CS 3391
Suffix Array

Hao YUAN

What is suffixes?

Given a string (or text)

$T = t_1\ t_2\ t_3 \dots t_{n-1}\ t_n$

then it has n suffixes,

they are

$Suffix(T,i) = S[i]$

$= t_i\ t_{i+1}\ t_{i+2} \dots t_n$

for

$1 \le i \le n$

T =	innovation
S[1]=	innovation
S[2]=	nnovation
S[3]=	novation
S[4]=	ovation
S[5]=	vation
S[6]=	ation
S[7]=	tion
S[8]=	ion
S[9]=	on
S[10]=	n

Why suffixes?

- Prefix of a string $T = t_1\ t_2\ t_3 \ldots t_{n-1}\ t_n$
  - $\mathrm{Prefix}(T,i) = t_1\ t_2\ t_3 \ldots t_{i-1}\ t_i$
- Tricky (keep in mind please)
  - Any substring (or pattern) of **T**, must be a prefix of some suffix from **T**!
  - Example **T = mississippi,**
    P = ssip

  then **P = Prefix(Suffix(T, 6), 4)**

Exact Pattern Matching

- How do you find the occurrence of a pattern **`P`** in a text **`T`**?
  - Test for each **`i`** whether **`P`** is a prefix of **`Suffix(T,i)`**
- Naïve implementation: **O(`PT`)** time, too slow!
- Knuth-Morris-Pratt (1977, SIAM J. Comput.)
  - Key Point: ignore testing impossible suffixes
  - **O(`P`)** preprocessing **`P`**
    - Calculate **`Next(k)`** : which suffix should try next when the first **`k`** chars of **`P`** are matched in the current suffix?
  - **O(`P + T`)** searchLCAing for any text **`T`**
  - Will be covered in CS 4335

Match Concurrently

- KMP test suffixes in sequential order
  - Why not do the testing ***concurrently***?

Match Concurrently (e.g.)

Example

T= `mississipi`

P= `ssip`

```
mississipi
 ississipi
  ssissipi
   sissipi
    issipi
     ssipi
      sipi
       ipi
        pi
         i
```

Match Concurrently (e.g.)

Example

T= mississippi

P= ssip

matching **s**

~~mississipi~~

~~ississipi~~

ssissipi

sissipi

~~issipi~~

ssipi

sipi

~~ipi~~

~~pi~~

~~i~~

Match Concurrently (e.g.)

Example

T= `mississippi`

P= `ssip`

matching `ss`

~~`mississipi`~~

~~`ississipi`~~

`ssissipi`

~~`sissipi`~~

~~`issipi`~~

`ssipi`

~~`sipi`~~

~~`ipi`~~

~~`pi`~~

~~`i`~~

Match Concurrently (e.g.)

Example

T= mississippi

P= ssip

matching **ssi**

~~mississipi~~
~~ississipi~~
ssissipi
~~sissipi~~
~~issipi~~
ssipi
~~sipi~~
~~ipi~~
~~pi~~
~~i~~

Match Concurrently (e.g.)

Example

T= **mississippi**

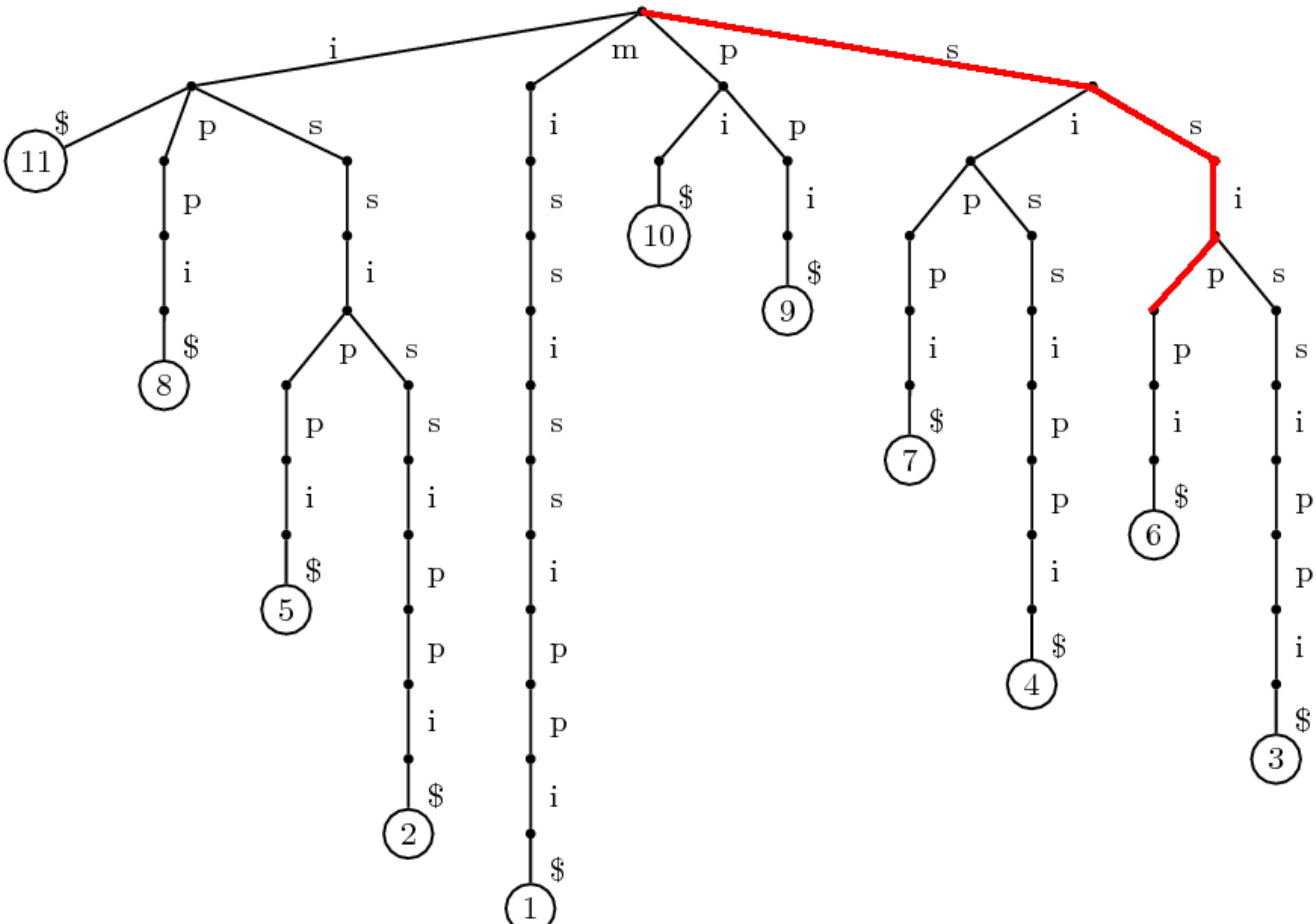
P= **ssip**

matching **ssip**

~~mississipi~~
~~ississipi~~
~~ssissipi~~
~~sissipi~~
~~issipi~~
ssipi
~~sipi~~
~~ipi~~
~~pi~~
~~i~~

Trie (“retrieval”)

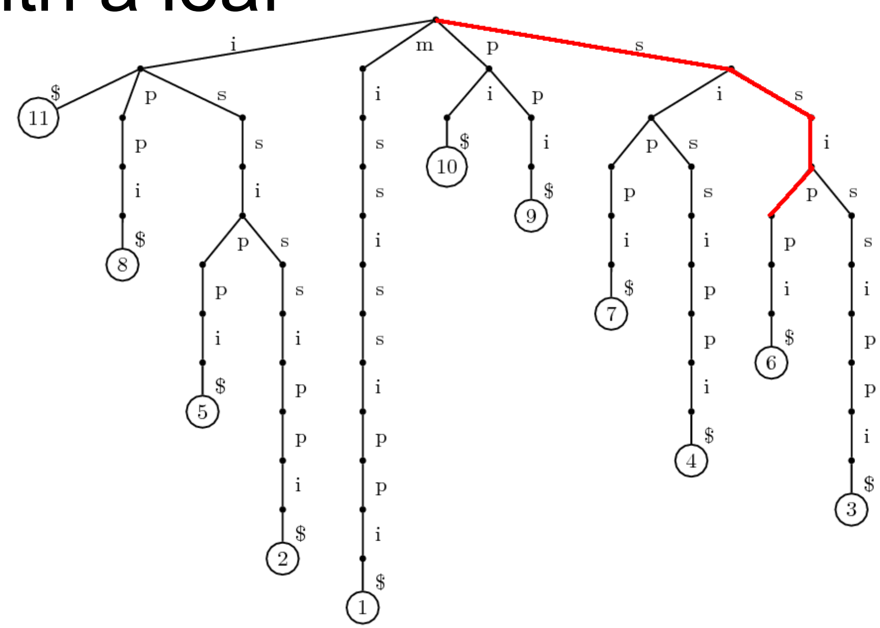
- But we do not have `|T|` CPUs!
- Put all suffixes into a Trie!



i
m
p
s
$
p
s
i
p
i
p
i
s
11
p
s
$
i
p
s
i
i
s
10
$
p
s
i
s
p
s
$
$
8
9
p
s
i
p
i
s
p
s
p
i
i
i
i
s
$
7
p
p
i
$
6
p
5
$
p
p
i
4
$
i
i
2
$
$
3
1
$

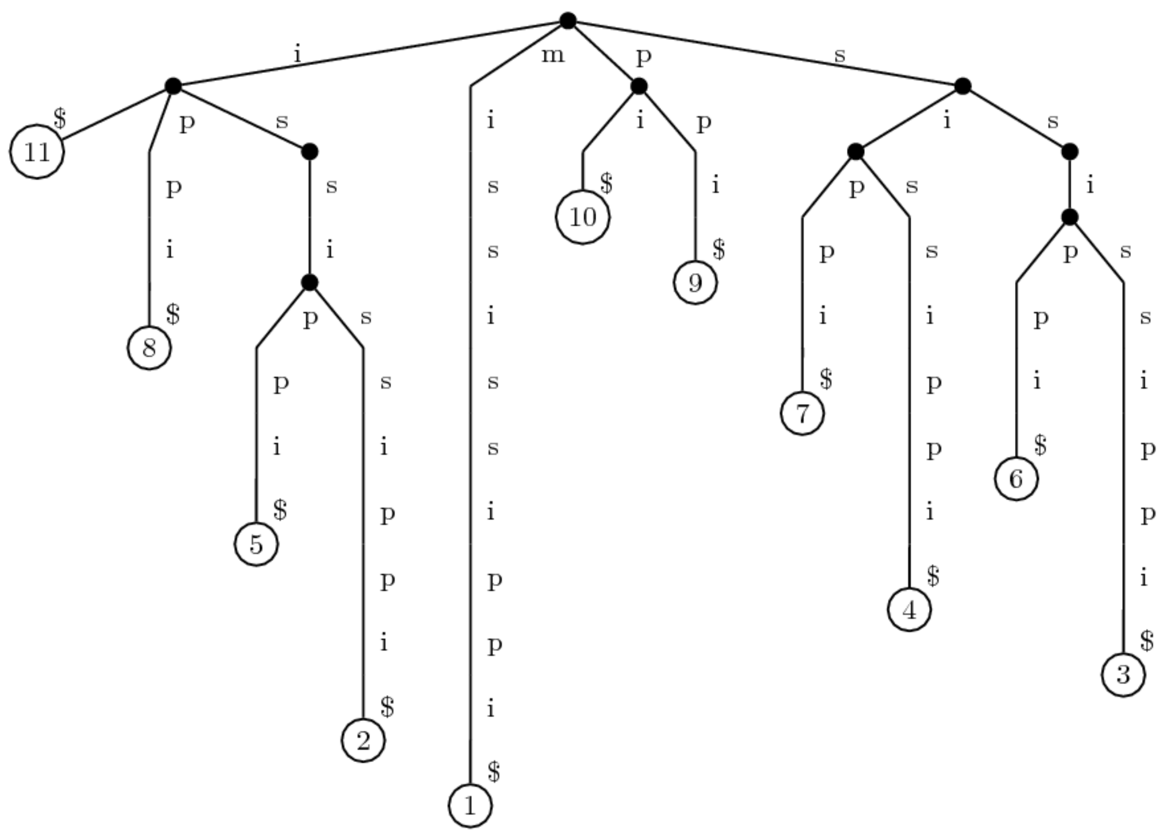
Trie (“retrieval”) (cont.)

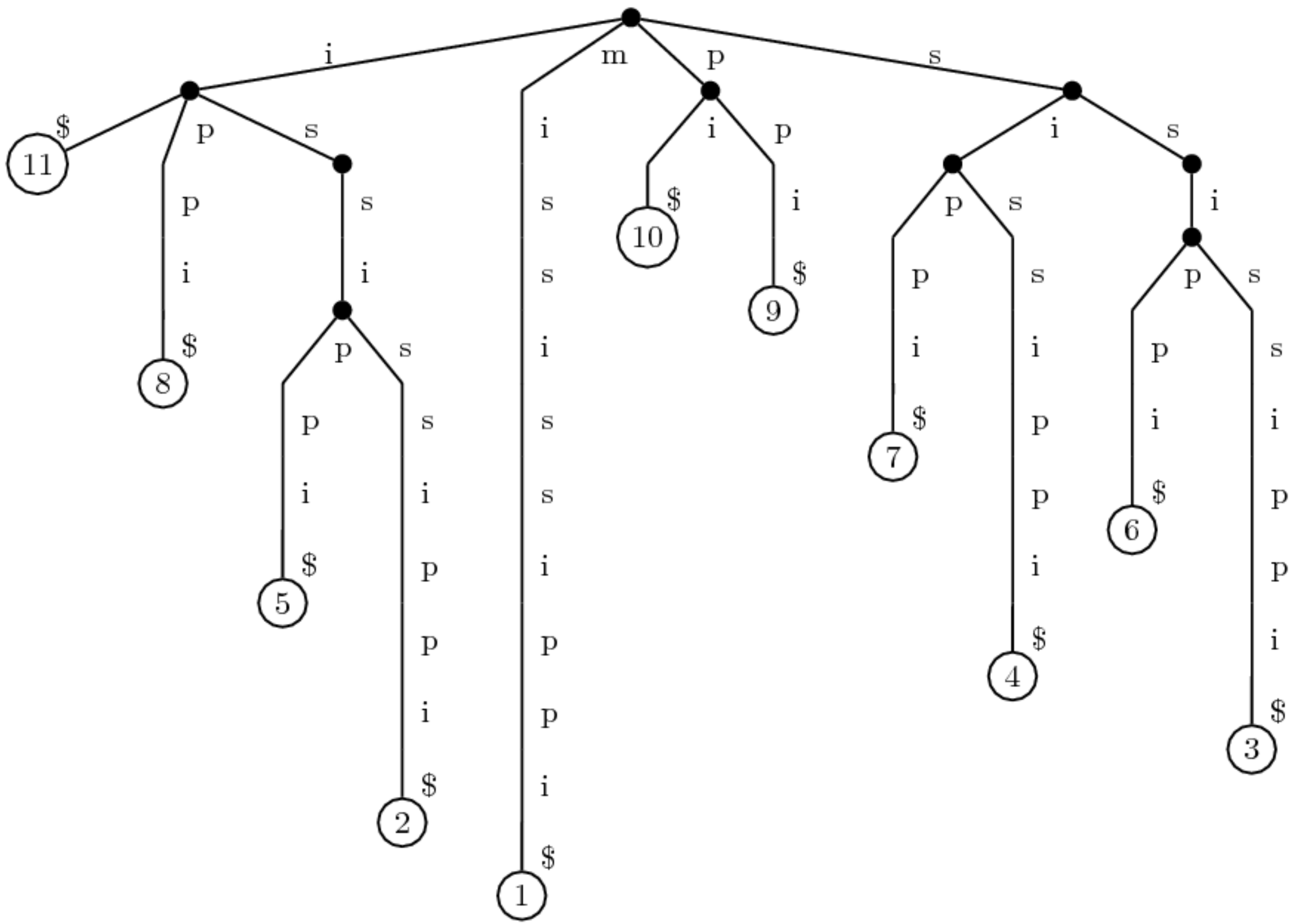
- Put all suffixes into a Trie!
  - Every top-down path starts from root corresponds to a substring
  - Those paths ending with a leaf correspond to suffixes
- Complexity
  - Preprocessing **$O(T^2)$**
  - Matching **$O(P)$**



Suffix Tree

- Trie of all suffixes: too much nodes! **O($\mathtt{T}^2$)**
- Suffix Tree
  - Compact Trie
  - **O($\mathtt{T}$)** nodes





i m p s
$ p s
11
p i $
8
s i
p s
p i $
5
s i p p i $
2
i s s i s s i p p i $
1
i p
$
10
i $
9
i s
p s
p i $
7
s i p p i $
4
i
p s
p i $
6
s i p p i $
3

Suffix Tree

- Trie of all suffixes: too much nodes! **$O(T^2)$**
- Suffix Tree
  - Compact Trie of all suffixes
  - **$O(T)$** nodes
  - Compact representation of substring
    - e.g. `T[3..7] = ` $t_3 t_4 \ldots t_7$
  - **$O(T)$** storage

Constructing Suffix Tree

- The ***Suffix Tree*** can be constructed in ***Linear Time***
  - Assume that the alphabet set is constant size.
  - English Alphabet: {a,b,c,…,z} : 26
  - Language of DNA: {A,T,C,G} : 4
- P. Weiner, 1973
  - ***Linear pattern matching algorithms***
  - the 14th IEEE Annual Symposium on Switching and Automata Theory
- E. M. McCreight, 1976
  - ***A space-economical suffix tree construction algorithm***
  - Journal of ACM, Volume 23 Issue 2
- E. Ukkonen, 1995
  - ***Constructing suffix trees on-line in linear time***
  - Algorithmica, Volume 14 Issue 3

KMP v.s. Suffix Tree

- KMP
  - Preprocess **P**attern in **O(P)** time
  - Search in any **T**ext in **O(P+T)** time
- Suffix Tree
  - Preprocess **T**ext in **O(T)** time
  - Search for any **P**attern in **O(P)** time

Match Char by Char

Example

T= mississippi

P= ssip

```
S[ 1]= mississippi
S[ 2]=  ississippi
S[ 3]=   ssissippi
S[ 4]=    sissippi
S[ 5]=     issippi
S[ 6]=      ssippi
S[ 7]=       sippi
S[ 8]=        ippi
S[ 9]=         ppi
S[10]=          pi
S[11]=           i
```

Match Char by Char

Example

T= mississippi

P= ssip

Sort Suffixes First!

```
S[11]= i
S[ 8]= ippi
S[ 5]= issippi
S[ 2]= ississippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 7]= sippi
S[ 4]= sissippi
S[ 6]= ssippi
S[ 3]= ssissippi
```

Match Char by Char

Suffix Array

T= **mississippi**

```
S 11 = i
S  8 = ippi
S  5 = issippi
S  2 = ississippi
S  1 = mississippi
S 10 = pi
S  9 = ppi
S  7 = sippi
S  4 = sissippi
S  6 = ssippi
S  3 = ssissippi
```

Match Char by Char

Suffix Array

T= mississippi

Search **s**

Binary Search for Lower Bound

Binary Search for Upper Bound

S	11	=	i	
S	8	=	ippi	
S	5	=	issippi	
S	2	=	ississippi	
S	1	=	mississippi	
S	10	=	pi	
S	9	=	ppi	
S	7	=	sippi	
S	4	=	sissippi	
S	6	=	ssippi	
S	3	=	ssissippi	

Match Char by Char

Suffix Array

T= mississippi

Search **ss**

```
S 11 = i
S  8 = ippi
S  5 = issippi
S  2 = ississippi
S  1 = mississippi
S 10 = pi
S  9 = ppi
S  7 = sippi
S  4 = sissippi
S  6 = ssippi
S  3 = ssissippi
```

Match Char by Char

Suffix Array

T= mississippi

Search **ssi**

S	11	=	i
S	8	=	ippi
S	5	=	issippi
S	2	=	ississippi
S	1	=	mississippi
S	10	=	pi
S	9	=	ppi
S	7	=	sippi
S	4	=	sissippi
S	6	=	ssippi
S	3	=	ssissippi

Match Char by Char

Suffix Array

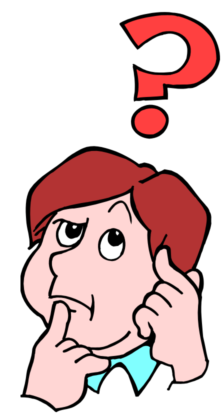
T= mississippi

Search **ssip**

```
S 11 = i
S  8 = ippi
S  5 = issippi
S  2 = ississippi
S  1 = mississippi
S 10 = pi
S  9 = ppi
S  7 = sippi
S  4 = sissippi
S  6 = ssippi
S  3 = ssissippi
```

Suffix Array

- Suffix Array (SA) : ***sorted indexes*** of all suffixes of a string in lexicographical order.
- Given the Suffix Array of **`T`**
  - Find all occurrences of **`P`** by a naïve binary search in **O(`P \log T`)** time
  - Can be done in **O(P)** time (more advanced topic)
- But how to get the Suffix Array?
  - In another words: How to sort suffixes?



Sorting Suffixes

- QSort or other comparison-based method
  - **$O(n^2 \log n)$**
  - Much faster in practice (real world problem)
- Radix Sort
  - **$O(n^2)$**

Efficient Algorithms

- Doubling Algorithm
  - Udi Manber and Gene Myers, ***Suffix arrays: a new method for on-line string searches*** SODA 1990, SIAM J. Comput. 1993
  - $O(n \log n)$
- Skew Algorithm (for integer alphabet)
  - Kärkkäinen, Sanders and Burkhardt, ***Linear Work Suffix Array Construction***, Journal of the ACM, 2006
  - $O(n)$

L-order

- Definition: `S[i]` $\leq_L$ `S[j]`
  - Use the first `L` chars of each suffixes as `key`

- Examples:
  - `ippi` $<_2$ `issippi`
  - `ssissippi` $=_3$ `ssippi`
  - `ssissippi` $>_4$ `ssippi`

Doubling Algorithm

- Sort by 1-order ($\leq_1$)

```
S[ 2]= ississippi
S[ 5]= issippi
S[ 8]= ippi
S[11]= i
S[ 1]= mississippi
S[ 9]= ppi
S[10]= pi
S[ 3]= ssissippi
S[ 4]= sissippi
S[ 6]= ssippi
S[ 7]= sippi
```

Doubling Algorithm

- Sort by 1-order ($\leq_1$)
- Sort by 2-order ($\leq_2$)

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Doubling Algorithm

- Sort by 1-order ($\leq_1$)
- Sort by 2-order ($\leq_2$)

Then…

- Sort by 3-order ($\leq_3$) ?
  - No! This is what ***Radix Sort*** do for general strings.
  - But we are sorting suffixes!
- Sort by 4-order ($\leq_4$) directly

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Extend 2-order to 4-order

- To compare
  - S[3]=ssissippi
  - S[6]=ssippi
- ss is sippi

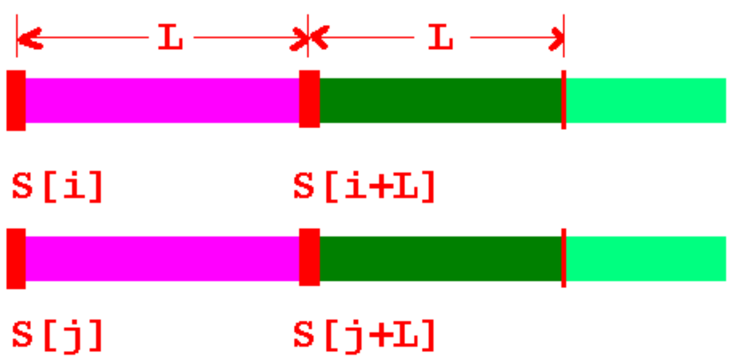
  ss ip pi
- issippi$>_2$ippi

  from S[5]$>_2$S[8]

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Extend L-order to 2L-order

- If we have the L-order, then 2L-order could be obtain by
  - S[i] $<_L$ S[j] → S[i] $<_{2L}$ S[j]
  - S[i] $>_L$ S[j] → S[i] $>_{2L}$ S[j]
  - S[i] $=_L$ S[j]
    - S[i+L] $<_L$ S[j+L] → S[i] $<_{2L}$ S[j]
    - S[i+L] $>_L$ S[j+L] → S[i] $>_{2L}$ S[j]



Complexity of Doubling Algorithm

- Doubling Algorithm
  - `L = 1,2,4,8,16,32,…` until exceeds `n`
  - `O( \log n )` phases, each phase `O( n )`
  - Total Complexity
    - Time `O( n \log n )`
    - Space `O( n )`
- Optimal for general alphabet set
- Expected Running Time for Uniform Alphabet
  - `O( n )`

Skew Algorithm

- Integer Alphabet: $\Sigma=\{1,2,3,\ldots,n\}$
- Basic Idea
  - Group 3 consecutive chars into a new one.
    - `mis  sis  sip  pi$`
    - Σ'= { 'mis', 'sis', 'sip', 'pi$' }
  - Sort the new alphabet set Σ'
  - Sort `S[i]` for { i mod 3 = 1 }
  - Sort `S[i]` for { i mod 3 = 2 }
  - Merge the result
- $T(n) = O(n) + 2T(n/3) = O(n)$
- Easy to implement! (See the JACM paper if interested)