UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

Linear Time Suffix Array Construction

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Overview

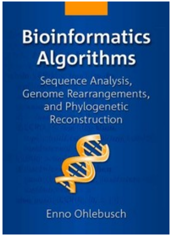
Previous Lectures

- Ukkonen's algorithm: linear time **suffix tree** construction
- **Suffix links**
- Kasai's algorithm: linear time **LCP array** construction

Today

- Direct linear time suffix array construction using **induced sorting**

Recommended Literature

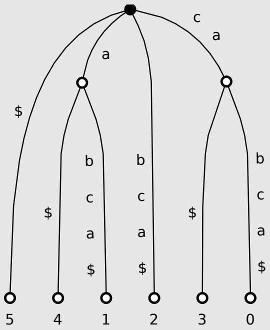


Further Reading

- Shrestha et al. A bioinformatician's guide to the forefront of suffix array construction algorithms. Brief. Bioinformatics 2014 Mar;15(2):138-54
- G. Nong, S. Zhang and W. H. Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. Proceedings of 19th Data Compression Conference (IEEE DCC), 2009.

Suffix trees and suffix arrays

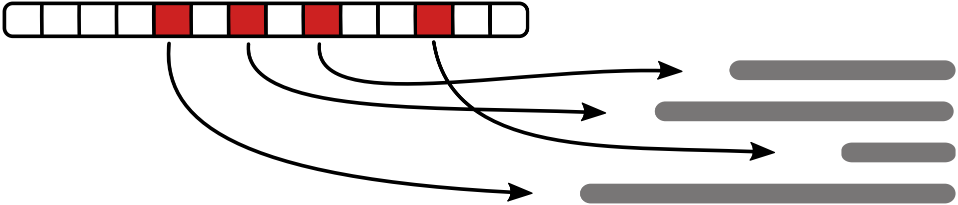
Suffix tree for the string $T = cabca\$$.



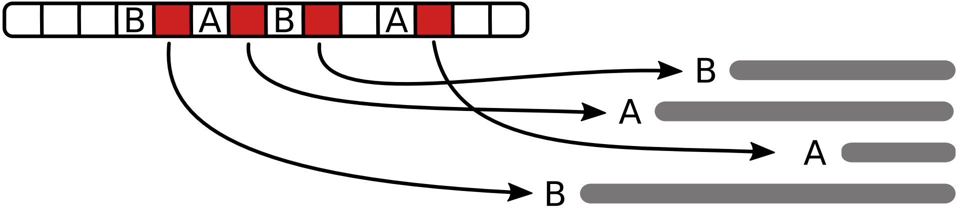
A suffix array of a string s$ with $|s\$| = n$ is defined as the permutation *pos* of $\{0, .., n-1\}$ that represents the lexicographic ordering of all suffixes of s$.
$\mathrm{pos} = [5, 4, 1, 2, 3, 0]$.

Induced Sorting Idea

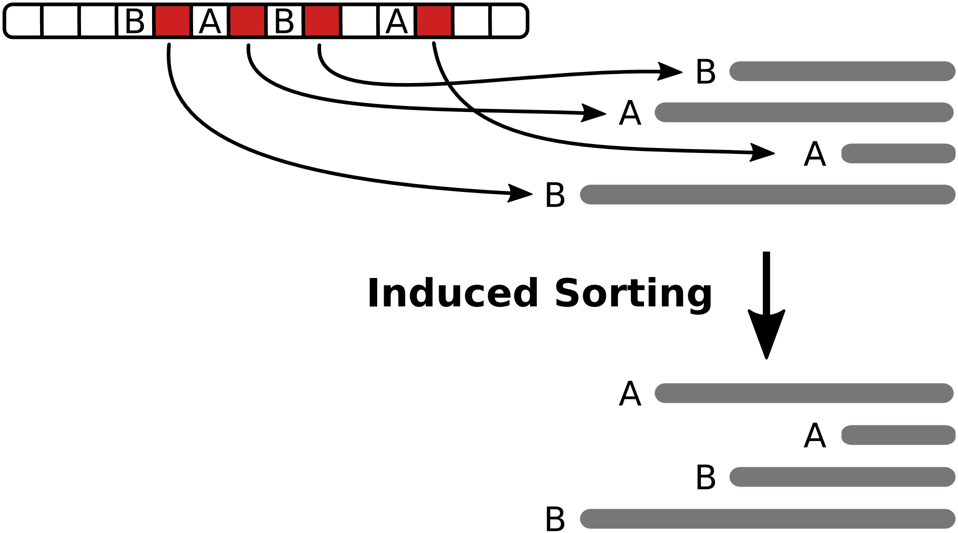
Induced Sorting Idea



Induced Sorting Idea



Induced Sorting Idea



Definition of L-/S-positions

Definition (L-position, S-position)

Let $s\$$ be a string of length n with sentinel, such that $s[n-1] = \$$.
Let $0 \leq p < n-1$ be a position in the text. We say,

- p is an **L-position** (L means **larger**), if $s[p \ldots] > s[p+1 \ldots]$,
- p is an **S-position** (S means **smaller**), if $s[p \ldots] < s[p+1 \ldots]$,
- The position of the sentinel $n-1$ is defined as S-position.

(Note that no two suffixes can be identical.)

Definition of L-/S-positions

Definition (L-position, S-position)

Let $s\$$ be a string of length n with sentinel, such that $s[n-1] = \$$.
Let $0 \leq p < n-1$ be a position in the text. We say,

- p is an **L-position** (L means **larger**), if $s[p \ldots] > s[p+1 \ldots]$,
- p is an **S-position** (S means **smaller**), if $s[p \ldots] < s[p+1 \ldots]$,
- The position of the sentinel $n-1$ is defined as S-position.

(Note that no two suffixes can be identical.)

```
            0.........1.........2.
Position p  0123456789012345678901
Sequence s  gccttaacattattacgccta$
type        LSSLLSSLSLLSLLSSLSSLLS
```

Computing the L-/S-positions in the type array

The type information can be computed in linear time
with a scan through the text from right to left:

```
def compute_types(T):
  n = len(T)
  typ = ['?'] * (n-1) + ['S']
  for i in range(n-2, -1, -1):
      typ[i] = 'L' if T[i] > T[i+1] else \
               'S' if T[i] < T[i+1] else typ[i+1]
  return typ
```

In a real implementation, we use a bit vector (0/1) to represent the types.

Definitions: LMS position / interval / substring

Definition (LMS-interval, LMS-substring)

- S-positions located to the right of an L-position are called **LMS positions** (for **leftmost S** position).
- A pair of positions $[i, j]$ is called **LMS interval** of s, if either
  - $i < j$ and both i and j are LMS-positions and there are no LMS-positions between i and j, or
  - $i = j = n - 1$.
- Each LMS interval $[i, j]$ is associated with its **LMS substring** $s[i \dots j]$.

Observations

- Position $n - 1$ with the sentinel is always an LMS-position.
- Whether an S-position is an LMS-position can be determined in constant time, looking up its type and the type to the left in the `typearray`.

Example: type array, LMS substrings

```
                0         1         2
position p      0123456789012345678901
sequence s      gccttaacattattacgcctta$
```

Example: type array, LMS substrings

```
                0         1         2
position p      0123456789012345678901
sequence s      gccttaacattattacgccta$

type            LSSLLSSLSLLSLLSSLSSLLS
```

Example: type array, LMS substrings

```
             0         1         2
position p   0123456789012345678901
sequence s   gccttaacattattacgccta$

type         LSSLLSSLSLLSLLSSLSSLLS

LMS?          *   *  *  *  *  *   *
LMS-substr    cctta  atta  acgc   $
                  aaca  atta  ccta$
```

Overview of Induced Sorting

Notation

- s is the input sequence,
- `pos` is the desired output suffix array of s.

Induced sorting

- Scan s to compute the `type` array
- Scan `type` to find all LMS positions in s
- Phase I - Sort suffixes at LMS positions (complex; recursive)
- Phase II - Sort all remaining suffixes of s (easy)
- Output `pos`

Code: Overview

```python
def sais_main(T, alphabet_size):
    # T: text (bytes, numpy array, not str!), T[n-1]=0
    # alphabet_size, 1 <= T[i] < alphabet_size for all i < n-1

    pos = np.empty(len(T), dtype=np.int64)
    # B[a]: total number of characters in T that are <= a
    B = count_cumulative_characters(T, alphabet_size)
    types = compute_types(T)
    lms_positions = find_lms_positions(types)
    # Phase 1 sorts lms_positions lexicographically in-place,
    # may recurse into sais_main() with a reduced text.
    phase1(T, B, types, lms_positions, pos)
    # Phase 2 sorts all suffixes from correctly sorted LMS.
    phase2(T, B, types, lms_positions, pos)
    return pos
```

Code: Initialization, buckets and types

```python
def count_cumulative_characters(T, alphabet_size):
    # B[a]: total number of characters in T that are <= a
    B = np.zeros(alphabet_size, dtype=np.uint64)
    for a in T:
        B[a] += 1
    for a in range(1, alphabet_size):
        B[a] += B[a-1]
    return B
```

```python
def compute_types(T):
    # Compute position types (SMALLER=0, LARGER=1) for T
    n = len(T)
    types = np.zeros(n, dtype=np.uint8)  # types[n-1] = SMALLER
    for i in range(n-2, -1, -1):
        types[i] = LARGER if T[i] > T[i+1] else \
                   SMALLER if T[i] < T[i+1] else types[i+1]
    return types
```

Code: Initialization, LMS positions

```python
def find_lms_positions(types):
    n = len(types)
    # count the number of LMS positions first
    m = 0
    for p in range(1, n):
        m += (types[p] == SMALLER and types[p-1] == LARGER)
    # allocate array of just the correct size m
    lms_positions = np.empty(m, dtype=np.int64)
    # now fill the array with the actual LMS positions
    m = 0
    for p in range(1, n):
        if types[p] == SMALLER and types[p-1] == LARGER:
            lms_positions[m] = p
            m += 1
    return lms_positions
```

Code: Overview again

```python
def sais_main(T, alphabet_size):
    # T: text (bytes, numpy array, not str!), T[n-1]=0
    # alphabet_size, 1 <= T[i] < alphabet_size for all i < n-1

    pos = np.empty(len(T), dtype=np.int64)
    # B[a]: total number of characters in T that are <= a
    B = count_cumulative_characters(T, alphabet_size)
    types = compute_types(T)
    lms_positions = find_lms_positions(types)
    # Phase 1 sorts lms_positions lexicographically in-place,
    # may recurse into sais_main() with a reduced text.
    phase1(T, B, types, lms_positions, pos)
    # Phase 2 sorts all suffixes from correctly sorted LMS.
    phase2(T, B, types, lms_positions, pos)
    return pos
```

Phase II

Sorting the non-LMS suffixes

Let's start with Phase II (Phase I uses elements of Phase II):

Definition (Bucket)

A maximal interval of the suffix array pos,
in which the referenced suffixes start with the same character,
is called a **bucket**.

There are as many buckets as characters in the used alphabet,
plus the one for the sentinel character.

Sorting the non-LMS suffixes

Lemma

Within each bucket of the suffix array, the L-positions appear before the S-positions.

Proof

Let p be an S-position, and let q be an L-position, let $s[p] = s[q] = b \in \Sigma$, so both p and q are in the b-bucket. Then the suffix $p+1$ is **larger** than suffix p, and suffix $q+1$ is **smaller** than suffix q. Because $s[p] = s[q]$, the order of p vs. q is determined by $p+1$ vs. $q+1$, but $q+1$ comes before $p+1$ in the lexicographic order.

Illustration

Let $a < b < c$; suffix q is b^+a, whereas p is b^+c:

```
q               p
bbb...a    <    bbb...c
L               S
```

Sorting the non-LMS suffixes

Lemma

Within each bucket of the suffix array, the L-positions appear before the S-positions.

Bucket	$		a		c		t	
	L	S	L	S	L	S	L	S

Sorting the non-LMS suffixes

Idea

- Use the already sorted **LMS-positions** (a subset of the S-positions) to **sort the L-positions** correctly, and then
- use the sorted **L-positions** to sort all **S-positions**.

This is why the algorithm is called **induced sorting**:
The order of one type of suffixes completely induces the ordering of the others.

Preparing the Suffix Array

Step (1)

- Initialize pos with **unknown** at each position
- Mark the beginning and end of each bucket by pointers
- Write the **sorted** LMS-positions (phase I) at the end of their respective buckets.

Preparing the Suffix Array

Step (1)

- Initialize pos with **unknown** at each position
- Mark the beginning and end of each bucket by pointers
- Write the **sorted** LMS-positions (phase I) at the end of their respective buckets.

```
             0         1         2
position p   0123456789012345678901
sequence s   gccttaacattattacgccta$
type         LSSLLSSLSLLSLLSSLSSLLS
LMS?          *   *  *  *  *  *   *

rank r   | 0| 1  2  3  4  5  6| 7  8  9 10 11 12|13 14|15 16 17 18 19 20 21|
bucket   | $| a  a  a  a  a  a| c  c  c  c  c  c| g  g| t  t  t  t  t  t  t|
pos/     |21| .  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|
```

Sorting the L-positions (Induced Sorting)

Step (2)

- Iterate through pos from **left to right** with index r.
- If $\text{pos}[r]$ is unknown, skip index r .
- Otherwise, look at $\text{pos}[r] - 1$:
  1. If $\text{pos}[r] - 1$ is an L-position, enter it at the first free position in its bucket.
  2. If $\text{pos}[r] - 1$ is an S-position, skip index r.

Result

All L-positions are entered in the suffix array in correct order.

Example: Sorting the L-positions

```
                0         1         2
position p      0123456789012345678901
sequence s      gccttaacattattacgcccta$
type            LSSLLSSLSLLSLLSSLSSLLS
LMS?             *   *  *  *  *  *   *

rank r      | 0| 1  2  3  4  5  6| 7  8  9 10 11 12|13 14|15 16 17 18 19 20 21|
bucket      | $| a  a  a  a  a  a| c  c  c  c  c  c| g  g| t  t  t  t  t  t  t|
pos         |21| .  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|
            |^S|vL               |                 |     |                    |
pos         |21|20  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|
            |  |^L               |                 |     |vL                  |
pos         |21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  .  .  .  .  .  .|
            |  |      ^S         |                 |     |   vL               |
pos         |21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4  .  .  .  .  .|
            |  |         ^S      |                 |     |      vL            |
pos         |21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4 13  .  .  .  .|
            |  |            ^S   |                 |     |         vL         | ... ...
pos         |21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3 12  9|
```

Sorting the S-positions (Induced Sorting)

Step (3)

1. Remove all the S-positions from pos, except \$.
2. Iterate through pos from **right to left** with index r.
3. If $\mathrm{pos}[r]$ is unknown, skip index r.
4. Otherwise, look at $\mathrm{pos}[r] - 1$:
   - If $\mathrm{pos}[r] - 1$ is an S-position, enter it at the rightmost free position in its bucket.
   - If $\mathrm{pos}[r] - 1$ is an L-position, skip index r.

Result

All S-positions are entered in the suffix array in correct order.

Example: Sorting the S-positions (Induced Sorting)

```
             0         1         2
position p   0123456789012345678901
sequence s   gccttaacattattacgccta$
type         LSSLLSSLSLLSLLSSLSSLLS
LMS?          *   *  *  *  *  *   *

rank r | 0| 1  2  3  4  5  6| 7  8  9 10 11 12|13 14|15 16 17 18 19 20 21|
bucket | $| a  a  a  a  a  a| c  c  c  c  c  c| g  g| t  t  t  t  t  t  t|
pos(2) |21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3 12  9|

pos    |21|20  .  .  .  .  8| 7  .  .  .  .  .|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  .  . 11  8| 7  .  .  .  .  .|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  .  . 11  8| 7  .  .  .  .  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  .  . 11  8| 7  .  .  . 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  .  . 11  8| 7  .  . 15 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  .  . 11  8| 7  .  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  .  . 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  . 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  .  6 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  5  6 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos    |21|20  5  6 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
```

Summary and Analysis of Phase II

Phase II:

1. Enter sorted LMS suffixes into pos, set bucket pointers
2. Sort L-suffixes based on sorted LMS-suffixes (induced sorting)
3. Sort S-suffixes based on sorted L-suffixes (induced sorting)

Summary and Analysis of Phase II

Phase II:

1. Enter sorted LMS suffixes into pos, set bucket pointers
2. Sort L-suffixes based on sorted LMS-suffixes (induced sorting)
3. Sort S-suffixes based on sorted L-suffixes (induced sorting)

Running Time Analysis

- Step (1) can be done in linear time.
- Step (2) and (3) each do a linear scan through the suffix array in linear time.
- $\Rightarrow$ Phase II takes linear time.

Summary and Analysis of Phase II

Phase II:

1. Enter sorted LMS suffixes into pos, set bucket pointers
2. Sort L-suffixes based on sorted LMS-suffixes (induced sorting)
3. Sort S-suffixes based on sorted L-suffixes (induced sorting)

Running Time Analysis

- Step (1) can be done in linear time.
- Step (2) and (3) each do a linear scan through the suffix array in linear time.
- $\Rightarrow$ Phase II takes linear time.

Correctness?

Correct Sorting of L-Positions

Lemma: Correctness of Step (2)

Assuming correctly ordered LMS-positions in each bucket, then after Step (2), **all** L-positions can be found at their **correct** positions.

Correct Sorting of L-Positions

Lemma: Correctness of Step (2)

Assuming correctly ordered LMS-positions in each bucket, then after Step (2), **all** L-positions can be found at their **correct** positions.

Proof idea

- If p is a text position with rank r in pos and $p-1$ is a L-position, then $p-1$ has a rank r' with $r' > r$ by definition of an L-position.

Correct Sorting of L-Positions

Lemma: Correctness of Step (2)

Assuming correctly ordered LMS-positions in each bucket, then after Step (2), **all** L-positions can be found at their **correct** positions.

Proof idea

- If p is a text position with rank r in pos and $p-1$ is a L-position, then $p-1$ has a rank r' with $r' > r$ by definition of an L-position.
- This assures that each L-position $p-1$ will
  1. be induced by an LMS- or L-position p
  2. be induced by a position further to the left

Correct Sorting of L-Positions

Lemma: Correctness of Step (2)

Assuming correctly ordered LMS-positions in each bucket, then after Step (2), **all** L-positions can be found at their **correct** positions.

Proof idea

- If p is a text position with rank r in pos and $p-1$ is a L-position, then $p-1$ has a rank r' with $r' > r$ by definition of an L-position.
- This assures that each L-position $p-1$ will
  1. be induced by an LMS- or L-position p
  2. be induced by a position further to the left
- Complete proof by induction: Show that the first k LMS- and L-positions all appear in the correct order.

Correct Sorting of S-Positions

Lemma: Correctness of Step (3)

Assuming correctly ordered L-positions in each bucket, then after step (3), **all** positions can be found at their **correct** positions.

Correct Sorting of S-Positions

Lemma: Correctness of Step (3)

Assuming correctly ordered L-positions in each bucket, then after step (3), **all** positions can be found at their **correct** positions.

Proof idea

- Let p be a text position with rank r in pos and $p-1$ is a S-position, then $p-1$ has a rank r' with $r' < r$ (by definition of an S-position).

Correct Sorting of S-Positions

Lemma: Correctness of Step (3)

Assuming correctly ordered L-positions in each bucket, then after step (3), **all** positions can be found at their **correct** positions.

Proof idea

- Let p be a text position with rank r in pos and $p-1$ is a S-position, then $p-1$ has a rank r' with $r' < r$ (by definition of an S-position).
- This assures that each S-position $p-1$ will be induced by a position p further to the right.

Correct Sorting of S-Positions

Lemma: Correctness of Step (3)

Assuming correctly ordered L-positions in each bucket, then after step (3), **all** positions can be found at their **correct** positions.

Proof idea

- Let p be a text position with rank r in pos and $p-1$ is a S-position, then $p-1$ has a rank r' with $r' < r$ (by definition of an S-position).
- This assures that each S-position $p-1$ will be induced by a position p further to the right.
- Complete proof by induction (in k):
  Show that the last k positions all appear in the correct order.

Code: Phase II

```
def phase2(T, B0, types, lms, pos):
    # T: Text, B0: cumulative bucket sizes, types: type array
    # lms: sorted or unsorted LMS positions
    # pos: suffix array (output)

    # 0. Initialize pos by inserting LMS positions,
    B = B0.copy()  # working copy of C, to be modified
    initialize_pos_from_lms(T, B, lms, pos)
    # 1. Do a left-to-right induction scan for L-positions,
    B[:] = B0[:]  # re-set B to a clean working copy of C
    induce_L_positions(T, B, types, pos)
    # 2. Do a right-to-left induction scan for S-positions.
    B[:] = B0[:]  # re-set B to a clean working copy of C
    induce_S_positions(T, B, types, pos)
    # Result: pos has been modified as desribed.
```

Code: Phase II, Initialization

```python
def initialize_pos_from_lms(T, B, lms, pos):
    pos[:] = -1  # set everything to "unknown"
    # Insert LMS positions at right end of their buckets,
    # right-to-left, so we know where to start in each bucket.
    for p in lms[::-1]:
        a = T[p]  # character determines the bucket
        B[a] -= 1
        pos[B[a]] = p
```

Code: Phase II, L-positions

```python
def induce_L_positions(T, B, types, pos):
    # Left-to-right scan: Induce L-positions from LMS-positions
    n = len(T)
    for r in range(n):
        p = pos[r]
        if p <= 0: continue  # unknown or 0 -> skip
        if types[p-1] == SMALLER: continue  # skip S positions
        a = T[p-1]  # determine bucket
        pos[B[a-1]] = p-1
        B[a-1] += 1
```

Code: Phase II, S-positions

```
def induce_S_positions(T, B, types, pos):
    # Right-to-left scan: Induce S-positions from L-positions
    n = len(T)
    for r in range(n-1, -1, -1):
        p = pos[r]
        if p == 0: continue  # skip position 0 (no p-1)
        if types[p-1] == LARGER: continue  # skip L positions
        a = T[p-1]  # determine bucket
        B[a] -= 1
        pos[B[a]] = p-1
```

Phase I

Idea for Phase 1

Goal (hard)

Sort the LMS suffixes (i.e., suffixes starting at LMS positions)

Idea for Phase 1

Goal (hard)

Sort the LMS suffixes (i.e., suffixes starting at LMS positions)

Plan

- Only sort the LMS substrings (up to next LMS position): shorter total length ($O(n)$ instead of $O(n^2)$).
- Expand alphabet and reduce text length (LMS substring $\mapsto$ character), keeping lexicographic order of LMS substrings (“**lexicographic naming**”).
- If all LMS substrings are distinct, we have also sorted the LMS suffixes, done!
- If there are equal LMS substrings, compute suffix array of reduced text (recursively with SAIS), use that to infer correct order of LMS suffixes.

Example: Alphabet Expansion and Text Reduction

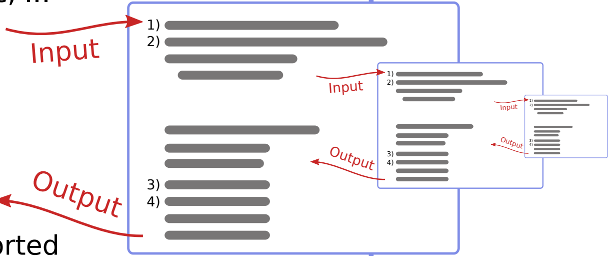
```
             0         1         2
position p   0123456789012345678901
text T       gccttaacattattacgccta$
type         LSSLLSSLSLLSLLSSLSSLLS
LMS?          *   *  *  *  *  *   *
LMS-substr    cctta  atta  acgc   $
                  aaca  atta  ccta$

p'            0   1  2  3  4  5   6
red. text R   E   A  C  C  B  D   $

r'            0   1  2  3  4  5   6
pos'[r']      6   1  4  3  2  5   0   reduced suffix array
RT[pos'[r']] 21   5 14 11  8 17   1   sorted LMS-positions
```

Overview with Recursion

1) **Phase 1:** Identify and sort LMS substrings
2) Reduce text by lexicographic naming

   A=aaca, B=acgc, ...

   E A C C B D $



   Suffix array of
   reduced text
   [6 1 4 3 2 5 0]

3) Translate into sorted LMS suffixes
4) **Phase 2:** Use sorted LMS suffixes to induce order of non-LMS suffixes

Achieving Phase I in Linear Time

Questions

1. How to sort the LMS substrings in linear time?
2. How to compare and name LMS substrings in linear time?
3. How to obtain order of LMS suffixes after recursive call ?

Achieving Phase I in Linear Time

Questions

1. How to sort the LMS substrings in linear time?
2. How to compare and name LMS substrings in linear time?
3. How to obtain order of LMS suffixes after recursive call ?

Sorting LMS Substrings

Surprisingly, it can be done by another run of Phase II:

- Enter **unsorted** LMS-positions into correct buckets of pos
- Induce order of L-positions based on **unsorted** LMS-positions
- Induce order of S-positions based on sorted S-positions

Result: Suffixes at LMS positions correctly sorted **up to next LMS position**:

```
...  SSSSLLLLLS  ...
...  *         * ...
```

Achieving Phase I in Linear Time

Text Reduction and Lexicographic Naming

1. Sort LMS substrings (phase 2) into $\mathtt{pos}$ (previous slide)
2. Extract partially sorted LMS positions from $\mathtt{pos}$
3. Compare LMS substrings in lexicographic order, \$ first,
   assign new "name" (number) if different from previous string.
4. In parallel, build new reduced text R from names at LMS positions,
   build map $\mathtt{RT}$ from R-positions to T-LMS-positions.
5. If all LMS substrings are unique, we already have sorted LMS suffixes.
   Otherwise recurse on R (next slide) to obtain $\mathtt{pos}'$.
6. Total time without recursion: $O(n)$.

Achieving Phase I in Linear Time

Recursion

Situation: We have

- paritally sorted LMS suffixes `lms`,
- reduced text R,
- map $\mathtt{RT}$ from R-positions to T-LMS-positions.

Left to do:

1. Recursively compute $\mathtt{pos}'$ of R by calling SAIS(R).
2. Overwrite `lms` by correct order of T is $\mathtt{RT}[\mathtt{pos}'[0]], \mathtt{RT}[\mathtt{pos}'[1]], \dots$.

Code: Phase I, Overview

```
def phase1(T, B, types, lms_positions, pos):
    # T: text;  B: cumulative charachter counts
    # lms_positions: LMS positions in ANY ORDER
    # pos: uninitialized, used to sort LMS positions
    alphabet_size = len(B)
    phase2(T, B, types, lms_positions, pos)
    # Compute reduced text from LMS substrings
    (R, reduced_alphabet_size, position_map) \
      = reduce_text(T, alphabet_size, types, pos, lms_positions)
    # If there are equal LMS substrings, recurse on reduced text
    if len(R) != reduced_alphabet_size:
        reduced_pos = sais_main(R, reduced_alphabet_size)
        # Re-map reduced_pos to original text positions;
        # these are the lms_positions in lexicographic order,
        for i, redp in enumerate(reduced_pos):
            lms_positions[i] = position_map[redp]
```

Code: Phase I, Text Reduction (Lexicographic Naming)

```python
def reduce_text(T, alphabet_size, types, pos, lms_positions):
    n, m = len(pos), len(lms_positions)
    names = np.full(n, -1, dtype=np.int64)  # the names
    last_lms = n-1;  names[last_lms] = 0  # sentinel at n-1
    reduced_alphabet_size = 1;  j = 0
    # go through the suffixes lexicographically, w/o sentinel
    for r in range(1, n):
        p = pos[r]  # if not LMS, skip it:
        if p==0 or types[p]!=SMALLER or types[p-1]!=LARGER:
            continue
        lms_positions[j]=p;  j+=1  # write sorted LMS positions
        if lms_substrings_unequal(T, types, last_lms, p):
            reduced_alphabet_size += 1
        names[p] = reduced_alphabet_size - 1
        last_lms = p
```

Code: Phase I, Comparison of LMS Substrings

```
def lms_substrings_unequal(T, types, p1, p2):
    """Return True iff LMS substrings at p1, p2 in T differ"""
    is_lms_p1 = is_lms_p2 = False
    while True:
        if T[p1] != T[p2]: return True  # unequal
        if types[p1] != types[p2]: return True  # unequal
        if is_lms_p1 and is_lms_p2: return False  # equal
        p1 +=1; p2 += 1  # look at next positions
        # check if both or only one LMS substring ends now
        is_lms_p1 = types[p1]==SMALLER and types[p1-1]==LARGER
        is_lms_p2 = types[p2]==SMALLER and types[p2-1]==LARGER
        if is_lms_p1 and is_lms_p2: continue  # final test
        if is_lms_p1 or is_lms_p2: return True  # unequal
```

Running Time Analysis

Observations about the recursion

- The alphabet size can grow, but is bounded by n (e.g. `a`, `c`, `g`, `t` expands to `A`–`E`).
- After each reduction step for a sequence of length n (including the sentinel), the new sequence has at most length $\lfloor n/2 \rfloor$ (again including the sentinel).

Running Time Analysis

Observations about the recursion

- The alphabet size can grow, but is bounded by n (e.g. a, c, g, t expands to A–E).
- After each reduction step for a sequence of length n (including the sentinel), the new sequence has at most length $\lfloor n/2 \rfloor$ (again including the sentinel).

Find a bound on the running time $T(n)$ for these three parts:

1. Phase I without recursion: $\leq c_1\, n$
2. Recursive call: $\leq T(n/2)$
3. Phase II: $\leq c_2\, n$

Running Time Analysis

Observations about the recursion

- The alphabet size can grow, but is bounded by n (e.g. `a`, `c`, `g`, `t` expands to `A`–`E`).
- After each reduction step for a sequence of length n (including the sentinel), the new sequence has at most length $\lfloor n/2 \rfloor$ (again including the sentinel).

Find a bound on the running time $T(n)$ for these three parts:

1. Phase I without recursion: $\leq c_1\, n$
2. Recursive call: $\leq T(n/2)$
3. Phase II: $\leq c_2\, n$

Claim

$T(n) = \mathcal{O}(n)$, i.e., SAIS takes linear time in $n = |T|$.

Running Time Analysis (Proof)

Proof of Claim $T(n) = \mathcal{O}(n)$

1. Phase I without recursion: $\leq c_1\, n$
2. Recursive call: $\leq T(n/2)$
3. Phase II: $\leq c_2\, n$

Let $C := c_1 + c_2$. Then $T(1) = \mathcal{O}(1)$, and thus

$$\begin{aligned} T(n) &\leq c_1\, n + T(n/2) + c_2\, n \\ &= C\, n + T(n/2) \\ &= C\, n + C\, n/2 + T(n/4) \\ &\leq C\, n\,(1 + 1/2 + 1/4 + \ldots) + T(1) \\ &= 2C\, n + \mathcal{O}(1) \; = \; \mathcal{O}(n). \end{aligned}$$

q.e.d.

Summary

Linear suffix array construction by induced sorting (SAIS)

1. Sorted LMS-suffixes can be used to induce sorting of L-suffixes.
2. Sorted L-suffixes can be used to induce sorting of S-suffixes.
3. Sort LMS-suffixes by sorting LMS-substrings first
   (how? induced sorting on **unsorted** LMS-positions)
4. Reduce text by lexicographic naming of LMS-substrings
5. If equal LMS-substrings exist, recurse on reduced text
6. LMS-order of original text is obtained from suffix array of reduced text

Possible exam questions

- Explain the principle of induced sorting.
- Why are L-positions on the left and S-positions on the right of each bucket?
- What is the goal of the text reduction step?
- Conduct the first iteration of induced sorting for a small example string.
- Explain why the induced sorting algorithm has linear running time.