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 S

Given a string $S[1\dots n]$, assume it ends always with a special unique terminal character smaller than any other character in S: $S[n] = \$$ (say).

Burrows-Wheeler Transform (BWT) of any given string $S[1\dots n]$

1. BWT(S) is a specific permutation of S, defined by the **last column** (L) of the matrix M containing **sorted cyclic permutations** of S. (See example below.)

2. Equivalently, BWT(S) is the string formed by the letters that (cyclically) **precede** those in the first column (F) of M.
   - In other words, subtracting one from the **suffix array** SA (that stores indexes of sorted suffixes of S) gives you the information of the last column.

Example: $S = \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \mathtt{g} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{o} & \mathtt{l} & \mathtt{\$} \end{array}$ $\Longrightarrow$ $BWT(S) = \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \mathtt{l} & \mathtt{o} & \mathtt{\$} & \mathtt{o} & \mathtt{o} & \mathtt{g} & \mathtt{g} \end{array}$

$M =$

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	o	g
2	o	o	g	o	l	$	g

$\equiv$

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	o	g
2	o	o	g	o	l	$	g

Algorithmically constructing BWT($S[1 \ldots n]$)

1. Construct Suffix Array SA of the input string $S[1 \ldots n]$.
   - You have learnt the '**prefix-doubling**' method of constructing a suffix array in FIT2004. Quiz: What is its time complexity? (Revise)
   - **Heads up**: By the end of next week, you would have learnt a $\mathcal{O}(n)$-time (and space) algorithm to construct a suffix array – this complexity is asymptotically optimal for the problem.
2. Construct BWT from SA in $\mathcal{O}(n)$ time (see previous slide for the relationship).

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 $

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

E.g., **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 $

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

E.g., **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 $

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

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 $S[1 \dots n]$ from BWT(S) $= L[1 \dots n]$.[a]
- We will use the notation BWT^{-1} to denote the **inverse** of a BWT of a string. By inverse it is implied that BWT^{-1}(BWT(S)) $= S$

[a] This is indeed magical (!) at first glance.

Naïve method to invert BWT(S) – never use this method!

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

Example: BWT('googol$')

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

The matrix below is shown for reference to eyeball the reconstructed (yellow) columns.

$$M = \begin{array}{|c|cccccc} \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l}\\ \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o}\\ \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$}\\ \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o}\\ \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o}\\ \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} & \texttt{o} & \texttt{o} & \texttt{g}\\ \texttt{o} & \texttt{o} & \texttt{g} & \texttt{o} & \texttt{l} & \texttt{\$} & \texttt{g} \end{array}$$

This sorting reconstructs the **first column** of the matrix M.
But **we know** that the **first column** succeeds (comes after) the **last (BWT) column** (in the sense of *cyclic permutations* of S)...

Naïve method to invert BWT(S) – never use this method!

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: googol$

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

The matrix below is shown for reference to eyeball the reconstructed (yellow) columns.

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

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**)...

Naïve method to invert BWT(S) – never use this method!

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: googol$

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

The matrix below is shown for reference to eyeball the reconstructed (yellow) columns.

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

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**)...

Naïve method to invert BWT(S) – never use this method!

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.

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

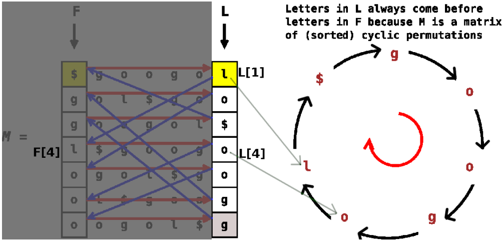
The naive approach is **highly inefficient** in both space and time.

Quiz: What is the time and space complexity of this approach?

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

Inverting via LF-Mapping: the bijection between the Last (L) and First (F) columns

With just the knowledge of letters $L[1 \dots n]$, and without having to reconstruct M, the original string $S[1 \dots n]$ can be recovered using LF-mapping:

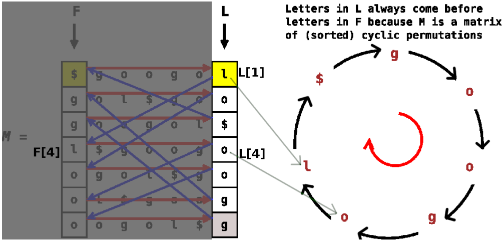


Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

- Each letter $L[i]$ in the last column (BWT) maps to some letter $F[pos]$ in the first column (refer slide 9).

Inverting via LF-**Mapping**: the **bijection** between the Last (L) and First (F) columns

With just the knowledge of letters $L[1 \dots n]$, and without having to reconstruct M, the original string $S[1 \dots n]$ can be recovered using LF-mapping:

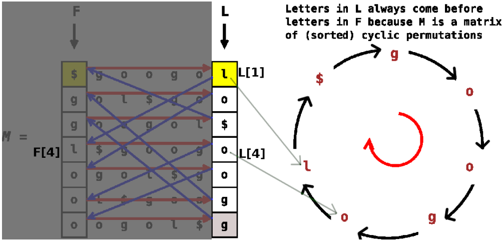


Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

- Each letter $L[i]$ in the last column (BWT) **maps to** some letter $F[\text{pos}]$ in the first column (refer slide 9).
- To start the recovery, we know $S[n]$ has to be \$. Further, we also know that $L[i = 1]$ has to be the second last letter ($S[n-1]$) in the original string – how???

Inverting via LF-**Mapping**: the **bijection** between the Last (L) and First (F) columns

With just the knowledge of letters $L[1\ldots n]$, and without having to reconstruct M, the original string $S[1\ldots n]$ can be recovered using LF-mapping:

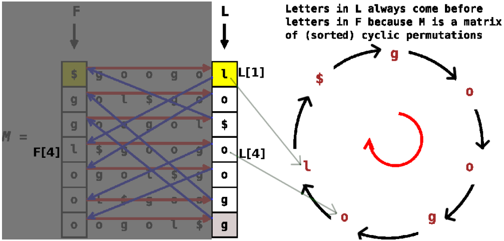


Letters in L always come before letters in F because M is a matrix of (sorted) cyclic permutations

- Each letter $L[i]$ in the last column (BWT) **maps to** some letter $F[\text{pos}]$ in the first column (refer slide 9).
- To start the recovery, we know $S[n]$ has to be \$. Further, we also know that $L[i=1]$ has to be the second last letter ($S[n-1]$) in the original string – how???
- Solely using the information in the BWT column (L), can we compute the mapping: $L[i] \mapsto F[\text{pos}]$?

Inverting via LF-**Mapping**: the **bijection** between the Last (L) and First (F) columns

With just the knowledge of letters $L[1 \ldots n]$, and without having to reconstruct M, the original string $S[1 \ldots n]$ can be recovered using LF-mapping:



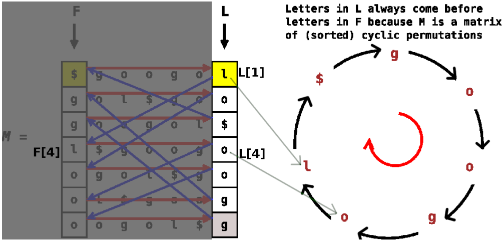
	F						L
M =	$	g	o	o	g	o	l
	g	o	l	$	g	o	o
	g	o	o	g	o	l	$
	l	$	g	o	o	g	o
	o	g	o	l	$	g	o
	o	l	$	g	o	o	g
	o	o	g	o	l	$	g

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

- Each letter $L[i]$ in the last column (BWT) **maps to** some letter $F[\text{pos}]$ in the first column (refer slide 9).
- To start the recovery, we know $S[n]$ has to be \$. Further, we also know that $L[i = 1]$ has to be the second last letter $(S[n-1])$ in the original string – how???
- Solely using the information in the BWT column (L), can we compute the mapping: $L[i] \mapsto F[\text{pos}]$?
- **Punchline:** If any $L[i]$ maps to some $F[\text{pos}]$, then $L[\text{pos}]$ precedes $L[i]$ in the original string (refer slide 11)

Inverting via LF-**Mapping**: the **bijection** between the Last (L) and First (F) columns

With just the knowledge of letters $L[1 \ldots n]$, and without having to reconstruct M, the original string $S[1 \ldots n]$ can be recovered using LF-mapping:



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

- Each letter $L[i]$ in the last column (BWT) **maps to** some letter $F[\text{pos}]$ in the first column (refer slide 9).
- To start the recovery, we know $S[n]$ has to be \$. Further, we also know that $L[i = 1]$ has to be the second last letter ($S[n-1]$) in the original string – how???
- Solely using the information in the BWT column (L), can we compute the mapping: $L[i] \mapsto F[\text{pos}]$?
- **Punchline:** If any $L[i]$ maps to some $F[\text{pos}]$, then $L[\text{pos}]$ precedes $L[i]$ in the original string (refer slide 11)
- Thus, by iteratively applying the mapping $L[i] \mapsto F[\text{pos}]$, starting from $i = 1$, we can **recover** the original string $S[n \ldots 1]$, one letter at a time, in the **backward** direction.

An 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 $F[7]$ – the mapping of all ‘o’s points to 3 **`consecutive rows`** in M starting position $\text{pos} = 5 = \textbf{Rank}(\text{‘o’})$.
- 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?

An observation to automate LF-**mapping** (formalism)

First ↓		Last ↓
$\vdots$	$\ldots\ldots\ldots$	$\vdots$
$F[i_1]$	$\cdots\cdots\cdots$	$L[i_1] = x$
$\vdots$ $\vdots$	$\ldots\ldots\ldots$ $\ldots\ldots\ldots$	$\vdots$ $\vdots$
$F[i_2]$	$\cdots\cdots\cdots$	$L[i_2] = x$
$\vdots$	$\ldots\ldots\ldots$	$\vdots$
$\vdots$ $\vdots$	$\ldots\ldots\ldots$ $\ldots\ldots\ldots$	$\vdots$ $\vdots$
$F[i_k]$	$\cdots\cdots\cdots$	$L[i_k] = x$

First ↓	2nd		Last ↓
$\vdots$ $\vdots$ $\vdots$ $\vdots$	$\ldots$ $\ldots$ $\ldots$ $\ldots$	$\ldots\ldots$ $\ldots\ldots$ $\ldots\ldots$ $\ldots\ldots$	$\vdots$ $\vdots$ $\vdots$ $\vdots$
$x = L[i_1] \equiv F[$pos$]$	$F[i_1]$	$\cdots\cdots$	$L[$pos$]$
$x = L[i_2] \equiv F[$pos+1$]$	$F[i_2]$	$\cdots\cdots$	$L[$pos+1$]$
$\vdots$	$\ldots$	$\ldots\ldots$	$\vdots$
$x = L[i_k] \equiv F[$pos+k-1$]$	$F[i_3]$	$\cdots\cdots$	$L[$pos+k-1$]$
$\vdots$	$\ldots$	$\ldots\ldots$	$\vdots$

Key Observation

- Let the letter x appear $k \geq 1$ times in **BWT column** L at positions $\{i_1, i_2, \ldots, i_k\}$.
- Then there **will be** k **consecutive rows** starting with the letter x in the sorted cyclic permulation matrix M, starting at position pos $=$ **Rank**(x).
- Specifically: $L[i_1] \mapsto F[\text{pos} + 0], L[i_2] \mapsto F[\text{pos} + 1], \ldots, L[i_k] \mapsto F[\text{pos} + k - 1]$.

Crucial rule to implement LF-mapping

The key observation on Slide #20 provides the mechanism to perform LF**-mapping**, and therefore the **backward reconstruction** of S from BWT(S).

For any letter $L[i] =$ 'x' in the **last (BWT) column**, $L[i] \mapsto F[\text{pos}]$, where:

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

In the above block:
$\texttt{Rank}(x) =$ The position where x first appears in F
$\texttt{nOccurrences}(x, L[1..i)) =$ number of times x appears in $L[1...i)$.*
(Rank and nOccurrences data structures are precomputed from L before recovery)

*Note: The range $[1...i)$ in L is **EXCLUSIVE** of i

Example: 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

Example: 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

Set $i = 1$.

$L[i] =$'l'. Note: The letter preceding this first letter in L has to be $ (always!).

Example: 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

Set $i = 1$.

$L[i] =$'l'. Note: The letter preceding this first letter in L has to be $ (always!).

Inversion is backwards:

? ←**l$**

To recover one more character (backwards)...

Example: Recovering the full original string from BWT(S) (in backwards direction)

What information we currently have

$\mathrm{BWT}(S) = \texttt{l o \$ o o g g}$

Symbol	$	g	l	o
Rank	1	2	4	5

Set $i = 1$.

$L[i] =$'l'. Note: The letter preceding this first letter in L has to be $ (always!).

Inversion is backwards:

$? \leftarrow \texttt{l\$}$

To recover one more character (backwards)...

...compute pos where this specific symbol 'l' would appear in F.
$\texttt{Rank}(L[i]) = \texttt{Rank}(\texttt{'l'}) = 4$
$\texttt{nOccurrences}(\texttt{'l'}, L[1..i]) = 0$
$\texttt{pos} = 4 + 0 = 4$

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

Example: Recovering the full original string from BWT(S) (in backwards direction)

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

From the **LF-mapping** of $L[i = 1] \mapsto F[\text{pos} = 4]$, we can recover one more character. **How?**

- $L[\text{pos}]$ will **precede** $F[\text{pos}]$ in the reference string.
- Since pos $= 4$, $L[\text{pos}] = \text{o}$ is the character before 'l' in the original string.

Example: Recovering the full original string from BWT(S) (in backwards direction)

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 = pos $= 4$

...compute pos where this specific 'o' would appear in F.
Rank($L[i]$) = **Rank**('o') $= 5$
nOccurrences('o', $L[1..4)$) $= 1$
(new) pos $= 6$
Since pos $= 5 + 1 = 6$, L[pos] = g is the character before 'o' in the original string.

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**

Example: Recovering the full original string from BWT(S) (in backwards direction)

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 $i =$ pos $= 6$

...compute pos where this specific 'g' would appear in F.
Rank($L[i]$) = Rank('g') $= 2$
nOccurrences('g', $L[1..6)$) $= 0$
(new) pos $= 2$
Since pos $= 2 + 0$, L[pos] = o is the character before 'g' in the original string.

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

Example: Recovering the full original string from BWT(S) (in backwards direction)

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 $i =$ pos $= 2$

...compute pos where this specific 'o' would appear in F.

Rank($L[i]$) = **Rank**('o') $= 5$

nOccurrences('o', $L[1..2)$) $= 0$

(new) pos $= 5$

Since pos $= 5 + 0 = 5$, L[pos] $=$ o is the character before 'o' in the original string.

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

Example: Recovering the full original string from BWT(S) (in backwards direction)

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 $= 5$

...compute pos where this specific ‘o’ would appear in F.

Rank$(L[i]) =$ **Rank**(‘o’) $= 5$

nOccurrences(‘o’, $L[1..5]$) $= 2$

(new) pos $= 7$

Since pos $= 5 + 2 = 7$, $L[$pos$] =$ g is the character before ‘o’ in the original string.

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

Example: Recovering the full original string from BWT(S) (in backwards direction)

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 $

When the length of the recovered string grows to length of L, STOP.

If you continued, you will cycle back to the last character in the string, $S[n] = \$$.

Reset i = pos = 7

Rank($L[i]$ = 'g') = 2

nOccurrences('g', $L[1..7]$) = 1

(new) pos = 3

L[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

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 $\mathcal{O}(m * n)$-time, worst-case
  - Z-algorithm, Boyer-MooreM, KMP all have a worst-case that takes $\mathcal{O}(m + n)$-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 [‡]

[†] Reason why these update rules are correct – after that refer to the proof document on Moodle

[‡] ALERT!!! In the **ep** computation above, the range L[1...ep] is INCLUSIVE of ep. In the previous case (for sp), it was EXCLUSIVE. Reason why, and refer to the proof document on moodle.

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
        SA  = 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
        SA = 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**
Search **pat[m...1]** (i.e., backwards).
So, start with **pat[i=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
        SA = 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
        SA = 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
        SA = 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
        SA = 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
        SA = 7 4 1 6 3 5 2       // suffix array index
```

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	

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: $\mathcal{O}(m * n)$-time worst-case for each pattern query on the same text.
- Z-algorithm, Boyer-Moore, KMP: $\mathcal{O}(m + n)$-time worst-case for each pattern query on the same text.
- BWT One-time precomputation of BWT in $\mathcal{O}(n)$-time. Then (using $\mathcal{O}(n)$ auxiliary-space), $\mathcal{O}(m +$ multiplicity$)$-time worst case for each pattern query on the same text.

In the next lecture...

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

-=o0o=-
END
-=o0o=-