**Prepared by:** [Arun Konagurthu]

# FIT3155: Advanced Algorithms and Data Structures

## Week 3: **Burrows-Wheeler Transform (BWT) and efficient string pattern matching**

Faculty of Information Technology, Monash University

# What is covered in this lecture?

- **Burrows-Wheeler Transform** (BWT) of Strings
- **Inverting a BWT**
- Efficient pattern matching using BWT as an index

# References

## Part I

- Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.
- Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In the proceedings of the 41st Annual Symposium on Foundations of Computer Science. 2000.

## Part II

- Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In the proceedings of the 41st Annual Symposium on Foundations of Computer Science. 2000.

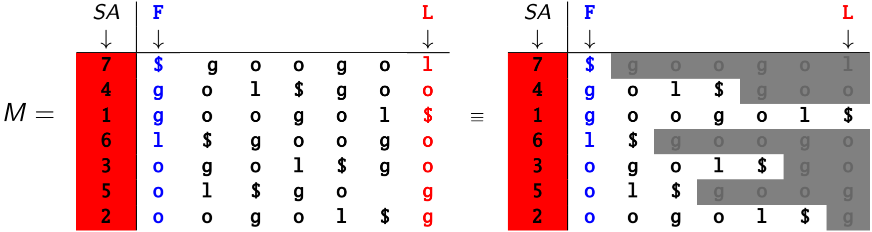
# Revise Suffix array if you have forgotten!

- This lecture will build on your understanding of the **`Suffix Array`** data structure introduced in FIT2004.
- If you have forgotten how to construct a **`suffix array`** of a given string, revise the **`prefix-doubling`** algorithm taught in FIT2004.
- Heads up: After the end of (next) week 4's content, you should be able to construct a suffix array in linear time in the length of a given string.

# PART I: Burrows-Wheeler Transform (BWT)

# Burrows-Wheeler Transform (BWT) of a string

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Reference string/text: | g | o | o | g | o | l | $ |



$M =$ … $\equiv$ …

| SA | F | | | | | | L |
|---|---|---|---|---|---|---|---|
| 7 | $ | g | o | o | g | o | l |
| 4 | g | o | l | $ | g | o | o |
| 1 | g | o | o | g | o | l | $ |
| 6 | l | $ | g | o | o | g | o |
| 3 | o | g | o | l | $ | g | o |
| 5 | o | l | $ | g | o | | g |
| 2 | o | o | g | o | l | $ | g |

## BWT definition

- The string formed by letters in the **last column** (*L*) of the (sorted) cyclic permutation matrix (*M*) is the **`Burrows-Wheeler Transform`** of the text.
- Equivalently, it is also the string formed by the (cyclically) previous letters to the letters in the first column (*F*). (In other words, subtracting one from the the suffix array (indexes) gives you the information of the last column.)

# Matrix $M$ – Property 1

Any column of the (sorted) cyclic permutation matrix $M$ is a **permutation** of **`S[1...n]`**

## Example

**`S[1...n] = g o o g o l $`**

$$M = \begin{array}{ccccccc} \$ & g & o & o & g & o & l \\ g & o & l & \$ & g & o & o \\ g & o & o & g & o & l & \$ \\ l & \$ & g & o & o & g & o \\ o & g & o & l & \$ & g & o \\ o & l & \$ & g & o & o & g \\ o & o & g & o & l & \$ & g \end{array}$$

E.g., **`o l o g o $ g`** is a permutation of the org. string **`g o o g o l $`**

# Matrix *M* – Property 2

Any 2 successive columns of the (sorted) cyclic permutation matrix *M* gives the **`permutation`** of all **2-mers** (substrings of size 2) in `S[1...n]`

## Example

`S[1...n] = g o o g o l $`

$$M = \begin{matrix} \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} \\ \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} \\ \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} \\ \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} \\ \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} \end{matrix}$$

E.g., `oo, l$, og, go, ol, $g, go`, is a permutation of 2-mers of *S*, `go, oo, og, go, ol, l$, $g`

# Matrix $M$ – Property 2 (corollary)

Since $M$ is a matrix of (sorted) cyclic permutations, the last column $L$ precedes the first column $F$.

## Example

**S = g o o g o l $**

$$M = \begin{array}{ccccccc} \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} \\ \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} \\ \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} \\ \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} \\ \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} \\ \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} \\ \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} \\ \hline \uparrow & & & & & & \uparrow \\ \mathtt{F} & & & & & & \mathtt{L} \end{array}$$

# Matrix $M$ – General property

**General property**

Any $k$ successive columns of the (sorted) cyclic permutation matrix $M$ give the **`permutation`** of all **k-mers** in **`S[1...n]`**

# Property: BWT($S$) is **invertible**!!!

## BWT($S$)

- BWT is **invertible**.
- This implies that we can throw away the original reference string $S$, and reconstruct $S$ from BWT($S$).[a]
- We will use the notation $\text{BWT}^{-1}$ to denote the **inverse** of a BWT of a string. By inverse it is implied that $\text{BWT}^{-1}(\text{BWT}(S)) = S$

---

[a] This is magical if you think about this!

# Dumb method to invert BWT(S)

Start with the BWT($S$). Sort it lexicographically.

**Example**

$$\begin{array}{c} \texttt{l} \\ \texttt{o} \\ \texttt{\$} \\ \texttt{o} \\ \texttt{o} \\ \texttt{g} \\ \texttt{g} \end{array} \xrightarrow{sort} \begin{array}{c} \texttt{\$} \\ \texttt{g} \\ \texttt{g} \\ \texttt{l} \\ \texttt{o} \\ \texttt{o} \\ \texttt{o} \end{array}$$

The matrix below is given here only as a reference for you to eyeball the reconstruction.

$$M = \left| \begin{array}{ccccccc} \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} \\ \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} \\ \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} \\ \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} \\ \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} \end{array} \right.$$

This reconstructs the **`first column`** of the matrix $M$.

But **`we know`** that the **`first column`** succeeds (comes after) the **`last (BWT) column`** (**in a cyclic way**)...

# Dumb method to invert BWT($S$)

We just reconstructed the **first column** of $M$. But we also have the **last BWT column** with us. Since the first column succeeds the last, **append the two columns** in their natural (cyclic) order, and sort the letters lexicographically.

**Example**

$$\begin{array}{cc} \texttt{l} & \texttt{\$} \\ \texttt{o} & \texttt{g} \\ \texttt{\$} & \texttt{g} \\ \texttt{o} & \texttt{l} \\ \texttt{o} & \texttt{o} \\ \texttt{g} & \texttt{o} \\ \texttt{g} & \texttt{o} \end{array} \xrightarrow{\textit{sort}} \begin{array}{cc} \texttt{\$} & \texttt{g} \\ \texttt{g} & \texttt{o} \\ \texttt{g} & \texttt{o} \\ \texttt{l} & \texttt{\$} \\ \texttt{o} & \texttt{g} \\ \texttt{o} & \texttt{l} \\ \texttt{o} & \texttt{o} \end{array}$$

The matrix below is given here only as a reference for you to eyeball the reconstruction.

$$M = \left|\begin{array}{ccccccc} \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} \\ \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} \\ \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} \\ \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} \\ \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} \end{array}\right.$$

This reconstructs the **first two columns** of the matrix $M$.
But, again, we know that the **first two columns** succeed the **last (BWT) column** (**in a cyclic way**)...

# Dumb method to invert BWT($S$)

We have now reconstructed the **`first two columns`** of $M$. But, again, we also have the **`last BWT column`**. Since these reconstructed columns succeeds the last column, **append the three columns** in their natural (cyclic) order, and sort lexicographically.

## Example

$$\begin{array}{ccc} \texttt{l} & \texttt{\$} & \texttt{g} \\ \texttt{o} & \texttt{g} & \texttt{o} \\ \texttt{\$} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{l} & \texttt{\$} \\ \texttt{o} & \texttt{o} & \texttt{g} \\ \texttt{g} & \texttt{o} & \texttt{l} \\ \texttt{g} & \texttt{o} & \texttt{o} \end{array} \xrightarrow{sort} \begin{array}{ccc} \texttt{\$} & \texttt{g} & \texttt{o} \\ \texttt{g} & \texttt{o} & \texttt{l} \\ \texttt{g} & \texttt{o} & \texttt{o} \\ \texttt{l} & \texttt{\$} & \texttt{g} \\ \texttt{o} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{l} & \texttt{\$} \\ \texttt{o} & \texttt{o} & \texttt{g} \end{array}$$

The matrix below is given here only as a reference for you to eyeball the reconstruction.

$$M = \left|\begin{array}{ccccccc} \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} \\ \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} \\ \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} \\ \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} \\ \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} \\ \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} \end{array}\right.$$

This reconstructs the **`first three columns`** of the matrix $M$.
But, yet again, we know that the **`first three columns`** succeed the **`last (BWT) column`** (**in a cyclic way**)...

# Dumb method to invert BWT($S$)

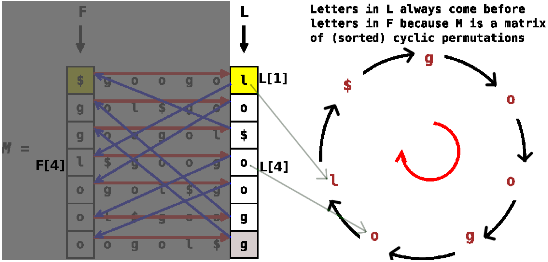
Iteratively appending the BWT column to reconstructed columns before sorting them over $n$ iterations generates the full matrix $M$ of cyclic permutations. The original string $S[1 \ldots n]\$$ is simply the **`first row`** of $M$.

$$M = \begin{array}{ccccccc}
\mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} \\
\mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} \\
\mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} \\
\mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} \\
\mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} \\
\mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} \\
\mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} & \mathtt{g}
\end{array}$$

The naive approach is **`highly inefficient`** in both space and time, so you **`should NOT`** use it in practice. When the reference string is long, this naive approach becomes intractable.

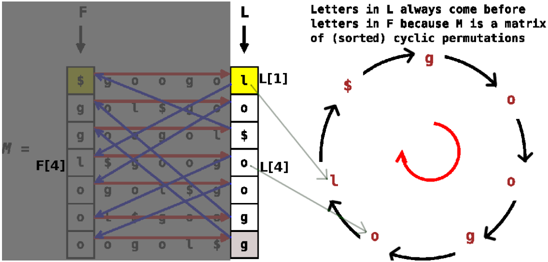
**`So, let's now look at an efficient method to invert a BWT(S)`**.

# LF-mapping: Relationship between the Last Column (L) and First Column (F) of the matrix *M*



- Each letter that appears in the last (or BWT) column L can be mapped to a corresponding letter in the first column (F) – see property 1 on Slide 8

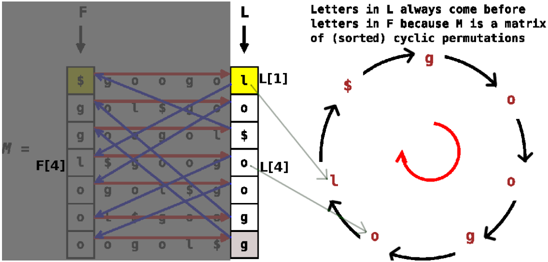
# LF-mapping: Relationship between the Last Column (L) and First Column (F) of the matrix $M$



Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

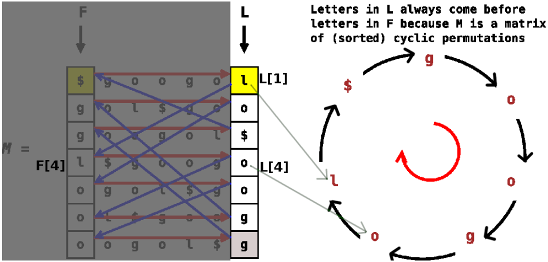
- Each letter that appears in the last (or BWT) column L can be mapped to a corresponding letter in the first column (F) – see property 1 on Slide 8
- L[1] has to be the final letter ($S[n]$) of the original string $S[1 \ldots n]$ (i.e., the letter preceding the artificial terminal symbol \$)– why???

# LF-mapping: Relationship between the Last Column (L) and First Column (F) of the matrix *M*



- Each letter that appears in the last (or BWT) column L can be mapped to a corresponding letter in the first column (F) – see property 1 on Slide 8
- L[1] has to be the final letter ($S[n]$) of the original string $S[1 \dots n]$ (i.e., the letter preceding the artificial terminal symbol \$)– why???
- Without reconstructing the First column (F) (or any other columns for that matter), and **solely with the information in the last/BWT column (L)**, can we compute at which position/index (pos) any $L[i]$ would appear in the first column (F)?

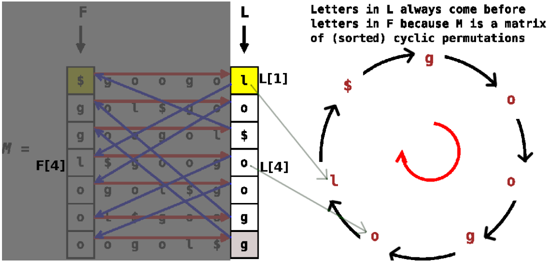
# LF-mapping: Relationship between the Last Column (L) and First Column (F) of the matrix $M$



Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

- Each letter that appears in the last (or BWT) column L can be mapped to a corresponding letter in the first column (F) – see property 1 on Slide 8
- L[1] has to be the final letter ($S[n]$) of the original string $S[1 \ldots n]$ (i.e., the letter preceding the artificial terminal symbol $)– why???
- Without reconstructing the First column (F) (or any other columns for that matter), and **solely with the information in the last/BWT column (L)**, can we compute at which position/index (pos) any $L[i]$ would appear in the first column (F)?
- Punchline: If any $L[i]$ maps to some $F[\text{pos}]$, then $L[\text{pos}]$ has to be its preceding letter due to the property discussed on Slide 10

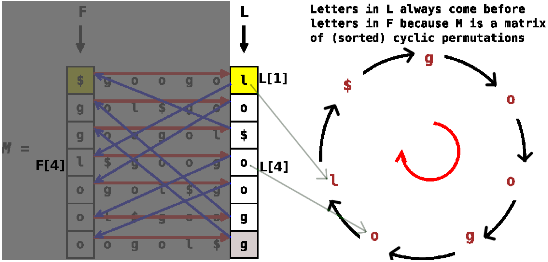
# LF-mapping: Relationship between the Last Column (L) and First Column (F) of the matrix $M$



Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

- Each letter that appears in the last (or BWT) column L can be mapped to a corresponding letter in the first column (F) – see property 1 on Slide 8
- L[1] has to be the final letter ($S[n]$) of the original string $S[1 \dots n]$ (i.e., the letter preceding the artificial terminal symbol \$)– why???
- Without reconstructing the First column (F) (or any other columns for that matter), and **solely with the information in the last/BWT column (L)**, can we compute at which position/index (pos) any $L[i]$ would appear in the first column (F)?
- Punchline: If any $L[i]$ maps to some $F[\text{pos}]$, then $L[\text{pos}]$ has to be its preceding letter due to the property discussed on Slide 10

# LF-mapping: Relationship between the Last Column (L) and First Column (F) of the matrix $M$



Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

- Each letter that appears in the last (or BWT) column L can be mapped to a corresponding letter in the first column (F) – see property 1 on Slide 8
- L[1] has to be the final letter ($S[n]$) of the original string $S[1\ldots n]$ (i.e., the letter preceding the artificial terminal symbol \$)– why???
- Without reconstructing the First column (F) (or any other columns for that matter), and **solely with the information in the last/BWT column (L)**, can we compute at which position/index (pos) any $L[i]$ would appear in the first column (F)?
- Punchline: If any $L[i]$ maps to some $F[\text{pos}]$, then $L[\text{pos}]$ has to be its preceding letter due to the property discussed on Slide 10

In general, this LF-mapping can be used to **recover** the original string $S$ (one letter at a time, in the **backwards direction** starting from the last letter $S[n]$).

# A crucial observation to automate *LF*-mapping (Example)

| | | | | | | |
|---|---|---|---|---|---|---|
| $ | g | o | o | g | o | l |
| g | o | l | $ | g | o | o |
| g | o | o | g | o | l | $ |
| l | $ | g | o | o | g | o |
| o | g | o | l | $ | g | o |
| o | l | $ | g | o | o | g |
| o | o | g | o | l | $ | g |

| | | | | | | |
|---|---|---|---|---|---|---|
| $ | g | o | o | g | o | l |
| g | o | l | $ | g | o | o |
| g | o | o | g | o | l | $ |
| l | $ | g | o | o | g | o |
| o | g | o | l | $ | g | o |
| o | l | $ | g | o | o | g |
| o | o | g | o | l | $ | g |

- Letter ‘o’ appear 3 times in the Last/**BWT** column *L* in the example above, at positions $i_1 = 2, i_2 = 4, i_3 = 5$.
- $L[i_1]$ maps to $F[5]$, $L[i_2]$ maps to $F[6]$, and finally $L[i_3]$ maps to $F[5]$ – the mapping of all ‘o’s points to 3 **consecutive rows** in $M$ starting position $\text{pos} = 5 =$ **Rank**(‘o’).
- For those positions, we see $F[i_1] =$ ‘g’, $F[i_2] =$ ‘l’, $F[i_3] =$ ‘o’ be their corresponding letters (in that order) in the **first column** *F*.
- Did you notice, these letters appear in the second column **in the same order** after the (block/run of) ‘o’s that appear in the first column of $M$?

# A crucial observation to automate $LF$-mapping (formalism)

| First ↓ | | Last ↓ |
|---|---|---|
| $\vdots$ | ......... | $\vdots$ |
| $F[i_1]$ | ......... | $L[i_1] = x$ |
| $\vdots$ | ......... | $\vdots$ |
| $F[i_2]$ | ......... | $L[i_2] = x$ |
| $\vdots$ | ......... | $\vdots$ |
| $\vdots$ | ......... | $\vdots$ |
| $\vdots$<br>$\vdots$ | .........<br>......... | $\vdots$<br>$\vdots$ |
| $F[i_3]$ | ......... | $L[i_3] = x$ |

| First ↓ | Second | | Last ↓ |
|---|---|---|---|
| $\vdots$ | ... | ...... | $\vdots$ |
| $\vdots$<br>$\vdots$ | ...<br>... | ......<br>...... | $\vdots$<br>$\vdots$ |
| $\vdots$ | ... | ...... | $\vdots$ |
| $\vdots$ | ... | ...... | $\vdots$ |
| $x = F[\text{pos}]$ | $F[i_1]$ | ...... | $L[\text{pos}]$ |
| $x = F[\text{pos}+1]$ | $F[i_2]$ | ...... | $L[\text{pos}+1]$ |
| $x = F[\text{pos}+2]$ | $F[i_3]$ | ...... | $L[\text{pos}+2]$ |
| $\vdots$ | ... | ...... | $\vdots$ |

Let the letter $x$ appear $k \geq 1$ times in **BWT column** $L$, at position $i_1, i_2, \ldots, i_k$ respectively. (In the above illustration $k = 3$.) Let $F[i_1], F[i_2], \cdots F[i_k]$ be their corresponding letters (in that order) in the **first column** $F$.

### Observation:

There **will have to be** $k$ **consecutive rows** in the sorted cyclic permulation matrix $M$ starting position $\text{pos} = \textbf{Rank}(x)$, where all identical $x$ would appear in a run-block, such that the second character after each $x$ will have to be the letters $F[i_1], F[i_2], \cdots F[i_k]$ in that order.

# Crucial rule to undertake *LF*-mapping

The observation on Slide #19 underpins ***LF*-mapping**, and hence the **backwards reconstruction** of $S$ from BWT($S$).

For any letter $L[i] =$ '$x$' in the **last or BWT column** $L$, there **has to be** a suffix starting with this **specific** $x$ in the **first column** $F$ at some position/index 'pos' in $F$. That is, $F[\text{pos}] = x$:

## Crucial Formula to find this 'pos'

$$\text{pos} = \mathbf{Rank}(x) + \mathbf{nOccurrences}(x, L[1...i))$$

$\mathbf{Rank}(x) =$ The position where $x$ first appears in $F$

$\mathbf{nOccurrences}(x, L[1..i)) =$ number of times $x$ appears in $L[1...i)$.[a]

---

[a] The range $[1...i)$ in $L$ is **EXCLUSIVE** of $i$

# Example – Find 'pos' in $F$ of the character 'o' @ $L[4]$

$$\text{pos} = \mathbf{Rank}(x) + \mathbf{nOccurrences}(x, L[1 \ldots i])$$

| **Pos** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1** | **$** | **g** | **o** | **o** | **g** | **o** | **l** | |
| **2** | **g** | **o** | **l** | **$** | **g** | **o** | **o** | |
| **3** | **g** | **o** | **o** | **g** | **o** | **l** | **$** | |
| **4** | **l** | **$** | **g** | **o** | **o** | **g** | **o** | ← |
| **5** | **o** | **g** | **o** | **l** | **$** | **g** | **o** | |
| **6** | **o** | **l** | **$** | **g** | **o** | **o** | **g** | |
| **7** | **o** | **o** | **g** | **o** | **l** | **$** | **g** | |

| | **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **1** | **$** | **g** | **o** | **o** | **g** | **o** | **l** |
| | **2** | **g** | **o** | **l** | **$** | **g** | **o** | **o** |
| | **3** | **g** | **o** | **o** | **g** | **o** | **l** | **$** |
| | **4** | **l** | **$** | **g** | **o** | **o** | **g** | **o** |
| | **5** | **o** | **g** | **o** | **l** | **$** | **g** | **o** |
| → | **6** | **o** | **l** | **$** | **g** | **o** | **o** | **g** |
| | **7** | **o** | **o** | **g** | **o** | **l** | **$** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

$\mathbf{Rank}(L[4] = \text{'o'}) = 5$

$\mathbf{nOccurrences}(\text{'o'}, L[1..4]) = 1$

$\text{pos} = 6$

# Recovering the full original string from BWT($S$) (in backwards direction)

## What information we currently have

BWT($S$) = `l o $ o o g g`

| Symbol | $ | g | l | o |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

# Recovering the full original string from BWT($S$) (in backwards direction)

## What information we currently have

BWT($S$) = **`l o $ o o g g`**

| Symbol | **`$`** | **`g`** | **`l`** | **`o`** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Inversion of BWT starts backwards:` $\cdots \leftarrow$ **`l`** $\leftarrow$ **`$`**

# Recovering the full original string from BWT($S$) (in backwards direction)

What information we currently have

BWT($S$) = **`l o $ o o g g`**

| Symbol | **`$`** | **`g`** | **`l`** | **`o`** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Inversion of BWT starts`
`backwards:` $\cdots \leftarrow$ **`l`** $\leftarrow$ **`$`**

Set $i = 1$.

# Recovering the full original string from BWT($S$) (in backwards direction)

What information we currently have

BWT($S$) = **l o $ o o g g**

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Inversion of BWT starts backwards:` $\cdots \leftarrow$ **l** $\leftarrow$ **$**

Set $i = 1$.

$L[i]$ =‘l’. The letter preceding this first letter in $L$ has to be **$** (always!).

# Recovering the full original string from BWT($S$) (in backwards direction)

## What information we currently have

BWT($S$) $=$ **`l o $ o o g g`**

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Inversion of BWT starts backwards:` $\cdots \leftarrow$ **`l`** $\leftarrow$ **`$`**

Set $i = 1$.

$L[i] =$ '`l`'. The letter preceding this first letter in $L$ has to be **`$`** (always!). Compute `pos` where this specific symbol '`l`' would appear in $F$.

**`Rank`**$(L[i] \equiv$ '`l`'$) = 4$

**`nOccurrences`**$($'`l`'$, L[1..i]) = 0$

`pos` $= 4$

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | $ | g | o | o | g | o | l |
| 2 | g | o | l | $ | g | o | o |
| 3 | g | o | o | g | o | l | $ |
| 4 | l | $ | g | o | o | g | o |
| 5 | o | g | o | l | $ | g | o |
| 6 | o | l | $ | g | o | o | g |
| 7 | o | o | g | o | l | $ | g |

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**l $**

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | $ | g | o | o | g | o | l |
| 2 | g | o | l | $ | g | o | o |
| 3 | g | o | o | g | o | l | $ |
| 4 | l | $ | g | o | o | g | o |
| 5 | o | g | o | l | $ | g | o |
| 6 | o | l | $ | g | o | o | g |
| 7 | o | o | g | o | l | $ | g |

We now have the **LF-mapping** of the letter ‘**l**’. **How?**

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**l $**

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | **$** | **g** | **o** | **o** | **g** | **o** | **l** |
| **2** | **g** | **o** | **l** | **$** | **g** | **o** | **o** |
| **3** | **g** | **o** | **o** | **g** | **o** | **l** | **$** |
| **4** | **l** | **$** | **g** | **o** | **o** | **g** | **o** |
| **5** | **o** | **g** | **o** | **l** | **$** | **g** | **o** |
| **6** | **o** | **l** | **$** | **g** | **o** | **o** | **g** |
| **7** | **o** | **o** | **g** | **o** | **l** | **$** | **g** |

We now have the **LF-mapping** of the letter '**l**'. **How?**

- $F[\text{pos}{=}4]$ is the same letter as $L[i = 1]$ (from previous slide).

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**l $**

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | **$** | **g** | **o** | **o** | **g** | **o** | **l** |
| **2** | **g** | **o** | **l** | **$** | **g** | **o** | **o** |
| **3** | **g** | **o** | **o** | **g** | **o** | **l** | **$** |
| **4** | **l** | **$** | **g** | **o** | **o** | **g** | **o** |
| **5** | **o** | **g** | **o** | **l** | **$** | **g** | **o** |
| **6** | **o** | **l** | **$** | **g** | **o** | **o** | **g** |
| **7** | **o** | **o** | **g** | **o** | **l** | **$** | **g** |

We now have the **`LF-mapping`** of the letter ‘**`l`**’. **`How?`**

- $F[\text{pos}=4]$ is the same letter as $L[i=1]$ (from previous slide).
- $L[\text{pos}]$ will **`precede`** $F[\text{pos}]$ in the reference string.

# Recovering the original string from BWT($S$)

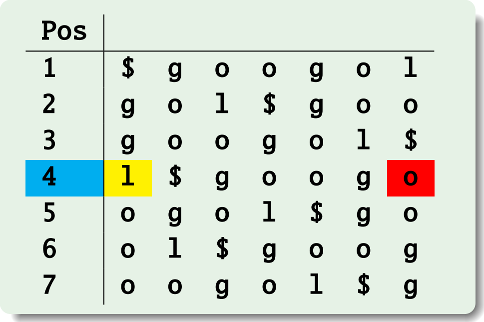
## What information we currently have

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**l $**



| **Pos** |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| **1** | $ | g | o | o | g | o | l |
| **2** | g | o | l | $ | g | o | o |
| **3** | g | o | o | g | o | l | $ |
| **4** | l | $ | g | o | o | g | o |
| **5** | o | g | o | l | $ | g | o |
| **6** | o | l | $ | g | o | o | g |
| **7** | o | o | g | o | l | $ | g |

We now have the **LF-mapping** of the letter '**l**'. **How?**

- $F[\text{pos}=4]$ is the same letter as $L[i=1]$ (from previous slide).
- $L[\text{pos}]$ will **precede** $F[\text{pos}]$ in the reference string.
- This gives the mapping to reconstruct/recover one more letter in the backwards direction.

# Recovering the original string from BWT($S$)

## What information we currently have

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**l $**

| Pos |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 1 | $ | g | o | o | g | o | l |
| 2 | g | o | l | $ | g | o | o |
| 3 | g | o | o | g | o | l | $ |
| 4 | l | $ | g | o | o | g | o |
| 5 | o | g | o | l | $ | g | o |
| 6 | o | l | $ | g | o | o | g |
| 7 | o | o | g | o | l | $ | g |

We now have the **`LF-mapping`** of the letter '**l**'. **`How?`**

- $F[\text{pos}=4]$ is the same letter as $L[i=1]$ (from previous slide).
- $L[\text{pos}]$ will **`precede`** $F[\text{pos}]$ in the reference string.
- This gives the mapping to reconstruct/recover one more letter in the backwards direction.

`Reconstructed string (so far)` **o l $**

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**o l $**

Now reset $i = \text{pos} = 4$

$\texttt{Rank}(L[i] = \text{‘o’}) = 5$

$\texttt{nOccurrences}(\text{‘o’}, L[1..4]) = 1$

(new) $\text{pos} = 6$

$L[\text{pos}] = \texttt{g}$

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | $ | g | o | o | g | o | l |
| **2** | g | o | l | $ | g | o | o |
| **3** | g | o | o | g | o | l | $ |
| **4** | l | $ | g | o | o | g | o |
| **5** | o | g | o | l | $ | g | o |
| **6** | o | l | $ | g | o | o | g |
| **7** | o | o | g | o | l | $ | g |

`Reconstructed string (so far)` **g o l $**

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**`g o l $`**

reset $i = \text{pos} = 6$

$\textbf{Rank}(L[i] = \text{'g'}) = 2$

$\texttt{nOccurrences}(\text{'g'}, L[1..6]) = 0$

(new) $\text{pos} = 2$

$L[\text{pos}] = \texttt{o}$

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | **$** | **g** | **o** | **o** | **g** | **o** | **l** |
| **2** | **g** | **o** | **l** | **$** | **g** | **o** | **o** |
| **3** | **g** | **o** | **o** | **g** | **o** | **l** | **$** |
| **4** | **l** | **$** | **g** | **o** | **o** | **g** | **o** |
| **5** | **o** | **g** | **o** | **l** | **$** | **g** | **o** |
| **6** | **o** | **l** | **$** | **g** | **o** | **o** | **g** |
| **7** | **o** | **o** | **g** | **o** | **l** | **$** | **g** |

`Reconstructed string (so far)` **`o g o l $`**

# Recovering the original string from BWT($S$)

**What information we currently have**

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**o g o l $**

Reset $i = \text{pos} = 2$

$\textbf{Rank}(L[i] = \text{`o'}) = 5$

$\textbf{nOccurrences}(\text{`o'}, L[1..2]) = 0$

(new) $\text{pos} = 5$

$L[\text{pos}] = \textbf{o}$

| Pos |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 1 | $ | g | o | o | g | o | l |
| 2 | g | o | l | $ | g | o | o |
| 3 | g | o | o | g | o | l | $ |
| 4 | l | $ | g | o | o | g | o |
| 5 | o | g | o | l | $ | g | o |
| 6 | o | l | $ | g | o | o | g |
| 7 | o | o | g | o | l | $ | g |

`Reconstructed string (so far)` **o o g o l $**

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | **l** | **o** | **$** | **o** | **o** | **g** | **g** |

| Symbol | **$** | **g** | **l** | **o** |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**o o g o l $**

Reset $i =$ pos $= 5$

**Rank**($L[i] =$ ‘o’) $= 5$

**nOccurrences**(‘o’, $L[1..5]$)) $= 2$

(new) pos $= 7$

$L$[pos] $=$ **g**

| Pos | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | $ | g | o | o | g | o | l |
| 2 | g | o | l | $ | g | o | o |
| 3 | g | o | o | g | o | l | $ |
| 4 | l | $ | g | o | o | g | o |
| 5 | o | g | o | l | $ | g | o |
| 6 | o | l | $ | g | o | o | g |
| 7 | o | o | g | o | l | $ | g |

`Reconstructed string (so far)` **g o o g o l $**

# Recovering the original string from BWT($S$)

## What information we currently have

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| BWT($S$) | `l` | `o` | `$` | `o` | `o` | `g` | `g` |

| Symbol | `$` | `g` | `l` | `o` |
|---|---|---|---|---|
| Rank | 1 | 2 | 4 | 5 |

`Reconstructed string (so far)`

**`g o o g o l $`**

Reset $i = \text{pos} = 7$

**Rank**$(L[i] = \text{‘g’}) = 2$

**nOccurrences**$(\text{‘g’}, L[1..7]) = 1$

(new) $\text{pos} = 3$

$L[\text{pos}] = \$$

| **Pos** | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | $ | g | o | o | g | o | l |
| 2 | g | o | l | $ | g | o | o |
| 3 | g | o | o | g | o | l | $ |
| 4 | l | $ | g | o | o | g | o |
| 5 | o | g | o | l | $ | g | o |
| 6 | o | l | $ | g | o | o | g |
| 7 | o | o | g | o | l | $ | g |

When $ is encounted, **STOP!!!**. Full string has been reconstructed.

# Part II: Exact pattern matching using BWT

# Summary of slides so far...

- We understood what BWT is and how to invert it.
- Recall, we introduced string pattern matching in weeks 1-2. We saw:
  - Naïve algorithm takes $O(m * n)$-time, worst-case
  - Z-algorithm, BM, KMP all have a worst-case that takes $O(m + n)$-time.

## Question to consider

Assume we have a **`very very big text`**, and a **`large number very very short patterns`** to search in that text for **exact matches**. Would the above algorithms for pattern matching be useful?

## In the subsequent slides...

You will see how Burrows-Wheelers Transform of any large reference text can be used to address this question effectively and efficiently – this algorithm is as beautiful as things can get in data structures and algorithms!

# Does **`pat[1...m]`** appear in **`txt[1...n]`**? If so, How many times?

- Number of times a pattern appears in some reference text is called **`multiplicity`**.
- Assume that we have preprocessed **`txt[1...n]`** to obtain its BWT. Then pattern matching becomes rather straight-forward, and requires backward search on **`pat[1...m]`**
- Initialize two pointers on BWT of **`txt`**:
  - **`sp = 1`** (for start of the range)
  - **`ep = n`** (for end of the range)
- these pointers are updated using the rules:
  - **`sp`** $=$ rank(**`pat[i]`**) $+$ nOccurrences(**`pat[i]`**, L[1...sp))
  - **`ep`** $=$ rank(**`pat[i]`**) $+$ nOccurrences(**`pat[i]`**, L[1...ep]) - 1 *

---

*ALERT!!! In the **`ep`** computation above, the range L[1...ep] is INCLUSIVE of ep. In the previous case (for sp), it was EXCLUSIVE.

# Computing Multiplicity and Occurrences

```
pat[1...m] = g o              // pattern
txt[1...n] = g o o g o l $  // reference text
```

```
          pos  = 1 2 3 4 5 6 7  // array index
     L[1...n] = l o $ o o g g  // BWT of ref. text
 suffix index = 7 4 1 6 3 5 2  // suffix array index
```

Initialize pointers **sp** to 1 and **ep** to $n = 7$.

| *SA* ↓ | **Pos** ↓ | | | | | | | **L** ↓ | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | $ | g | o | o | g | o | l | ← **sp** |
| 4 | 2 | g | o | l | $ | g | o | o | |
| 1 | 3 | g | o | o | g | o | l | $ | |
| 6 | 4 | l | $ | g | o | o | g | o | |
| 3 | 5 | o | g | o | l | $ | g | o | |
| 5 | 6 | o | l | $ | g | o | o | g | |
| 2 | 7 | o | o | g | o | l | $ | g | ← **ep** |

# Example of pattern matching on BWT

```
    pat[1...m] = g o            // pattern
    txt[1...n] = g o o g o l $  // reference text

          pos  = 1 2 3 4 5 6 7  // array index
      L[1...n] = l o $ o o g g  // BWT of ref. text
  suffix index = 7 4 1 6 3 5 2  // suffix array index
```

**sp** = rank(**pat[i]**) + nOccurrences(**pat[i]**, L[1...sp))
**ep** = rank(**pat[i]**) + nOccurrences(**pat[i]**, L[1...ep]) - 1

Initialize **sp = 1** **ep = 7** **i = m = 2**
Search **pat[1...m]** backwards.
So, start with **pat[m=2] = 'o'**
rank(**o**) = 5 nOccurrences(**o**,L[1...sp)) = 0 nOccurrences(**o**,L[1...ep]) = 3
**(updated) sp =** 5 + 0 **(updated) ep =** 5 + 3 -1
These updated pointers give the range (in $M$) of all suffixes starting with **o**.

# Computing Multiplicity and Occurrences

```
    pat[1...m] = g o            // pattern
    txt[1...n] = g o o g o l $  // reference text
          pos  = 1 2 3 4 5 6 7  // array index
      L[1...n] = l o $ o o g g  // BWT of ref. text
  suffix index = 7 4 1 6 3 5 2  // suffix array index
```

Updated sp and ep illustration after searching for **o** is completed (see previous slide).

| *SA* ↓ | **Pos** ↓ | | | | | | | **L** ↓ | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | $ | g | o | o | g | o | l | |
| 4 | 2 | g | o | l | $ | g | o | o | |
| 1 | 3 | g | o | o | g | o | l | $ | |
| 6 | 4 | l | $ | g | o | o | g | o | |
| 3 | 5 | o | g | o | l | $ | g | o | ← sp |
| 5 | 6 | o | l | $ | g | o | o | g | |
| 2 | 7 | o | o | g | o | l | $ | g | ← ep |

# Example of pattern matching on BWT

```
    pat[1...m] = g o            // pattern
    txt[1...n] = g o o g o l $  // reference text

          pos  = 1 2 3 4 5 6 7  // array index
      L[1...n] = l o $ o o g g  // BWT of ref. text
  suffix index = 7 4 1 6 3 5 2  // suffix array index
```

**sp** = rank(**pat[i]**) + nOccurrences(**pat[i]**, L[1...sp))
**ep** = rank(**pat[i]**) + nOccurrences(**pat[i]**, L[1...ep]) - 1

Current **sp = 5** **ep = 7**
Continue searching backwards on the pattern. Now for **pat[1] = 'g'**
rank(**g**) = 2 nOccurrences(**g**,L[1...sp)) = 0 nOccurrences(**g**,L[1...ep]) = 2
**(updated) sp =** 2 + 0 **(updated) ep =** 2 + 2 -1 = 3
These updated pointers give the range of all suffixes starting with **go**.

# Computing Multiplicity and Occurrences

```
    pat[1...m] = g o              // pattern
    txt[1...n] = g o o g o l $    // reference text
          pos  = 1 2 3 4 5 6 7    // array index
      L[1...n] = l o $ o o g g    // BWT of ref. text
  suffix index = 7 4 1 6 3 5 2    // suffix array index
```

Updated sp and ep illustration after searching for **g** is completed (see previous slide).

| *SA* ↓ | **Pos** ↓ | | | | | | | **L** ↓ | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | $ | g | o | o | g | o | l | |
| 4 | 2 | g | o | l | $ | g | o | o | ← **sp** |
| 1 | 3 | g | o | o | g | o | l | $ | ← **ep** |
| 6 | 4 | l | $ | g | o | o | g | o | |
| 3 | 5 | o | g | o | l | $ | g | o | |
| 5 | 6 | o | l | $ | g | o | o | g | |
| 2 | 7 | o | o | g | o | l | $ | g | |

# Computing Multiplicity and Occurrences

```
   pat[1...m] = g o               // pattern
   txt[1...n] = g o o g o l $  // reference text

         pos  = 1 2 3 4 5 6 7  // array index
     L[1...n] = l o $ o o g g  // BWT of ref. text
 suffix index = 7 4 1 6 3 5 2  // suffix array index
```

- Once the **entire** **pat**[m. . . 1] is searched backwards, the resulting updated **sp** and **ep** values give the range of positions (in $M$) which all start with **pat**[1. . . m].
- Multiplicity = **ep** - **sp** +1. In this example, Multiplicity of "**go**" in the reference text is $3 - 2 + 1 = 2$
- Note, Multiplicity = 0 (i.e., no occurrences found), when **ep**<**sp**
- To identify the positions in **txt**[1. . . n] where the pattern occurs, if any, simply look up the **suffix array indexes** in the range [**sp**,**ep**].

# Computing Multiplicity and Occurrences

```
    pat[1...m] = g o            // pattern
    txt[1...n] = g o o g o l $  // reference text
          pos  = 1 2 3 4 5 6 7  // array index
      L[1...n] = l o $ o o g g  // BWT of ref. text
  suffix index = 7 4 1 6 3 5 2  // suffix array index
```

| *SA*<br>↓ | **Pos**<br>↓ | | | | | | | **L**<br>↓ | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | $ | g | o | o | g | o | l | |
| 4 | 2 | g | o | l | $ | g | o | o | ← sp |
| 1 | 3 | g | o | o | g | o | l | $ | ← ep |
| 6 | 4 | l | $ | g | o | o | g | o | |
| 3 | 5 | o | g | o | l | $ | g | o | |
| 5 | 6 | o | l | $ | g | o | o | g | |
| 2 | 7 | o | o | g | o | l | $ | g | |

**Multiplicity**= `ep - sp + 1` $= 3 - 2 + 1 = 2$

Where does the **`pat[1...m]`** occur in **`txt[1...n]`**?

Lookup the corresp. *SA* in the range `[sp..ep]`: positions 4 and 1 in the reference text (these positions will be unordered, but correct!).

# Exact pattern matching – Summary

- Naive algorithm: $O(m * n)$-time, worst-case
- Z-algorithm, Boyer-Moore, KMP: $O(n)$-time worst-case
- Using BWT (with $O(n)$ auxiliary space): $O(m)$-time

**In the next lecture...**

Linear-time Suffix Tree (and suffix array) construction using Ukkonen's algorithm

-=o0o=-

END

-=o0o=-