Linear time algorithm for construction of suffix arrays (suffix sorting)

CS176 Lecture 5

Suffix arrays

$T = \text{bississippi\$}$

$T[A_T[i]..\lvert T\rvert]$	$A_T[i]$
\$	12
bississippi\$	1
i\$	11
ippi\$	8
issippi\$	5
ississippi\$	2
pi\$	10
ppi\$	9
sippi\$	7
sissippi\$	4
ssippi\$	6
ssissippi\$	3

Suffix array of a string T (A_T) :
$A_T[i]$ is the start position in T of the i-th lexicographically smallest suffix of T

P=is

	$T[A_T[i]..\lvert T\rvert]$		$A_T[i]$
L=1	$		12
	bississippi$		1
	i$		11
	ippi$		8
	issippi$		5
M=ceil((L+R)/2)	ississippi$	$=T[A_T[M]..\lvert S\rvert]$	2
	pi$		10
	ppi$		9
	sippi$		7
	sissippi$		4
	ssippi$		6
R=$\lvert S\rvert$	ssissippi$		3

Binary search for the first position in A_T that includes P as a prefix:
If P matches a prefix of $T[A_T[M]..|T|]$ or if it is smaller than $T[A_T[M]..|T|]$
Go to the **L** direction; otherwise, go to the **R** direction.

$P = is$

	$T[A_T[i]..\|T\|]$	$A_T[i]$
L	\$	12
	bississippi\$	1
M	i\$	11
	ippi\$	8
	issippi\$	5
R	ississippi\$	2
	pi\$	10
	ppi\$	9
	sippi\$	7
	sissippi\$	4
	ssippi\$	6
	ssissippi\$	3

P=is

	$T[A_T[i]..\|T\|]$	$A_T[i]$
	$	12
	bississippi$	1
L	i$	11
	ippi$	8
M	issippi$	5
R	ississippi$	2
	pi$	10
	ppi$	9
	sippi$	7
	sissippi$	4
	ssippi$	6
	ssissippi$	3

P=is

	$T[A_T[i]..\|T\|]$	$A_T[i]$
	$	12
	bississippi$	1
L	i$	11
M	ippi$	8
R	issippi$	5
	ississippi$	2
	pi$	10
	ppi$	9
	sippi$	7
	sissippi$	4
	ssippi$	6
	ssissippi$	3

P=is

	$T[A_T[i]..\|T\|]$	$A_T[i]$
	$	12
	bississippi$	1
L	i$	11
R=M	ippi$	8
	issippi$	5
	ississippi$	2
	pi$	10
	ppi$	9
	sippi$	7
	sissippi$	4
	ssippi$	6
	ssissippi$	3

P=is

R=L=M $\longrightarrow$

$T[A_T[i]..\|T\|]$	$A_T[i]$
$	12
bississippi$	1
i$	11
ippi$	8
issippi$	5
ississippi$	2
pi$	10
ppi$	9
sippi$	7
sissippi$	4
ssippi$	6
ssissippi$	3

For k matches of P in T - search running time is $O(\log(|T|)|P| + k)$

-- In practice, the actual number of character comparisons in each iteration of the binary each may be small, leading to an effective $O(\log(|T|) + |P| + k)$ running time

-- Theoretically a $O(\log(|T|) + |P| + k)$ can be obtained using the LCP-based speedup

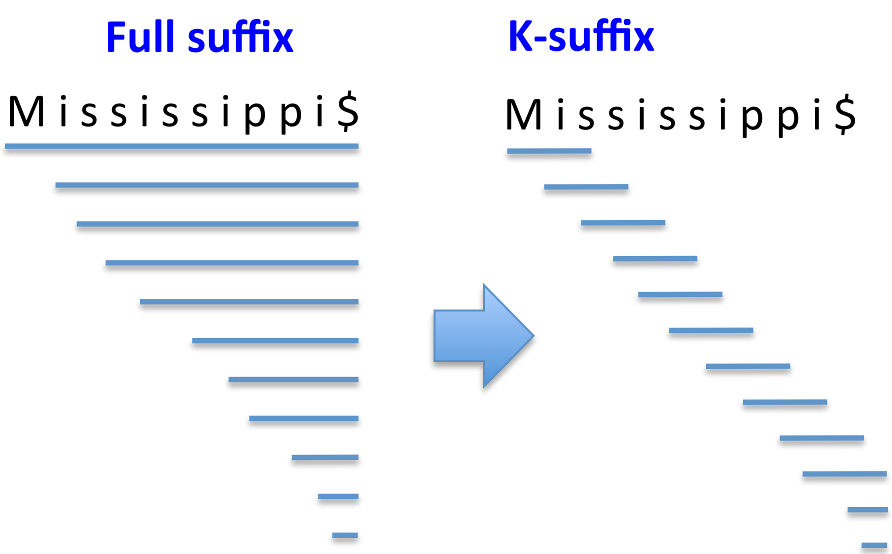
- Constructing a suffix array:
  - Naïve merge sort : O(n^2 log(n))
  - Naïve Radix sort : O(n^2)
  - Suffix tree: O(n) but with large space requirement

- Direct suffix sorting with linear complexity was an **open problem** for a while. Resolved ~2005

- **Today:** KS algorithm (Kärkkäinen and Sanders 2006). A (no so naïve) combination of merge sort and radix sort, leading to linear time/space suffix sorting.

- Intuition (#1):

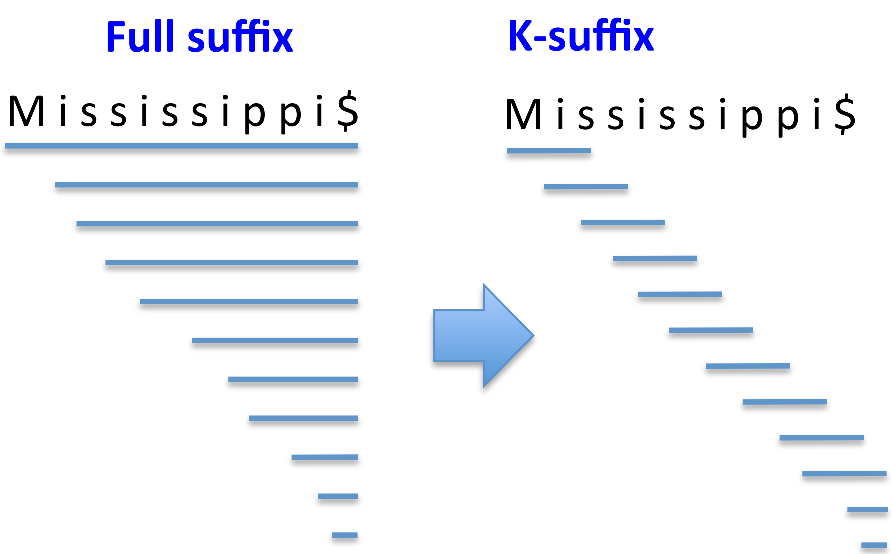
It would save a lot of time if we could get away with considering **only** the first *k* characters of each suffix (why?) Radix sort: O(k|T|) = O(|T|)



Full suffix	K-suffix
mississippi$	mis
ississippi$	iss
ssissippi$	ssi
sissippi$	sis
issippi$	iss
ssippi$	ssi
sippi$	sip
ippi$	ipp
ppi$	ppi
pi$	pi$
i$	i$
$	$

- Intuition (#1):

It would save a lot of time if we could get away with considering **only** the first *k* characters of each suffix (why?)   Radix sort: O(k|T|) = O(|T|)

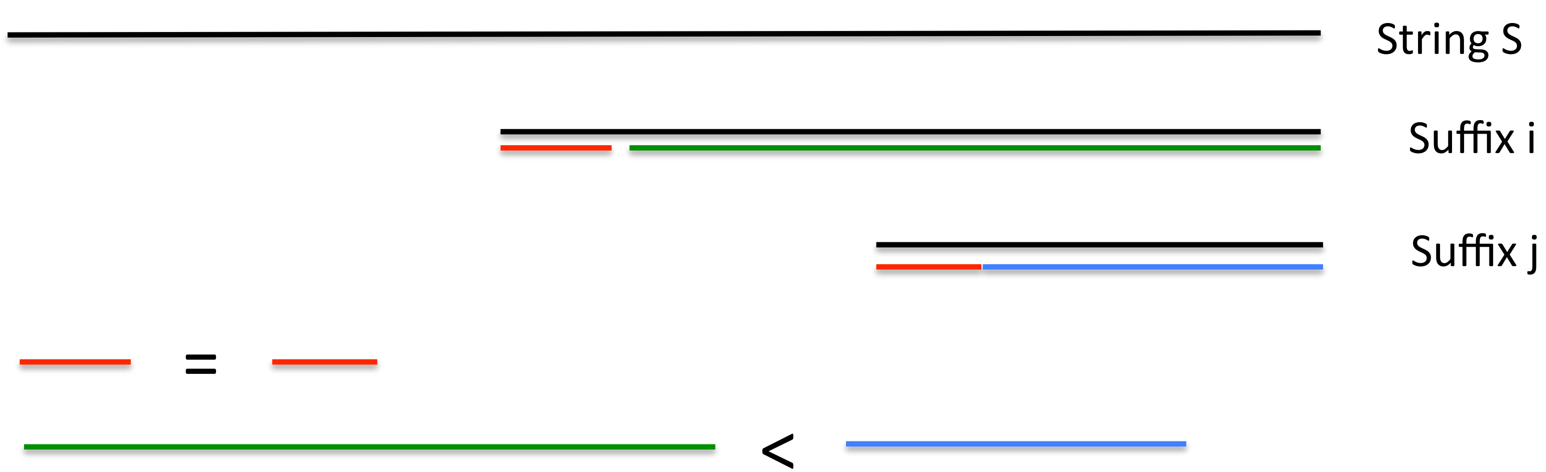


How to handle ambiguities?

Full suffix	K-suffix
mississippi$	mis
ississippi$	iss
ssissippi$	ssi
sissippi$	sis
issippi$	iss
ssippi$	ssi
sippi$	sip
ippi$	ipp
ppi$	ppi
pi$	pi$
i$	i$
$	$

- # Intuition (#2):

The lexicographic ordering between two suffixes *i*, and *j* that have the same prefix depends on the rank of their subsequent suffixes (i.e., indices i+1, i+2… and j+1, j+2…)



$$\Rightarrow S[i \ldots |S|] < S[j \ldots |S|]$$

- Intuition (#2):

The lexicographic ordering between two suffixes *i*, and *j* that have the same prefix depends on the rank of their subsequent suffixes (i.e., indices i+1, i+2… and j+1, j+2…)

mississippi$	mis
ississippi$	**iss**
ssissippi$	ssi
sissippi$	sis
issippi$	**iss**
ssippi$	ssi
sippi$	sip
ippi$	ipp
ppi$	ppi
pi$	pi$
i$	i$
$	$

ississippi$ Suffix i (i=2)

issippi$ Suffix j (j=5)

iss = iss

issippi$ > ippi$

=> S[i …|S|] > S[j … |S|]

- Intuition (#2):

The lexicographic ordering between two suffixes *i*, and *j* that have the same prefix depends on the rank of their subsequent suffixes (i.e., indices i+1, i+2… and j+1, j+2…)

→ Can be used for tie breaking between k-suffixes:

mississippi$	mis
ississippi$	**iss**
ssissippi$	ssi
sissippi$	sis
issippi$	**iss**
ssippi$	ssi
sippi$	sip
ippi$	ipp
ppi$	ppi
pi$	pi$
i$	i$
$	$

iss, ssi, sis K-Suffix i..i+2 (i=2)

iss, ssi, sip K-Suffix j..j+2 (j=5)

iss = iss, ssi=ssi

Ks (j) = Ks (i)
Ks (j+1) = Ks (i+1)

sis > sip

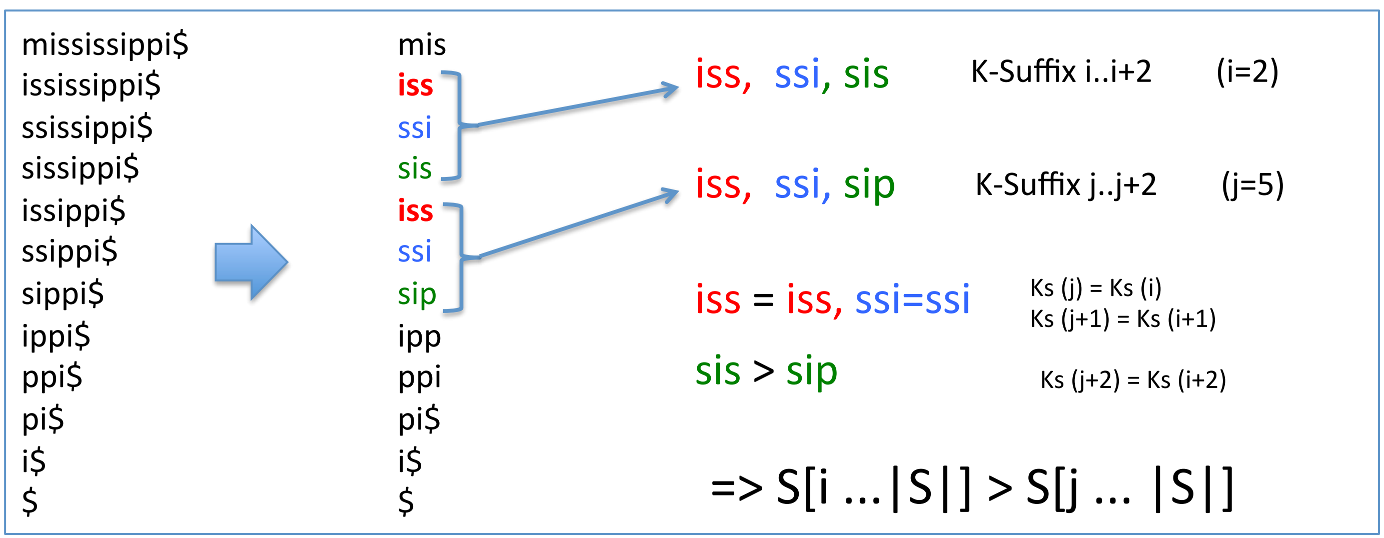
Ks (j+2) = Ks (i+2)

=> S[i …|S|] > S[j … |S|]

First attempt at linear complexity:

- K-suffixes can be sorted in **linear time**
  (but with possible ties)

- Tie breaking between K-suffixes can be done by looking at the lexicographic ranks of their subsequent *K-1* K-suffixes
  (unless the subsequent suffixes are also tied, in which case we will have to continue our search for tie break recursively)



- First attempt at linear complexity (ALG1):

1. Sort the suffixes by the first k(=3) characters using Radix sorting
(this will only take linear time)

2. If there are no ambiguities, we are done!

Otherwise:

3. Replace each character by the rank of the respective suffix (as computed in step 1)

4. Repeat the process (i.e., go to step #1) with the new string

- First attempt at linear complexity:

Full suffix	**K-suffix**	**Lex-Rank**
mississippi$	mis	6
ississippi$	iss	4
ssissippi$	ssi	11
sissippi$	sis	10
issippi$	iss	4
ssippi$	ssi	11
sippi$	sip	9
ippi$	ipp	3
ppi$	ppi	8
pi$	pi$	7
i$	i$	2
$	$	1

- First attempt at linear complexity:

Full suffix	K-suffix	Lex-Rank	New string
mississippi$	mis	6	
ississippi$	iss	4	
ssissippi$	ssi	11	
sissippi$	sis	10	
issippi$	iss	4	
ssippi$	ssi	11	
sippi$	sip	9	6, 4, 11, 10, 4, 11, 9, 3, 8, 7, 2, 1
ippi$	ipp	3	
ppi$	ppi	8	
pi$	pi$	7	
i$	i$	2	
$	$	1	

- First attempt at linear complexity:

Full suffix	K-suffix	Lex-Rank		A_T
6, 4, 11, 10, 4, 11, 9, 3, 8, 7, 2, 1	6, 4, 11	6	mississippi$	12
4, 11, 10, 4, 11, 9, 3, 8, 7, 2, 1	4, 11, 10	5	ississippi$	11
11, 10, 4, 11, 9, 3, 8, 7, 2, 1	11, 10, 4	12	ssissippi$	8
10, 4, 11, 9, 3, 8, 7, 2, 1	10, 4, 11	10	sissippi$	5
4, 11, 9, 3, 8, 7, 2, 1	4, 11, 9	4	issippi$	2
11, 9, 3, 8, 7, 2, 1	11, 9, 3	11	ssippi$	1
9, 3, 8, 7, 2, 1	9, 3, 8	9	sippi$	10
3, 8, 7, 2, 1	3, 8, 7	3	ippi$	9
8, 7, 2, 1	8, 7, 2	8	ppi$	7
7, 2, 1	7, 2, 1	7	pi$	4
2, 1	2, 1	2	i$	6
1	1	1	$	3

No ambiguities!

- First attempt at linear complexity:

- Running time?
  - Each iteration takes O(n)
  - #Iterations: O(n)
  - O(n^2) time …
    (how does the worst case look like?)
    (how would an “easy” case look like?)

- Intuition (#3):

Divide and conquer: solve a smaller instance of the problem and extend it to a full solution in linear time. Apply it in a recursive manner.

- Intuition (#3):

Divide and conquer: solve a smaller instance of the problem and extend it to a full solution in linear time. Apply it in a recursive manner.

Let T be the running time and q>1 a constant

If $T(n) = \underbrace{T(n/q)}_{\text{Smaller instance (divide)}} + \underbrace{O(n)}_{\text{Extension to full solution (merge)}}$ and $T(1)=1$

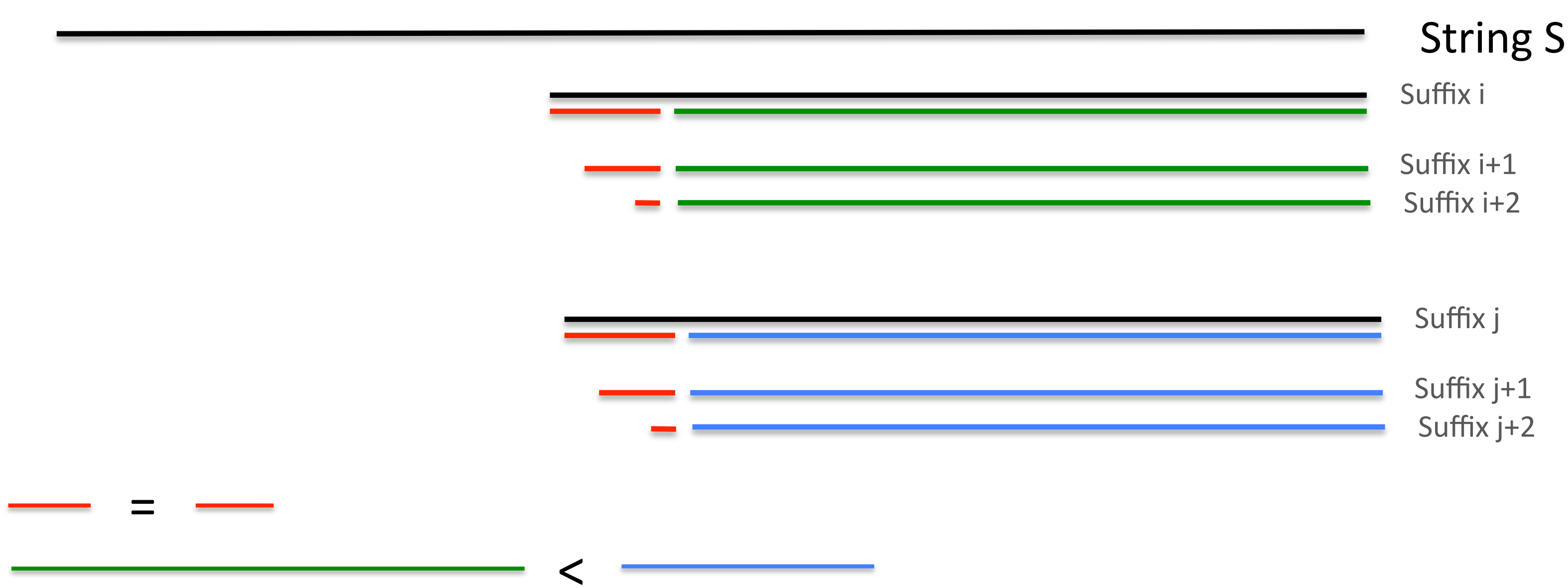
Then: $$T(n) = n\sum_{i=0}^{n}\frac{1}{q^i} = O(n)$$

- Intuition (#3): Divide and conquer

For the purpose of tie breaking, the k-suffix representation is somewhat **redundant**. To resolve the lexicographic ordering between two suffixes i, and j that have the same k-prefix it is enough (for k=3) to know the “true” rank of **only one** representative from {i+1, i+2} and **one** from {j+1, j+2}

- Intuition (#3): Divide and conquer

For the purpose of tie breaking, the k-suffix representation is somewhat **redundant**. To resolve the lexicographic ordering between two suffixes i, and j that have the same k-prefix it is enough (for k=3) to know the "true" rank of **only one** representative from {i+1, i+2} and **one** from {j+1, j+2}



=> S[i ...|S|] < S[j ... |S|]

- Divide and conquer strategy:
  Knowing the “true” ranks of two thirds of the suffixes (evenly distributed) is sufficient to annotate the remaining third!

Split: For every triplet of suffixes (at indices [i, i+1, i+2]) take only two of them (total of 2/3 of suffixes) and sort as in ALG1.

Merge: Use the sorting results to add the remaining third of the suffixes and output the joint sorted list

- Second (and last) attempt at linear complexity:

```
KS-algorithm(string S)
1. Divide the suffixes into two groups:
   Index group “A”: {i}such that i mod 3 ≠ 0
   Index group “B”: {i} such that i mod 3 = 0
2. Sort the suffixes in group A using ALG2
3. Merge the suffixes in group B into the
   sorted list of group A using ALG3
```

KS-algorithm(string **S)**
1. Divide the suffixes into two groups:
   Index group "A": {i}such that $i \bmod 3 \neq 0$
   Index group "B": {i} such that $i \bmod 3 = 0$
2. Sort the suffixes in group A using **ALG2**
3. Merge the suffixes in group B into the sorted list of group A using **ALG3**

ALG2 (string **S,** group of suffix indices **A)**

1. Sort the suffixes in the group by the first k(=3)characters using Radix sorting
2. If there are no ambiguities: return the sorted order
3. Otherwise:
   1. Generate a modified string *S′* by substituting each suffix in A with its rank (based on the first 3 characters)
   2. Recursively apply the **KS** algorithm on the modified string *S′* and return the results

KS-algorithm(string **S**)
1. Divide the suffixes into two groups:
   Index group "A": {i}such that i mod 3 ≠ 0
   Index group "B": {i} such that i mod 3 = 0
2. Sort the suffixes in group A using **ALG2**
3. Merge the suffixes in group B into the sorted list of group A using **ALG3**

ALG3 (string S, rank-sorted group **A**, un-sorted group **B)**

1. Represent each index i in B by the pair <S[i],rank[i+1]> // the *rank* information is taken from **A**
2. Radix sort the array of pairs // Now the suffixes in **B** are sorted as well
3. Merge A and B // Simply traverse **A** and **B** in parallel

   To determine the relative order between two suffixes i in A and j in B:

   1. If S[i]≠S[j], order them accordingly//Easy case: First character is different
   2. Otherwise: // Consider the immediately subsequent suffixes
      1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by definition j+1 must be in A; so we are using two "comparable" ranks
      2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and <S[j+1],rank[j+2]>

```
KS-algorithm(string S)
1. Divide the suffixes into two groups:
   Index group "A": {i}such that i mod 3 ≠ 0
   Index group "B": {i} such that i mod 3 = 0
2. Sort the suffixes in group A using ALG2
3. Merge the suffixes in group B into the sorted
   list of group A using ALG3
```

- Time:
  - Step 2: $T(n) = \underbrace{T(2n/3)}^{\text{Recursion}} + \overbrace{O(n)}^{\text{Radix sort}}$

    $\Rightarrow T(n) = O(n)$
  - Step 3: $O(n)$ [for radix sort of group B and merge of groups A and B]
  - Overall: **$O(n)$**

```
KS-algorithm(string S)
1. Divide the suffixes into two groups:
   Index group "A": {i}such that i mod 3 ≠ 0
   Index group "B": {i} such that i mod 3 = 0
2. Sort the suffixes in group A using ALG2
3. Merge the suffixes in group B into the sorted
   list of group A using ALG3
```

S= mississippi$

Index	Suffix	Group
1	mississippi$	A
2	ississippi$	A
3	ssissippi$	B
4	sissippi$	A
5	issippi$	A
6	ssippi$	B
7	sippi$	A
8	ippi$	A
9	ppi$	B
10	pi$	A
11	i$	A
12	$	B

```
ALG2 (string S, group of suffix indices A)
1. Sort the suffixes in the group by the first
   k(=3)characters using Radix sorting
2. If there are no ambiguities: return the sorted
   order
3. Otherwise:
   1. Generate a modified string S by substituting each
      suffix in A with its rank (based on the first 3
      characters)
   2. Return KS(S)
```

S= mississippi$
A= {1,2,4,5,7,8,10,11}

Group A suffixes	K-suffix	Lex-Rank
mississippi$	mis	5
ississippi$	iss	3
sissippi$	sis	8
issippi$	iss	3
sippi$	sip	7
ippi$	ipp	2
pi$	pi$	6
i$	i$	1

New string

5 3 8 3 7 2 6 1

```
KS-algorithm(string S)
1. Divide the suffixes into two groups:
   Index group "A": {i}such that i mod 3 ≠ 0
   Index group "B": {i} such that i mod 3 = 0
2. Sort the suffixes in group A using ALG2
3. Merge the suffixes in group B into the sorted
   list of group A using ALG3
```

S= 5 3 8 4 7 2 6 1

Index	Suffix	Group
1	5 3 8 3 7 2 6 1	A
2	3 8 3 7 2 6 1	A
3	8 3 7 2 6 1	B
4	3 7 2 6 1	A
5	7 2 6 1	A
6	2 6 1	B
7	6 1	A
8	1	A

ALG2 (string **S**, group of suffix indices **A)**

1. Sort the suffixes in the group by the first k(=3) characters using Radix sorting
2. If there are no ambiguities: return the sorted order
3. Otherwise:
   1. Generate a modified string S by substituting each suffix in A with its rank (based on the first 3 characters)
   2. Return **KS**(S)

S= 5 3 8 4 7 2 6 1
A= {1,2,4,5,7,8}

Group A suffixes	K-suffix	Lex-Rank
5 3 8 3 7 2 6 1	5 3 8	4
3 8 3 7 2 6 1	3 8 3	3
3 7 2 6 1	3 7 2	2
7 2 6 1	7 2 6	6
6 1	6 1	5
1	1	1

No ambiguities!

```
KS-algorithm(string S)
1. Divide the suffixes into two groups:
   Index group "A": {i}such that i mod 3 ≠ 0
   Index group "B": {i} such that i mod 3 = 0
2. Sort the suffixes in group A using ALG2
3. Merge the suffixes in group B into the sorted
   list of group A using ALG3
```

S= 5 3 8 4 7 2 6 1

Index	Suffix	Group	Rank (A)
1	5 3 8 3 7 2 6 1	A	4
2	3 8 3 7 2 6 1	A	3
3	8 3 7 2 6 1	B	--
4	3 7 2 6 1	A	2
5	7 2 6 1	A	6
6	2 6 1	B	--
7	6 1	A	5
8	1	A	1

```
ALG3 (string S, rank-sorted group A, un-sorted group B)
1. Represent each index i in B by the pair
   <S[i],rank[i+1]> // the rank information is taken from A
2. Radix sort the array of pairs // Now the suffixes in B
   are sorted as well

3. Merge A and B // Simply traverse A and B in parallel
   To determine the relative order between two suffixes
   i in A and j in B:
  1. If S[i]≠S[j], order them accordingly//Easy case: First character is
     different
  2. Otherwise: // Consider the immediately subsequent suffixes
     1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by
        definition j+1 must be in A; so we are using two "comparable" ranks
     2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and
        <S[j+1],rank[j+2]>
```

S= 5 3 8 4 7 2 6 1
A= {1,2,4,5,7,8}
B= {3,6}

Index	Suffix	Group	Rank (A)	Pair representation of B	Order (B)
1	5 3 8 3 7 2 6 1	A	4	--	--
2	3 8 3 7 2 6 1	A	3	--	--
3	8 3 7 2 6 1	B	--	<8, 2>	2
4	3 7 2 6 1	A	2	--	--
5	7 2 6 1	A	6	--	--
6	2 6 1	B	--	<2, 5>	1
7	6 1	A	5	--	--
8	1	A	1	--	--

```
ALG3 (string S, rank-sorted group A, un-sorted group B)
1. Represent each index i in B by the pair
   <S[i],rank[i+1]> // the rank information is taken from A
2. Radix sort the array of pairs // Now the suffixes in B
   are sorted as well

3. Merge A and B // Simply traverse A and B in parallel
   To determine the relative order between two suffixes
   i in A and j in B:
    1. If S[i]≠S[j], order them accordingly//Easy case: First character is
       different
    2. Otherwise: // Consider the immediately subsequent suffixes
        1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by
           definition j+1 must be in A; so we are using two "comparable" ranks
        2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and
           <S[j+1],rank[j+2]>
```

S= 5 3 8 4 7 2 6 1

A= {1,2,4,5,7,8}

B= {3,6}

Sorted A: {1, 37261, 3837261, 53837261, 61, 7261}

Sorted B: {261, 837261}

All cases are easy (can be determined by first character, as in line 3.1 in code)

Sorted: {1, 261, 37261, 3837261, 53837261, 61, 7261, 837261}

```
ALG3 (string S, rank-sorted group A, un-sorted group B)
1. Represent each index i in B by the pair
   <S[i],rank[i+1]> // the rank information is taken from A
2. Radix sort the array of pairs // Now the suffixes in B
   are sorted as well

3. Merge A and B // Simply traverse A and B in parallel
   To determine the relative order between two suffixes
   i in A and j in B:
   1. If S[i]≠S[j], order them accordingly//Easy case: First character is
      different
   2. Otherwise: // Consider the immediately subsequent suffixes
      1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by
         definition j+1 must be in A; so we are using two "comparable" ranks
      2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and
         <S[j+1],rank[j+2]>
```

S= 5 3 8 4 7 2 6 1

A= {1,2,4,5,7,8}

B= {3,6}

Index	Suffix	Group	Rank (both sets)
1	5 3 8 3 7 2 6 1	A	5
2	3 8 3 7 2 6 1	A	4
3	8 3 7 2 6 1	B	8
4	3 7 2 6 1	A	3
5	7 2 6 1	A	7
6	2 6 1	B	2
7	6 1	A	6
8	1	A	1

```
KS-algorithm(string S)
1. Divide the suffixes into two groups:
   Index group "A": {i}such that i mod 3 ≠ 0
   Index group "B": {i} such that i mod 3 = 0
2. Sort the suffixes in group A using ALG2
3. Merge the suffixes in group B into the sorted
   list of group A using ALG3
```

S= mississippi$

Index	Suffix	Group	Rank (A)
1	mississippi$	A	5
2	ississippi$	A	4
3	ssissippi$	B	--
4	sissippi$	A	8
5	issippi$	A	3
6	ssippi$	B	--
7	sippi$	A	7
8	ippi$	A	2
9	ppi$	B	--
10	pi$	A	6
11	i$	A	1
12	$	B	--

```
ALG3 (string S, rank-sorted group A, un-sorted group B)
1. Represent each index i in B by the pair
   <S[i],rank[i+1]> // the rank information is taken from A
2. Radix sort the array of pairs // Now the suffixes in B
   are sorted as well

3. Merge A and B // Simply traverse A and B in parallel
   To determine the relative order between two suffixes
   i in A and j in B:
   1. If S[i]≠S[j], order them accordingly//Easy case: First character is
      different
   2. Otherwise: // Consider the immediately subsequent suffixes
      1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by
         definition j+1 must be in A; so we are using two "comparable" ranks
      2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and
         <S[j+1],rank[j+2]>
```

S= mississippi$
A= {1,2,4,5,7,8,10,11}
B= {3,6,9,12}

Index	Suffix	Group	Rank (A)	Pair representation of B	Order (B)
1	mississippi$	A	5	--	--
2	ississippi$	A	4	--	--
3	ssissippi$	B	--	<s, 8>	4
4	sissippi$	A	8	--	--
5	issippi$	A	3	--	--
6	ssippi$	B	--	<s, 7>	3
7	sippi$	A	7	--	--
8	ippi$	A	2	--	--
9	ppi$	B	--	<p, 6>	2
10	pi$	A	6	--	--
11	i$	A	1	--	--
12	$	B	--	<$, Ø>	1

```
ALG3 (string S, rank-sorted group A, un-sorted group B)
1. Represent each index i in B by the pair
   <S[i],rank[i+1]> // the rank information is taken from A
2. Radix sort the array of pairs // Now the suffixes in B
   are sorted as well

3. Merge A and B // Simply traverse A and B in parallel
   To determine the relative order between two suffixes
   i in A and j in B:
   1. If S[i]≠S[j], order them accordingly//Easy case: First character is
      different
   2. Otherwise: // Consider the immediately subsequent suffixes
      1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by
         definition j+1 must be in A; so we are using two "comparable" ranks
      2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and
         <S[j+1],rank[j+2]>
```

S= mississippi$
A= {1,2,4,5,7,8,10,11}
B= {3,6,9,12}

Index	Suffix	Group	Rank (A)	Order (B)	Order (Merged)
1	mississippi$	A	→ 5	--	6 (rationale: m<p)
2	ississippi$	A	→ 4	--	5 (rationale: i<p)
3	ssissippi$	B	--	4 ←	12 (rationale: A is exhausted)
4	sissippi$	A	→ 8	--	10 (rank[i+1]=3 < rank[j+1]=7)
5	issippi$	A	→ 3	--	4 (rationale: i<p)
6	ssippi$	B	--	3 ←	11 (rationale: A is exhausted)
7	sippi$	A	→ 7	--	9 (rank[i+1]=2 < rank[j+1]=7)
8	ippi$	A	→ 2	--	3 (rationale: i<p)
9	ppi$	B	--	2 ←	8 (rationale: p<s)
10	pi$	A	→ 6	--	7 (rank[i+1]=1 < rank[j+1]=6)
11	i$	A	→ 1	--	2 (rationale: i<p)
12	$	B	--	1 ←	1 (rationale: $<i)

```
ALG3 (string S, rank-sorted group A, un-sorted group B)
1. Represent each index i in B by the pair
   <S[i],rank[i+1]> // the rank information is taken from A
2. Radix sort the array of pairs // Now the suffixes in B
   are sorted as well

3. Merge A and B // Simply traverse A and B in parallel
   To determine the relative order between two suffixes
   i in A and j in B:
   1. If S[i]≠S[j], order them accordingly//Easy case: First character is
      different
   2. Otherwise: // Consider the immediately subsequent suffixes
      1. If i+1 is in A then compare rank[i+1] with rank[j+1] // Note that by
         definition j+1 must be in A; so we are using two "comparable" ranks
      2. Otherwise, lexicographically compare <S[i+1],rank[i+2]> and
         <S[j+1],rank[j+2]>
```

S= mississippi$

A= {1,2,4,5,7,8,10,11}

B= {3,6,9,12}

Index	Suffix	Group	Rank (A)	Order (B)	Order (Merged)
1	mississippi$	A	5	--	6
2	ississippi$	A	4	--	5
3	ssissippi$	B	--	4	12
4	sissippi$	A	8	--	10
5	issippi$	A	3	--	4
6	ssippi$	B	--	3	11
7	sippi$	A	7	--	9
8	ippi$	A	2	--	3
9	ppi$	B	--	2	8
10	pi$	A	6	--	7
11	i$	A	1	--	2
12	$	B	--	1	1

A_T

12
11
8
5
2
1
10
9
7
4
6
3

- Intuition (#1): It would save a lot of time if we could get away with considering only the first k characters of each suffix

- Intuition (#2): The lexicographic ordering between two suffixes i, and j that have the same prefix depends on the rank of their subsequent suffixes (e.g., i+1, i+2 and j+1, j+2)

- Intuition (#3): The k-suffix representation is **redundant**. To resolve the lexicographic ordering between two suffixes i, and j that have the same prefix it is enough (for k=3) to know the rank of one representative from {i+1, i+2} and one from {j+1, j+2}

- Algorithm (divide and conquer):

  Split: For every triplet {i, i+1, i+2} take only two of them and sort 3-suffixes (total of 2/3 of suffixes); Repeat recursively to resolve ambiguities

  Merge: Use the sorting results to add the remaining third of the suffixes to the sorted list