CS 176 Algorithms for Computational Biology **Fall 2015**

Lecture 5: September 14: Building Suffix Arrays with the KS Algorithm

Lecturer: Nir Yosef *Scribe: Manny Jois, Carolyn Sy*

5.1 Introduction and Motivation

In the previous lectures, we learned about suffix arrays. The suffix array of a string S is an array of indices A such that $A_T[i]$ is the start position in T of the i^{th} lexicographically smallest suffix of T.

Figure 5.1: Example of a suffix array for string T="bississippi"

There are many ways of sorting the suffixes; they are listed below:

1. naive mergesort: $O(n^2log(n))$
2. radix sort: $O(n^2)$
3. suffix tree: $O(n)$, but a large space requirement
4. KS algorithm: $O(n)$, will be covered in today's lecture

Although the construction of the suffix tree is linear time, there is a huge space requirement. In this lecture, we will go over an algorithm called the KS algorithm that will serve as a combination of merge and radix sort, which will runs in linear time as well as take up only linear space.

5.2 Intuition 1

It would save a lot of time to consider only the first k characters of each suffix–having a limited number of characters to compare would make radix sort really fast. For example, for the string "mississippi" we would see the following suffixes and their associated k-suffixes (where k=3).

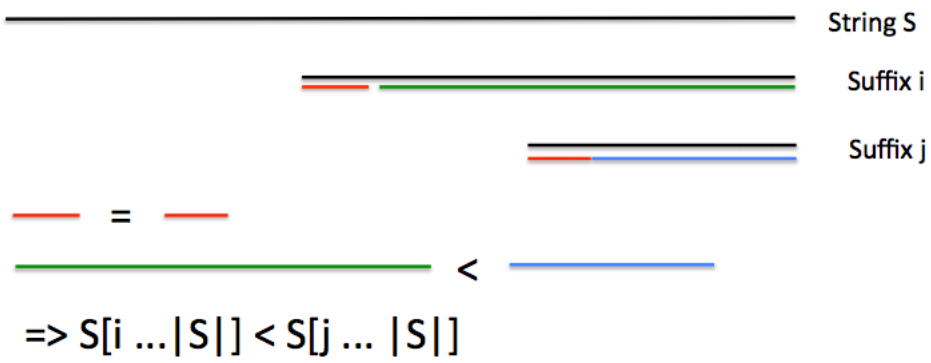
Figure 5.2: Suffixes and K-suffixes for T="mississippi"

Sometimes, just looking at this is enough to determine the lexographical ordering of the suffixes of the string. In that case, using radix sort would be ideal. However, sorting on the basis of the first k characters might lead to ordering ambiguities (i.e when two strings share the same first k characters). To illustrate this, in the example above, there are two k-suffixes that are "iss" and "ssi".

5.3 Intuition 2

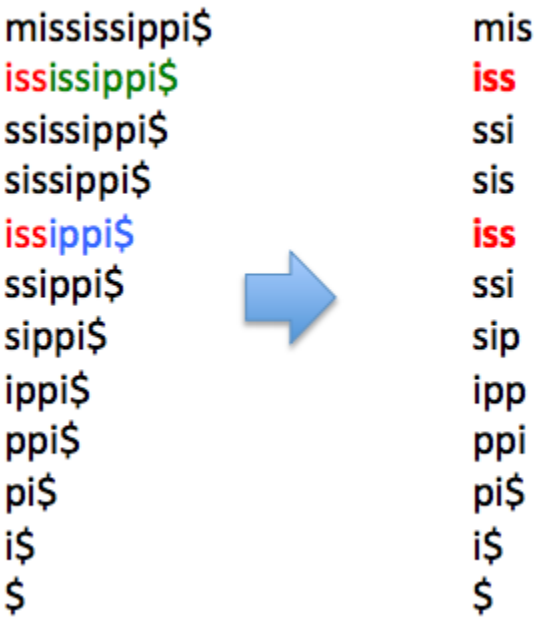
The lexicographic ordering between 2 suffixes i and j that have the same prefix depends on the rank of their subsequent suffixes (i.e. just look at the subsequent letters). We can use this idea for tie-breaking between k-suffixes. If suffix $i + k$ is lexicographically smaller than $j + k$, then i is smaller than j.

Figure 5.3: Visual representation of lexicographic tie-breaking



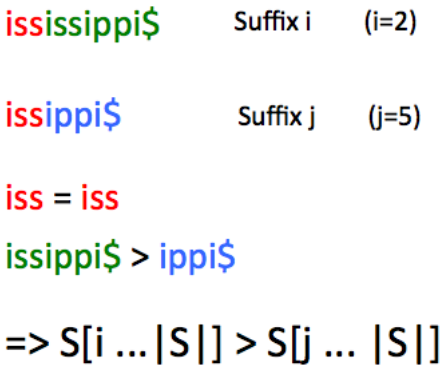
Example: For the string "mississippi", we can use radix sort on the k-suffixes, but that will leave us with some ambiguities:

Figure 5.4: k-suffixes for the string "mississippi"



Notice that there are two 'iss' strings. One thing we see is that we can determine the rank of each 'iss' by comparing the rank of their subsequent suffixes. Essentially, this means that if we have the same k-suffix, we look at the next character to tell us something about the rank; if those two characters are tied, then we look at the next character until one of lexographically different from the other.

Figure 5.5: Tie-breaking on suffixes of "mississippi"



5.4 Algorithm 1 (First attempt at linear time)

5.4.1 Intuition

k-suffixes can be sorted in linear time given constant k (typically $k = 3$), but there will possibly be ties. Tie-breaking can be handled by looking at the lexicographic ranks of the next $k - 1$ k-suffixes (unless the subsequent suffixes are also tied, in which case we have to continue the search for tie breaks recursively).

5.4.2 KS Algorithm

```
1. Use radix sort to sort the suffixes by the first k characters.
2. If there are no ambiguities, end.

Otherwise:
    3. Replace each character by the rank of its respective suffix (computed in step 1).
       Literally replace each character with an integer.
    4. Repeat algorithm (go to step 1) with the newly formed string (a list of integers).
```

5.4.3 Example

For the string "mississippi", we can sort the K-suffixes of the original string, which will yield us a list of lexicographic ranks with one ambiguity (figure 5.6).

We then take the resulting list of lexicographic ranks as our new string and run the algorithm again (figure 5.7). Each k-suffix in this new string now effectively carries $2k-1$ characters of information from the previous string. This time there are no ambiguities.

Figure 5.6: First iteration on "mississippi", with a new string of lexicographic ranks resulting

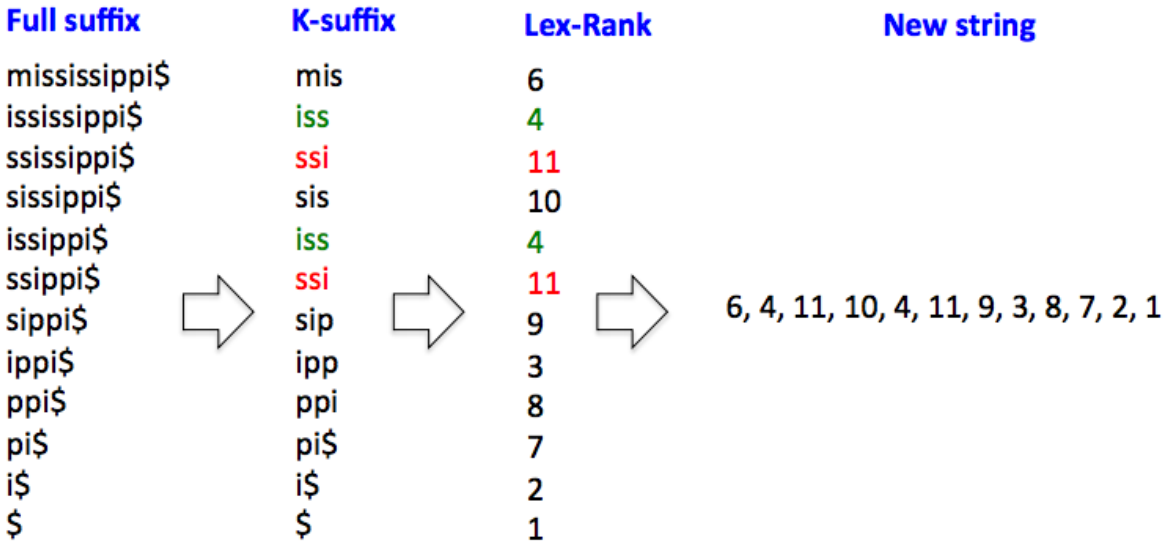
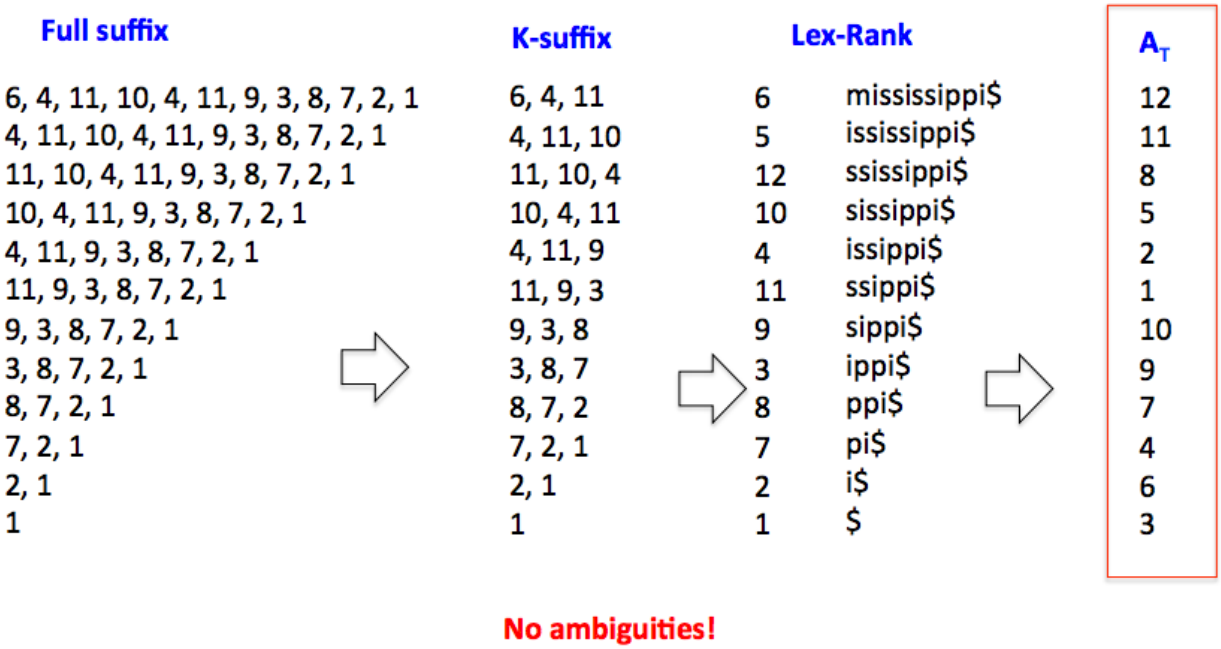


Figure 5.7: Second iteration on the resulting string of lexicographic ranks



5.4.4 Analysis

This algorithm works! However, each iteration takes $O(n)$ time. In a worst case scenario we may have to do n iterations (i.e. we may have to recurse in step 4 $O(n)$ times), which would make the overall running time $O(n^2)$. For example, if we had the string S=xxxxxx$, we can verify that this will take n calls to this naive algorithm in order for condition 2 to be satisfied. We can do better!

5.5 Intuition 3

We can use divide and conquer to help speed up the algorithm.
Divide and conquer: solve a smaller instance of the problem and extend it to a full solution in linear time.

5.5.1 Proof of linearity of divide-and-conquer

Let T=running time and q = number of subproblems and $q > 1$ be a constant value:

$$T(n) = T(\frac{n}{q}) + O(n)$$

$$T(1) = 1$$

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

To resolve the lexicographic ordering of 2 suffixes i and j, it is enough (for $k = 3$) to know the "true" rank of only one representative from $\{i+1, i+2\}$ and one representative from $\{j+1, j+2\}$

5.5.2 Divide-and-conquer for our algorithm

Knowing the "true" ranks of $^2/_3$ of the suffixes (evenly distributed) is sufficient to annotate the remaining third.

Split: For every triplet of suffixes $\{i, i+1, i+2\}$ take only two of them and sort as in Algorithm 1.

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

5.6 KS-Algorithm

5.6.1 Overall Algorithm

```
1. Divide suffixes into 2 groups:
     Group A: {i} such that i % 3 != 0
     Group B: {i} such that i % 3 == 0
2. Sort group A with Algorithm 2 (described below)
3. Merge group B into group A with Algorithm 3 (described below)
```

5.6.2 Algorithm 2

```
1. Use radix sort to sort suffixes by the first k=3 characters.
2. If there are no ambiguities, return the sorted order.
3. Otherwise:
    A. Generate a modified string S' by substituting each suffix
       in A with its rank (based on the first k=3 characters).
    B. Recursively apply the KS-algorithm on the modified string
       S' and return the results.
```

5.6.3 Algorithm 3

```
1. Represent every index i in B by the pair <S[i], rank[i+1]> (with
   rank calculated from group A).
2. Radix sort the array of pairs.
3. Merge group A and group B:
    To determine the relative order between i in A and j in B:
```

```
A. If S[i] != S[j], order accordingly
B. If S[i] == S[j]:
    1. If i+1 is in A then compare rank[i+1] against rank[j+1].
       This works because both i+1 and j+1 are in A, and you
       already know their relative ranks from Algorithm 2.
    2. Otherwise, lexicographically compare <S[i+1], rank[i+2]>
       and <S[j+1], rank[j+2]>. This works because i+2 must be
       in A if i+1 is in B, although you still need to compare the actual characters.
```

5.7 Analysis

Algorithm 2 Running time (use Masters Theorem to simplify):

$$T(n) = T(\frac{2n}{3}) + O(n)$$

$$T(n) = O(n)$$

Algorithm 3 Running time:

$$T(n) = O(n)$$

Altogether, the overall running time is $O(n)$.

5.8 Example

Figure 5.8: Step 1: Partition the suffix array into two groups, A and B

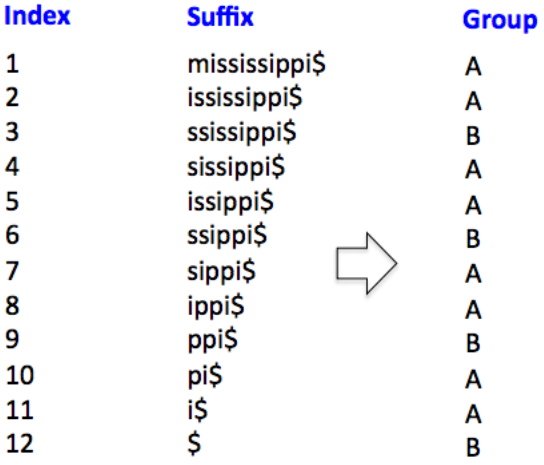


Figure 5.9: Step 2: Sort group A, if there are ambiguities, run KS on a new string S by replacing each suffix in A with its lex-rank.

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

New string: 5 8 7 6 3 3 2 1

Figure 5.10: Step 3: Running KS on the modified string, so we split into groups A and B.

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

Figure 5.11: Step 4: Run ALG2 on group A. No ambiguities

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

No ambiguities!

Figure 5.12: Step 5: Using the ranks of A and pair representation of B, we can merge the groups of A and B

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

Figure 5.13: Step 6: The resulting merged rankings of A and B.

Index	Suffix	Group	Rank (both sets)	Original suffix
1	5 8 7 6 3 3 2 1	A	5	mississippi$
2	8 7 6 3 3 2 1	A	8	sissippi$
3	7 6 3 3 2 1	B	7	sippi$
4	6 3 3 2 1	A	6	pi$
5	3 3 2 1	A	4	ississippi$
6	3 2 1	B	3	issippi$
7	2 1	A	2	ippi$
8	1	A	1	i$

Figure 5.14: Step 7: Now we have our original group A and pair-wise representation of B, we can sort B.

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

Figure 5.15: Step 8: Merge the two lists A and B to determine the final ordering

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)