# COMP9319 Web Data Compression and Search

## Space Efficient Linear Time Construction of Suffix Arrays

# Suffix Array

- Sorted order of suffixes of a string *T*.
- Represented by the starting position of the suffix.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Text | M | I | S | S | I | S | S | I | P | P | I | $ |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Suffix Array | 12 | 11 | 8 | 5 | 2 | 1 | 10 | 9 | 7 | 4 | 6 | 3 |

# Brief History

- Introduced by Manber and Myers in 1989.
- Takes O(*n log n*) time, and 8n bytes.
- Many other non-linear time algorithms.

| Authors | Time | Space (bytes) |
|---|---|---|
| Manber & Myers | $n \log n$ | $8n$ |
| Sadakane | $n \log n$ | $9n$ |
| String-sorting | $n^2 \log n$ | $5n$ |
| Radix-sorting | $n^2$ | $5n$ |

# Our Result

1. Among the first linear-time direct suffix array construction algorithms. Solves an important open problem.
2. For constant size alphabet, only uses $8n$ bytes.
3. Easily implementable.
4. Can also be used as a space efficient suffix tree construction algorithm.

# Notation

- String $T = t_1 \ldots t_n$.
- Over the alphabet $\Sigma = \{1 \ldots n\}$.
- $t_n$ = '\$', '\$' is a unique character.
- $T_i = t_i \ldots t_n$, denotes the $i$-th suffix of $T$.
- For strings $\alpha$ and $\beta$, $\alpha < \beta$ denotes $\alpha$ is lexicographically smaller than $\beta$.

# Overview

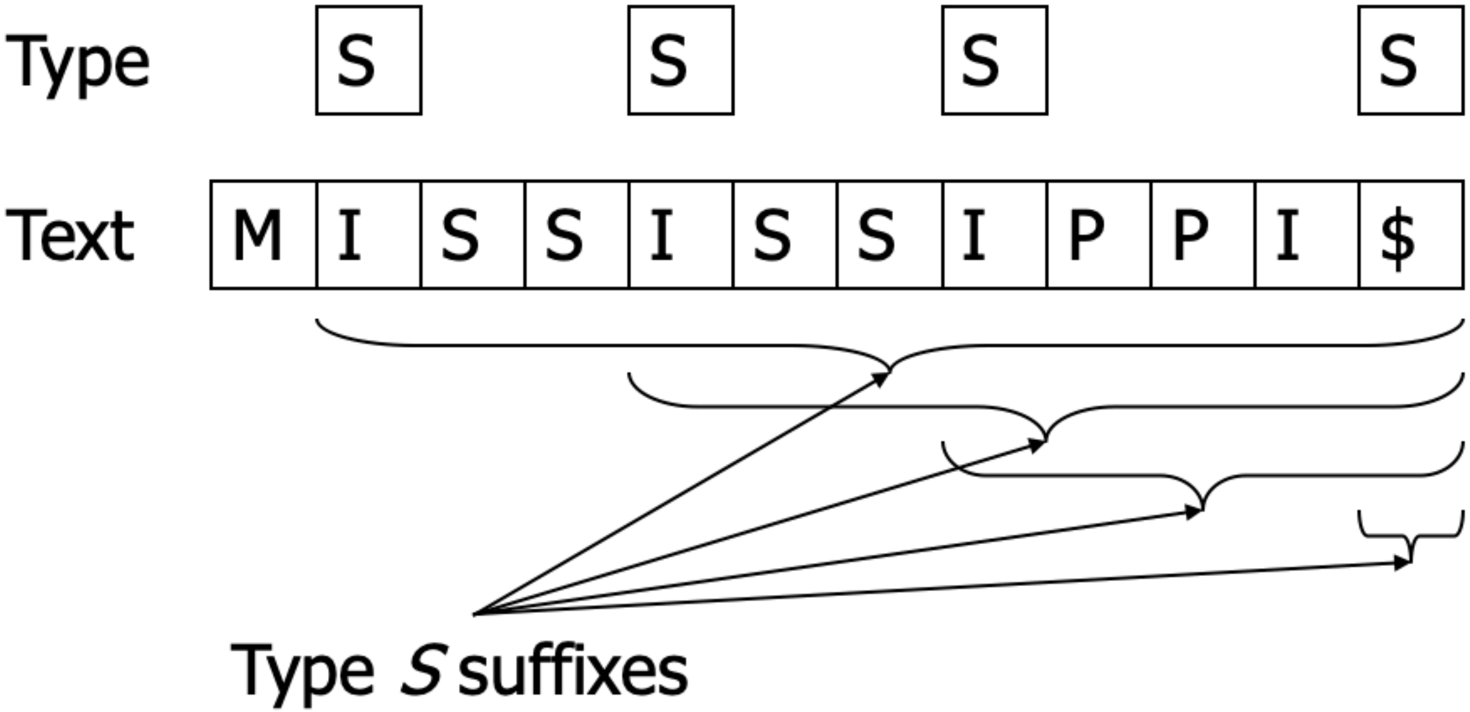
- Divide all suffixes of $T$ into two types.
  - Type S suffixes = $\{T_i \mid T_i < T_{i+1}\}$
  - Type L suffixes = $\{T_j \mid T_j > T_{j+1}\}$
  - The last suffix is both type $S$ and $L$.
- Sort all suffixes of one of the types.
- Obtain lexicographical order of all suffixes from the sorted ones.

# Identify Suffix Types

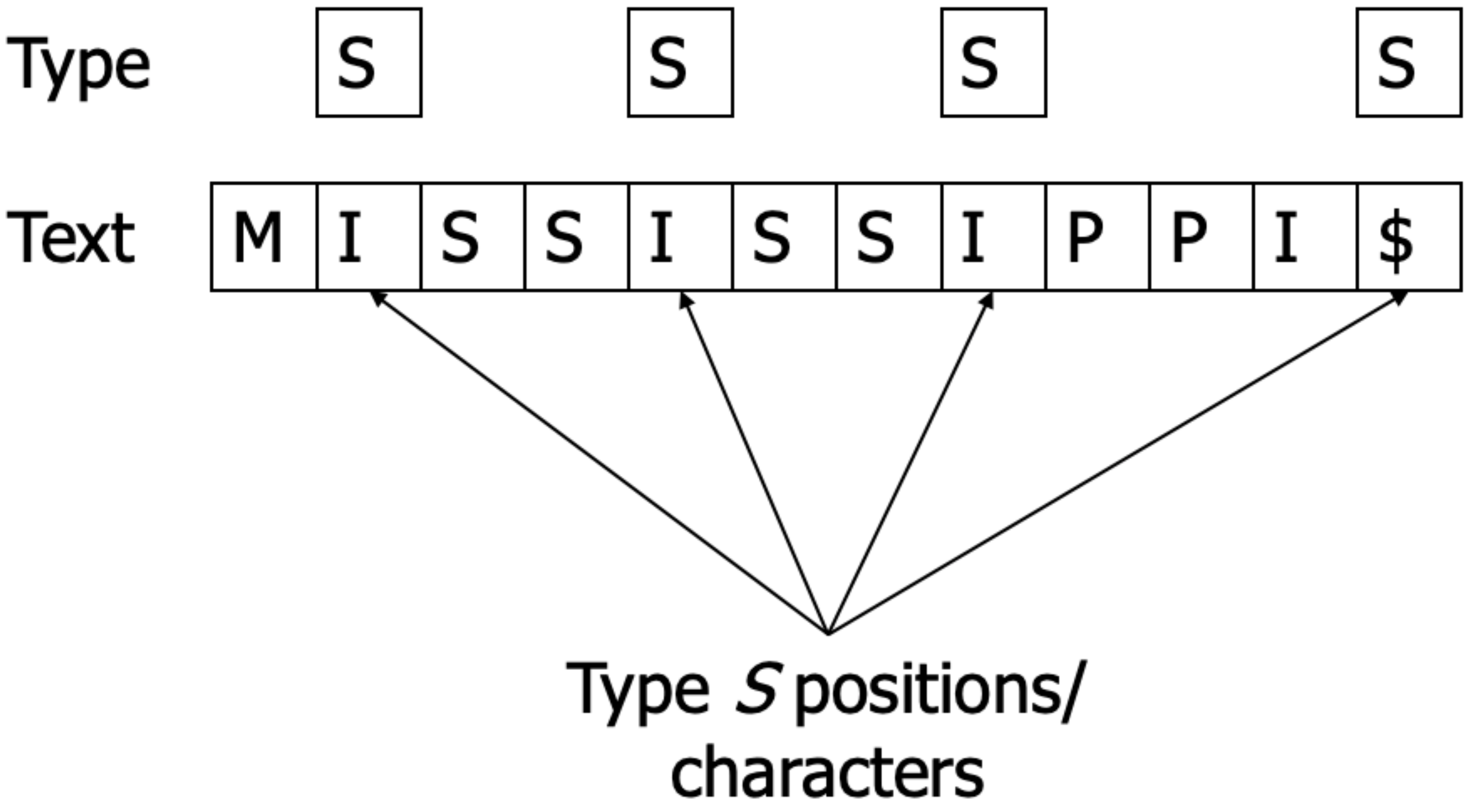
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Type | L | S | L | L | S | L | L | S | L | L | L | L/S |
| Text | M | I | S | S | I | S | S | I | P | P | I | $ |

The type of each suffix in *T* can be determined in one scan of the string.

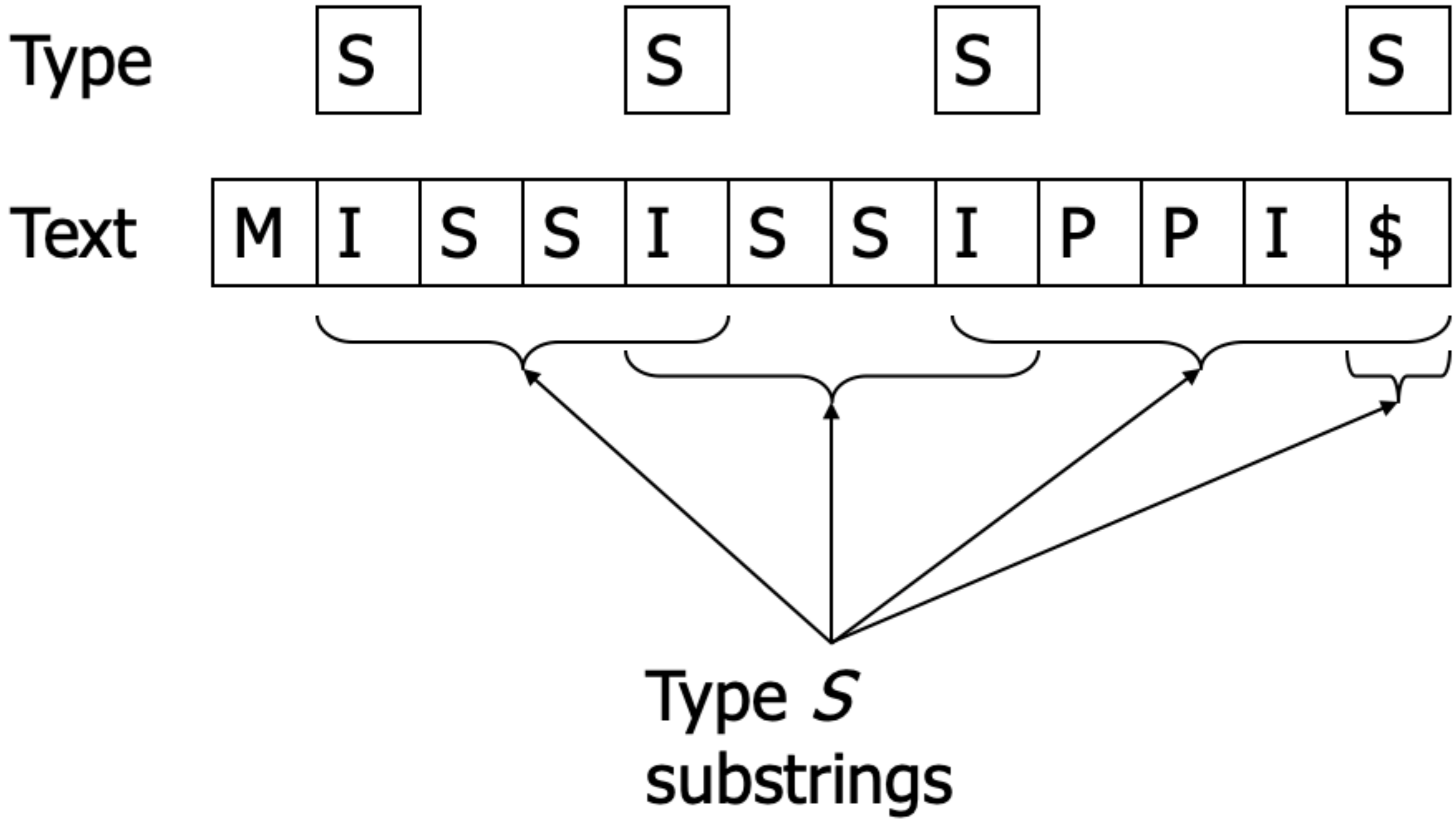
# Notation



# Notation



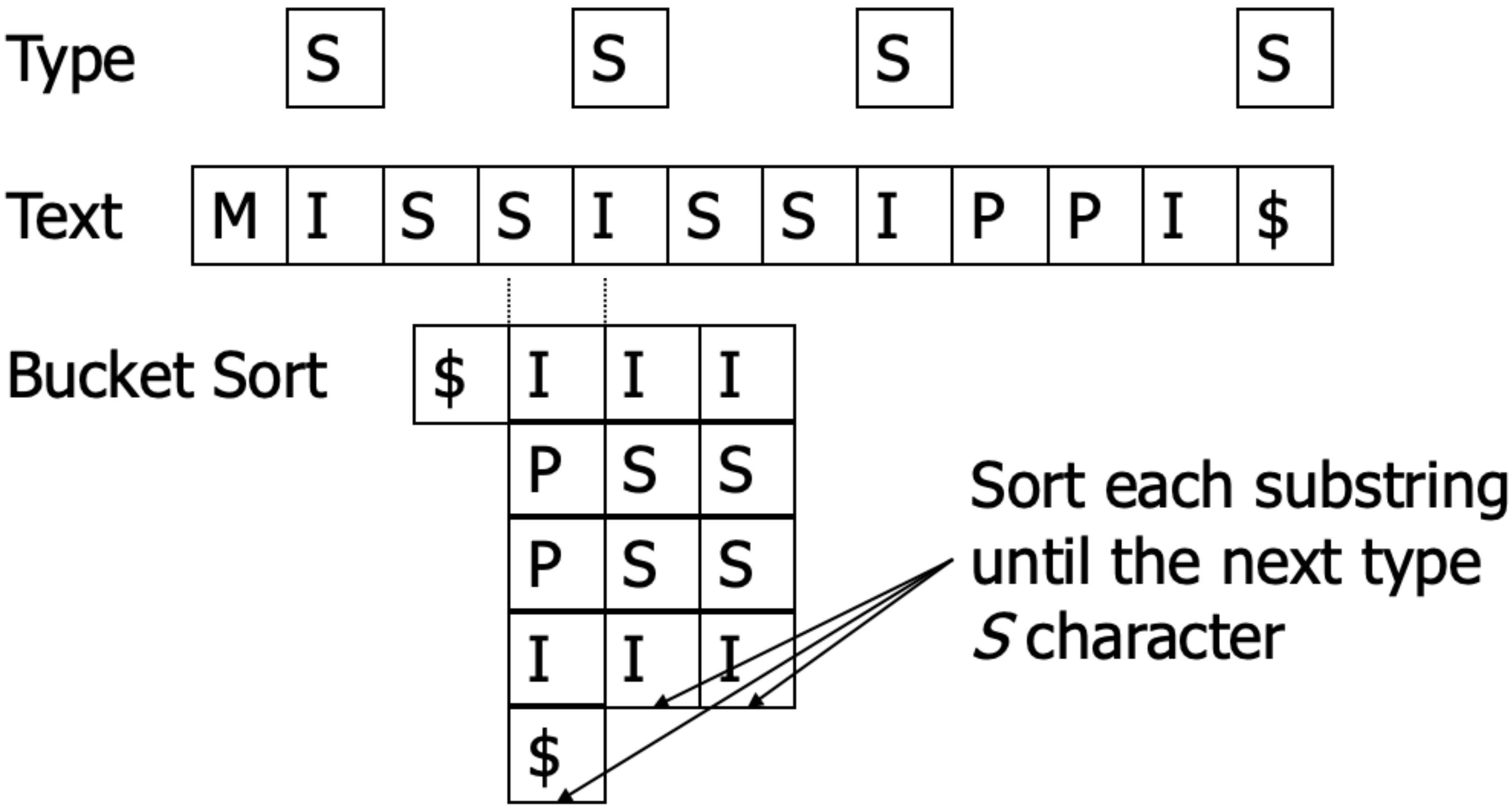
# Notation



# Sorting Type *S* Suffixes

- Sort all type *S* substrings.
- Replace each type *S* substrings by its bucket number.
- New string is the sequence of bucket numbers.
- Sorting all type *S* suffixes = Sorting all suffixes of the new string.

# Sorting Type *S* Substrings



# Sorting Type *S* Substrings

Type

Text

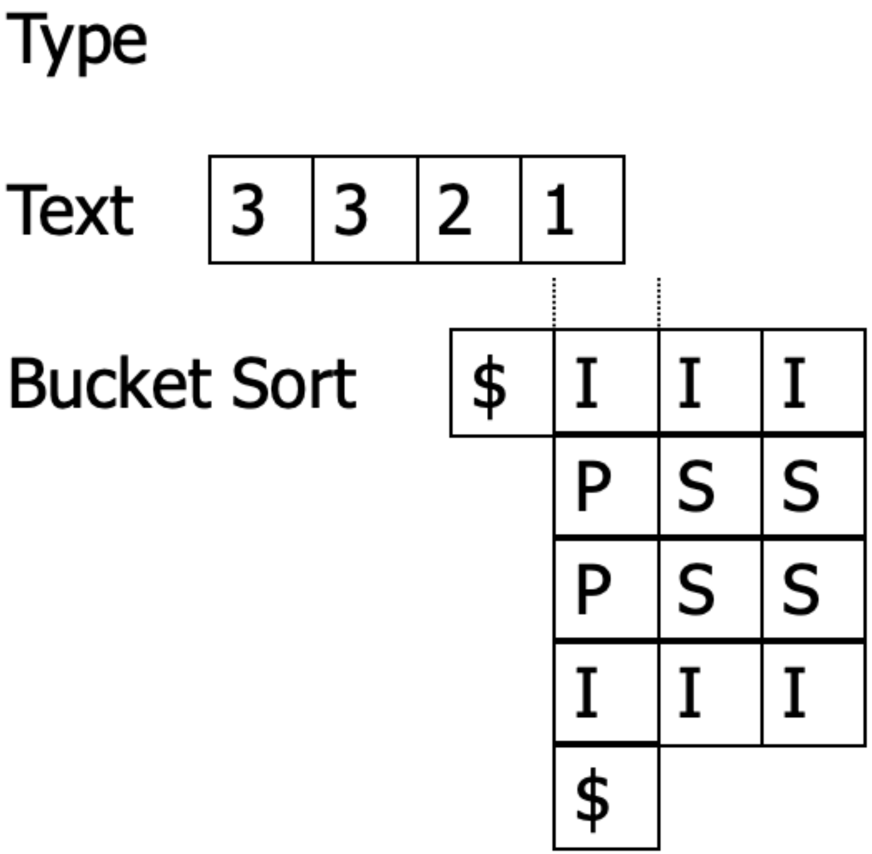
| | | | |
|---|---|---|---|
| 3 | 3 | 2 | 1 |

Bucket Sort

| | | | |
|---|---|---|---|
| $ | I | I | I |
| | P | S | S |
| | P | S | S |
| | I | I | I |
| | $ | | |

Substitute the substrings with the bucket numbers to obtain a new string. Apply sorting recursively to the new string.

# Sorting Type *S* Substrings




Bucket Sort takes potentially O($n^2$) time

# Solution

- Observation: Each character participates in the bucket sort at most twice.
  - Type *L* characters only participate in the bucket sort once.
- Solution:
  - Sort all the characters once.
  - Construct m lists according the distance to the closest type *S* character to the left

# Illustration

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Type | | S | | | S | | | S | | | | S |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Text | M | I | S | S | I | S | S | I | P | P | I | $ |
| Distance | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 4 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sorted Order of characters | 12 | 2 | 5 | 8 | 11 | 1 | 9 | 10 | 3 | 4 | 6 | 7 |

# Illustration

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Type | | S | | | S | | | S | | | | S |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Text | M | I | S | S | I | S | S | I | P | P | I | $ |
| Distance | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 4 |

The Lists

| | | |
|---|---|---|
| 9 | 3 | 6 |
| 10 | 4 | 7 |
| 5 | 8 | 11 |
| 12 | | |

Sort the type *S* substrings using the lists

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | M | I | S | S | I | S | S | I | P | P | I | $ |
| Type | | S | | | S | | | S | | | | S |
| Pos | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| A | 12 | 2 | 5 | 8 | 11 | 1 | 9 | 10 | 3 | 4 | 6 | 7 |

Step 1. Record the S−distances

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pos | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Dist | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 4 |

Step 2. Construct S−distance Lists

1 | 9 | 3 6 |

2 | 10 | 4 7 |

3 | 5 8 11 |

4 | 12 |

Step 3. Sort all type S substrings

Original

| 12 | 2 5 8 |

Sort according to list 1

| 12 | 8 | 5 2 |

Sort according to list 2

| 12 | 8 | 5 2 |

Sort according to list 3

| 12 | 8 | 5 2 |

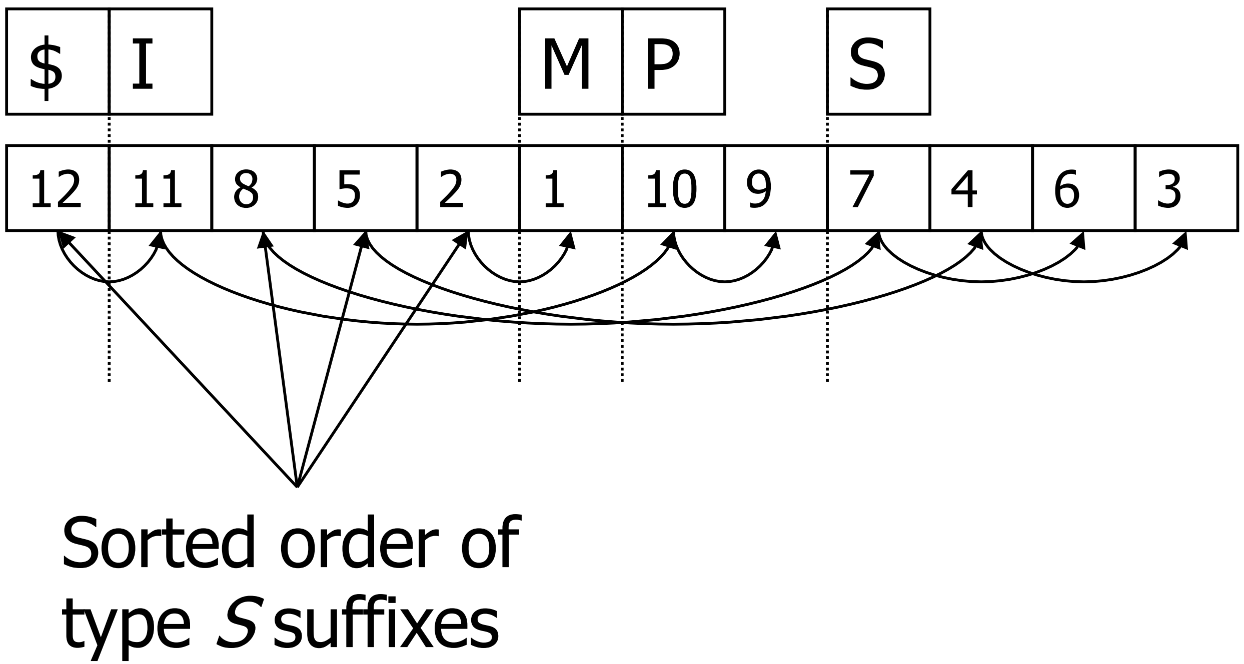
Sort according to list 4

| 12 | 8 | 5 2 |

**Fig. 3.** Illustration of the sorting of type $S$ substrings of the string MISSISSIPPI$.

# Construct Suffix Array for all Suffixes

- The first suffix in the suffix array is a type S suffix.
- For $1 \leq i \leq n$, if $T_{SA[i]-1}$ is type $L$, move it to the current front of its bucket



# Run-Time Analysis (Sketch)

- Identify types of suffixes -- O($n$) time.
- Bucket sort type $S$ (or $L$) substrings -- O($n$) time.
- Construct suffix array from sorted type $S$ (or $L$) suffixes -- O($n$) time.

# Conclusion

- Among the first suffix array construction algorithm takes O(*n*) time.
- The algorithm can be easily implemented in 8*n* bytes (plus a few Boolean arrays).
- Equal or less space than most non-linear time algorithm.

# Exercise

- Consider the popular example string S:
- **`bananainpajamas$`**

1. Construct the suffix array of S using the linear time algorithm
2. Then compute the BWT(S)
3. What's the relationship between the suffix array and BWT ?

# Step – Identify the type of each suffix

- `LSLSLSSSLSLSLSL` `L/S`
- `bananainpajamas$`
- `          1`
- `1234567890123456`

# Step – Compute the distance from S

- **LSLSLSSSLSLSLSL**L/S
- **bananainpajamas$**
- **         1111111**
- **1234567890123456**
- **0012121112121212**

# Step – Sort order of chars

```
LSLSLSSSLSLSLSLL/S
bananainpajamas$
          1111111
1234567890123456
0012121112121212
$a      bijmn  ps
1   111   11    1
6246024171335895
```

# Step – Construct m-Lists

- LSLSLSSSLSLSLSL L/S
- bananainpajamas$
- 1111111
- 1234567890123456
- 0012121112121212
- $a bijmn ps
- 1 111 11 1
- 6246024171335895

Scan this once and bucket it according to dist.

# Step – Generate m-Lists

- **List 1**
- **[7],[11],[13],[3,5,8],[9],[15]**
- **List 2**
- **[16],[4,6,10,12,14]**

```
2022222011111111
$a     bijmn  ps
1   111  11    1
6246024171335895
```

# Step – Sort S substrings

**Bucket the S substrings**

**[16],[2,4,6,10,12,14],[7],[8]**

```
         1111111
1234567890123456
0012121112121212
$a     bijmn  ps
1   111  11    1
6246024171335895
```

# Step – Sort S substrings

**Bucket the S substrings**

**[16],[2,4,6,10,12,14],[7],[8]**

**After using List 1:**

**[16],[6],[10],[12],[2,4],[14],[7],[8]**

**List 2 useless. Then?**

- **List 1**
- **[7],[11],[13],[3,5,8],[9],[15]**
- **List 2**
- **[16],[4,6,10,12,14]**

# Step – Sort S substrings

**Bucket the S substrings**

**[16],[2,4,6,10,12,14],[7],[8]**

**After using List 1:**

**[16],[6],[10],[12],[2,4],[14],[7],[8]**

**List 2 useless. Consider 6 before 4:**

**[16],[6],[10],[12],[4],[2],[14],[7],[8]**

- **List 1**
- **[7],[11],[13],[3,5,8],[9],[15]**
- **List 2**
- **[16],[4,6,10,12,14]**

# Step – Generate the Suffix Array

**[16],[6],[10],[12],[4],[2],[14],[7],[8]**

- `$a     bijmn  ps`
- `1   111  11    1`
- `6246024171335895`

- `$a     ins`
- `1 11  1  1`
- `6602424785`

# Step – Generate the Suffix Array

```
$a     bijmn  ps
1   111  11    1
6246024171335895
```

```
$a     in s
1 11  1   1
66024247585
```

# Step – Generate the Suffix Array

```
$a     bijmn  ps
1   111  11    1
6246024171335895

$a     in ps
1 11  1    1
660242475895
```

# Step – Generate the Suffix Array

```
$a     bijmn  ps
1   111  11    1
6246024171335895

$a     ijn ps
1 11  1 1   1
6602424715895
```

# Step – Generate the Suffix Array

```
$a     bijmn  ps
1   111  11    1
6246024171335895
```

```
$a     ijn  ps
1 11  1 1    1
66024247153895
```

type S

# Step – Generate the Suffix Array

```
$a     bijmn  ps
1   111  11    1
6246024171335895
```

```
$a     bijn  ps
1 11  1  1    1
660242417153895
```

# Step – Generate the Suffix Array

```
$a     bijmn  ps
1   111  11    1
6246024171335895
```

```
$a     bijmn  ps
1 11  1  11    1
6602424171353895
```

# Final answer

```
bananainpajamas$
         1111111
1234567890123456

Suffix Array:
1 11  1  11    1
6602424171353895
```

# Final answer

```
▪ bananainpajamas$
▪          1111111
▪ 1234567890123456

▪ Suffix Array:
▪ 1 11  1  11    1
▪ 6602424171353895
```

**What is the BWT(S) ?**

# BWT is easy!

- `bananainpajamas$`
- `         1111111`
- `1234567890123456`

- **`Suffix Array:`**
- `1 11  1  11    1`
- `6602424171353895`
- **`BWT:`**
- `1  1  11 11    1`
- `5591313660242784`

# BWT construction in linear time

```
bananainpajamas$
         1111111
1234567890123456

BWT:
1  1  11 11    1
5591313660242784
snpjnbm$aaaaaina
```