of all suffixes of T. Instead of storing the full strings, we store only the starting indices of the suffixes in lexicographical order.

2.2.1 Example: $T =$ banana$

First, we list all suffixes with their starting positions:

idx	**Suffix**
0	`banana$`
1	`anana$`
2	`nana$`
3	`ana$`
4	`na$`
5	`a$`
6	`$`

After sorting these suffixes lexicographically ($\$ < a < b < \dots$), we obtain the Suffix Array:

	0	1	2	3	4	5	6
$SA \rightarrow$	6	5	3	1	0	4	2
	$	a$	ana$	anana$	banana$	na$	nana$

2.2.2 Complexity

- **Search Time:** Using binary search, we can find a pattern P in $\mathcal{O}(P \log |T|)$ (can be improved to $\mathcal{O}(P + \log |T|)$ using LCP array and RMQ [Range Minimum Query]).
- **Construction Time:** $\mathcal{O}(T + Sort(|\Sigma|)$ (Advanced algorithms can construct the array in $\mathcal{O}(T)$).
- **Total Space Complexity:**

$$\underbrace{6 \times 10^9}_{\text{Genome String}} + \underbrace{3 \times 10^9 \times \lceil \log(3 \times 10^9) \rceil}_{\text{Suffix Array (SA)}} \text{ bits}$$

  - **In Bytes:**

$$\frac{6 \times 10^9 + 3 \times 10^9 \times \lceil \log(3 \times 10^9) \rceil}{8} \text{ bytes}$$

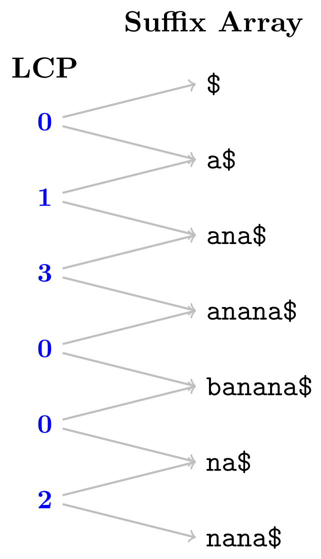
  - **Breakdown:**

* **Genome String (T):** Requires 6×10^9 bits (2 bits per base pair for 3 billion base pairs).
* **Integer Array (SA):** Since the Suffix Array stores indices from 0 to $|T|$, each of the 3 billion entries requires $\lceil \log |T| \rceil$ bits to represent that integer value.

2.3 Longest Common Prefix (LCP)

To optimize the search from $\mathcal{O}(P \log |T|)$ to $\mathcal{O}(P + \log |T|)$, we utilize the **LCP Array**, which stores the length of the longest common prefix between adjacent suffixes in the sorted SA.

SA Index	LCP Value	SA[i]	Suffix
0	—	6	`$`
1	0	5	`a$`
2	1	3	`ana$`
3	3	1	`anana$`
4	0	0	`banana$`
5	0	4	`na$`
6	2	2	`nana$`



3 Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler Transform (BWT) is a reversible permutation of the characters of a string, originally developed for data compression. It is used in compression algorithms like run-length encoding.

3.1 Construction Algorithm

To compute the BWT of a string T (e.g., $T =$ `abaaba$`), we follow these steps:

1. **Cyclic Rotations:** Generate all possible cyclic rotations of string T.
2. **Lexicographical Sort:** Sort these rotations alphabetically.
3. **Extract Columns:** The first column is denoted as F and the last column as L. The column L is the BWT of T.

3.1.1 Step 1: All Cyclic Rotations

First, we list every cyclic shift of the string $T =$ `abaaba$`.

abaaba$
baaba$a
aaba$ab
aba$aba
ba$abaa
a$abaab
$abaaba

3.1.2 Step 2: Sorted Rotations

Next, we sort the rows lexicographically (assuming $\$ < a < b$). The first column is labeled F and the last column is labeled L.

	Rotations	
F	(Sorted Rotations)	**L**
$	abaab	a
a	$abaa	b
a	aba$a	b
a	ba$ab	a
a	baaba	$
b	a$aba	a
b	aaba$	a

- **Column F:** Contains the characters of T sorted lexicographically.
- **Column L:** Contains the BWT of T. This is the string we store.
- **Result:** $BWT(T) =$ `abba$aa`

Note: We do not need to store F explicitly during the BWT process.

3.2 De-Construction (Inverse BWT)

To reconstruct the original string T from the BWT, we utilize a concept known as T-Rank.

3.2.1 T-Rank Definition

For any character c at a specific position in T, its **T-Rank** is defined as the number of occurrences of c that appear *before* that position in T.

Example: Let $T =$ `abaaba$`

Index:	0	1	2	3	4	5	6
$T \rightarrow$	a	b	a	a	b	a	\$
T-Rank $\rightarrow$	0	0	1	2	1	3	0

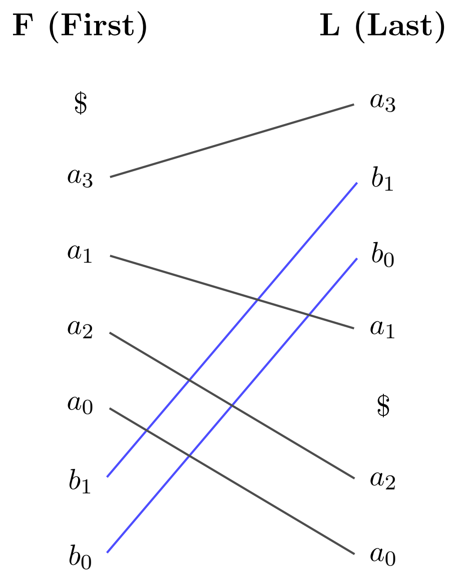
Note: T-Ranks are generally not explicitly stored in memory but are computed on the fly during the inversion process.

3.2.2 BWM with T-Ranking

The core property that allows us to reverse the BWT is the relationship between the First column (F) and the Last column (L).

A bipartite graph where we match characters with the same character and T-Rank in both columns. We can see that the graph is non-intersecting for the same characters.

Note: a's and b's occur in the same order in first and last columns



3.2.3 LF Mapping Property

The Last-to-First (LF) Mapping property state that:

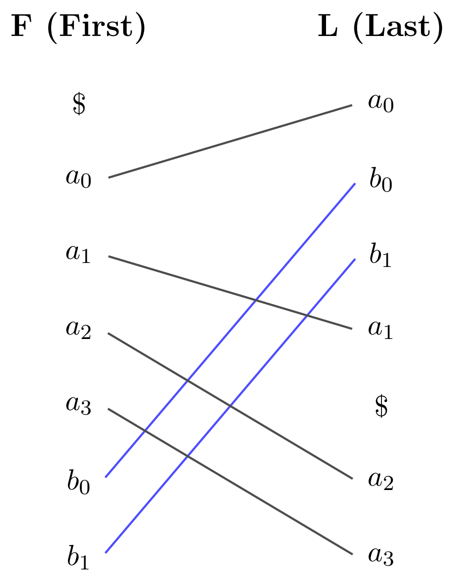
The i-th occurrence of a character c in the Last column (L) corresponds to the exact same character in the original text T as the i-th occurrence of c in the First column (F).

In other words, the relative order of identical characters is preserved between L and F.

- If you look at all the 'a's in column L (from top to bottom), they appear in the same relative order as the 'a's in column F.

3.2.4 B Ranking

To optimize the mapping process, we assign ranks such that for any given character, the ranks appear in strictly ascending order in the First column (F). This simplifies the implementation of the LF-mapping.



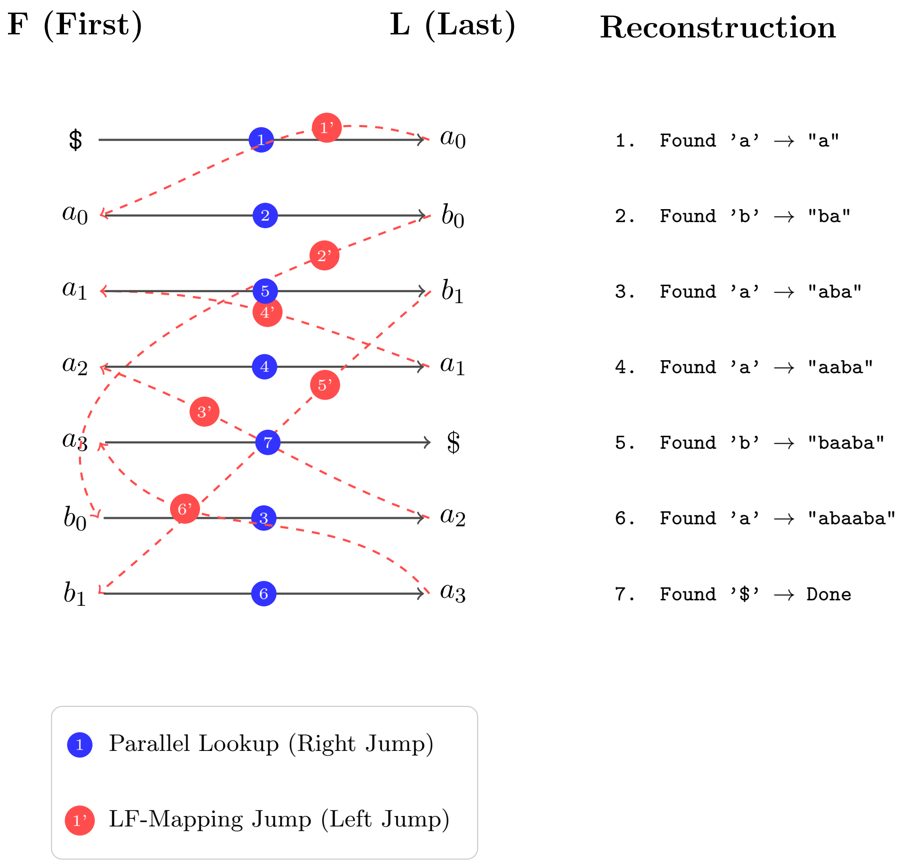
3.2.5 Reconstruction

Reconstructing the original string T from the BWT involves tracing the permutation cycles in reverse order. We utilize the **Last-to-First (LF) Mapping** property to navigate from the end of the string back to the beginning.

Steps:

1. **Initialize:** Start at the row containing \$ in the First column (F).
2. **Identify Predecessor:** Look at the character x in the same row of the Last column (L). This character x immediately precedes the current suffix.
3. **Append:** Add x to the *beginning* of your reconstructed string.

4. **Jump:** Find the row in F where this specific instance of x starts.
5. **Repeat:** Repeat steps 2–4 until the character found in L is \$.



```
1.  Found 'a' → "a"
2.  Found 'b' → "ba"
3.  Found 'a' → "aba"
4.  Found 'a' → "aaba"
5.  Found 'b' → "baaba"
6.  Found 'a' → "abaaba"
7.  Found '$' → Done
```

References

[1] M. Burrows, D.J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. *DEC SRC Research Report 124*, 1994.

[2] U. Manber, G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 1993.