Connections between Suffix Trees and Arrays

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Previous Lectures

- **Suffix trees** and **suffix arrays**
- Enhancing suffix arrays with **Longest Common Prefix (LCP)** arrays
- **Applications** of suffix trees
- **Applications** of enhanced suffix arrays
- Linear time construction algorithms

Today's Lecture

Relationship of suffix trees and suffix arrays

- Can enhanced suffix arrays be used as **"virtual" suffix trees**?
- How to do top-down traversals using enhanced suffix arrays?
  - Characterizing child intervals
  - **Range Minimum Queries (RMQs)**
- Application: pattern search

Correspondence between Suffix Tree Nodes and Suffix Array Intervals

Suffix Trees vs. Suffix Arrays

Observation I

Every **suffix tree node** corresponds to **suffix array interval**.

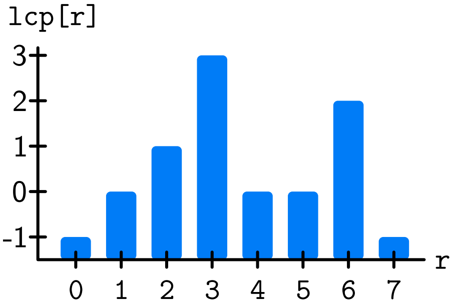
Observation II

The **opposite is not true**, i.e. some suffix array intervals (there are $O(n^2)$) do not correspond to suffix tree nodes (there are $O(n)$).

Question

How to characterize SA intervals that do correspond to a ST node?

Example: Suffix Array Intervals



r	lcp[r]	T[pos[r]...]
0	-1	$
1	0	a$
2	1	ana$
3	3	anana$
4	0	banana$
5	0	na$
6	2	nana$
7	-1	

d-intervals

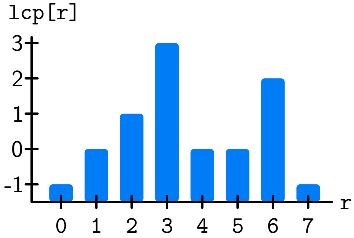
Let $\texttt{pos}$ and $\texttt{lcp}$ be the suffix array and lcp array of a text $T \in \Sigma^n$, respectively.
An interval $[L, R]$ is called **d-interval** if

- $\texttt{lcp}[L] < d$,
- $\texttt{lcp}[R+1] < d$,
- $\texttt{lcp}[r] \geq d$ for $L < r \leq R$, and
- $\min\{\texttt{lcp}[r] \mid L < r \leq R\} = d$.

d-intervals

Let pos and `lcp` be the suffix array and lcp array of a text $T \in \Sigma^n$, respectively.
An interval $[L, R]$ is called **d-interval** if

- $\texttt{lcp}[L] < d$,
- $\texttt{lcp}[R+1] < d$,
- $\texttt{lcp}[r] \geq d$ for $L < r \leq R$, and
- $\min\{\texttt{lcp}[r] \mid L < r \leq R\} = d$.

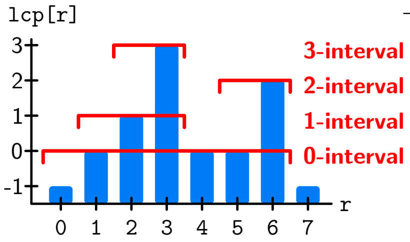


r	lcp[r]	T[pos[r]...]
0	-1	$
1	0	a$
2	1	ana$
3	3	anana$
4	0	banana$
5	0	na$
6	2	nana$
7	-1	

d-intervals

Let pos and lcp be the suffix array and lcp array of a text $T \in \Sigma^n$, respectively.
An interval $[L, R]$ is called **d-interval** if

- $\texttt{lcp}[L] < d$,
- $\texttt{lcp}[R+1] < d$,
- $\texttt{lcp}[r] \geq d$ for $L < r \leq R$, and
- $\min\{\texttt{lcp}[r] \mid L < r \leq R\} = d$.



r	lcp[r]	T[pos[r]...]
0	-1	$
1	0	a$
2	1	ana$
3	3	anana$
4	0	banana$
5	0	na$
6	2	nana$
7	-1	

Mapping Intervals to Nodes

Observation

Let $[L, R]$ be a d-interval and $[L', R']$ be a d'-interval. Then **either**

- $[L, R]$ and $[L', R']$ are disjoint, **or**
- $[L, R]$ is included in $[L', R']$ or vice versa.

By this property, the **"included in" relationship** between intervals induces a tree, called the **LCP interval tree**.

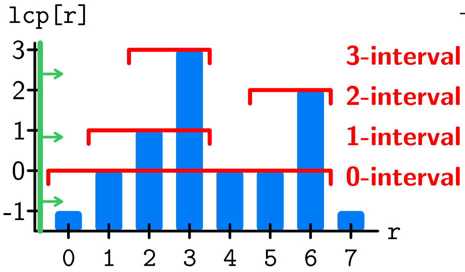
Lemma

The LCP interval tree is **isomorphic** to the suffix tree (w/o leafs).

Traversing the LCP interval tree

Idea

- Sweep from left to right
- Keep active intervals in stack
- Current lcp value higher than top of stack: create interval
- Current lcp value lower than top of stack: end/output interval



r	lcp[r]	T[pos[r]...]
0	-1	$
1	0	a$
2	1	ana$
3	3	anana$
4	0	banana$
5	0	na$
6	2	nana$
7	-1	

Code: Bottom-Up Traversal

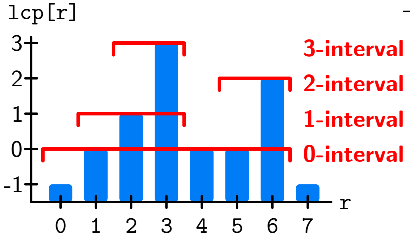
```
def bottom_up_traversal(pos, lcp):
  # store pairs: (lcp value, left boundary)
  node_stack = stack()
  node_stack.push((0,0))
  for k in range(1,len(pos)+1):
    next_start = k - 1
    while (not node_stack.empty()) and
          (lcp[k] < node_stack.peek()[0]):
      lcp_value, start = node_stack.pop()
      yield lcp_value, start, k
      next_start = start
    if (not node_stack.empty()) and
       (lcp[k] > node_stack.peek()[0]):
      node_stack.push([lcp[k], next_start])
```

Getting from Parent to Child Interval

d-intervals

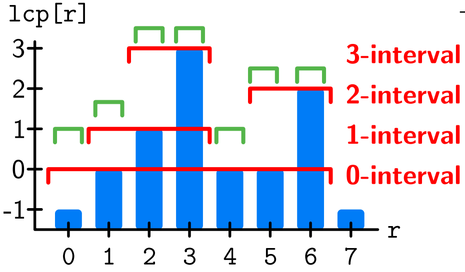
Let pos and lcp be suffix array and LCP array of a text $T \in \Sigma^n$, respectively.
An interval $[L, R]$ is called ***d*-interval** if

- $\texttt{lcp}[L] < d$,
- $\texttt{lcp}[R+1] < d$,
- $\texttt{lcp}[r] \geq d$ for $L < r \leq R$, and
- $\min\{\texttt{lcp}[r] \mid L < r \leq R\} = d$.



r	lcp[r]	T[pos[r]...]
0	-1	$
1	0	a$
2	1	ana$
3	3	anana$
4	0	banana$
5	0	na$
6	2	nana$
7	-1	

Characterizing Child Intervals



r	lcp[r]	T[pos[r]...]
0	-1	$
1	0	a$
2	1	ana$
3	3	anana$
4	0	banana$
5	0	na$
6	2	nana$
7	-1	

Lemma

Let $[L, R]$ be a d-interval, and let $i_1, \ldots, i_M$ be all positions such that $L < i_1 < \ldots < i_M \leq R$ and $\mathtt{lcp}[i_k] = d$ for all k. These positions are called **d-indices**. Then the **child intervals** of $[L, R]$ are now given by $[L, i_1 - 1], [i_1, i_2 - 1], \ldots, [i_M, R]$.

Why are Child Intervals d'-Intervals for some $d' \geq d$?

Let $[L, R]$ be a d-interval, and let $i_1, \ldots, i_M$ be all positions such that $L < i_1 < \ldots < i_M \leq R$ and $\texttt{lcp}[i_k] = d$ for all k. These positions are called **d-indices**. Then the **child intervals** of $[L, R]$ are now given by $[L, i_1 - 1], [i_1, i_2 - 1], \ldots, [i_M, R]$.

Let $\texttt{pos}$ and $\texttt{lcp}$ be suffix array and LCP array of a text $T \in \Sigma^n$, respectively. An interval $[L, R]$ is called **d'-interval** if

- $\texttt{lcp}[L] < d', \quad \texttt{lcp}[R+1] < d'$,
- $\texttt{lcp}[r] \geq d'$ for $L < r \leq R, \quad \min\{\texttt{lcp}[r] \mid L < r \leq R\} = d'$.

Why are Child Intervals d'-Intervals for some $d' \geq d$?

Let $[L, R]$ be a d-interval, and let $i_1, \ldots, i_M$ be all positions such that $L < i_1 < \ldots < i_M \leq R$ and $\texttt{lcp}[i_k] = d$ for all k. These positions are called **d-indices**. Then the **child intervals** of $[L, R]$ are now given by $[L, i_1 - 1], [i_1, i_2 - 1], \ldots, [i_M, R]$.

Let $\texttt{pos}$ and $\texttt{lcp}$ be suffix array and LCP array of a text $T \in \Sigma^n$, respectively. An interval $[L, R]$ is called **d'-interval** if

- $\texttt{lcp}[L] < d'$, $\quad \texttt{lcp}[R+1] < d'$,
- $\texttt{lcp}[r] \geq d'$ for $L < r \leq R$, $\quad \min\{\texttt{lcp}[r] \mid L < r \leq R\} = d'$.

Answer for $[i_1, i_2 - 1]$

We have $\texttt{lcp}[j] \geq d + 1$ for all $j \in \{L+1, \ldots, R\} \setminus \{i_1, \ldots, i_M\}$.
Let $d' := \min\{\texttt{lcp}[j] : \; i_1 < j \leq i_2 - 1\} > d$.
Then also $\texttt{lcp}[i_1] = d < d'$, $\texttt{lcp}[i_2] = d < d'$, $\texttt{lcp}[r] \geq d'$ for $i_1 < r \leq i_2 - 1$.

Summary: Finding Child Intervals

Let $[L, R]$ be a d-interval, and let $i_1, \dots, i_M$ be all positions such that $L < i_1 < \dots < i_M \leq R$ and $\texttt{lcp}[i_k] = d$ for all k. These positions are called **d-indices**. Then the **child intervals** of $[L, R]$ are now given by $[L, i_1 - 1], [i_1, i_2 - 1], \dots, [i_M, R]$.

In other words

To find **child intervals**,
we need to find the positions in the interval, where **the lcp value is minimal**.

Solution: Range Minimum Queries

- **Given:** Array A
- **Query:** For interval $[i, j]$, what is the smallest position $i' \in [i, j]$ such that $A[i'] = \min\{A[i], \dots, A[j]\}$?

Application: Top-Down Pattern Search (`issi`)

r	`pos`[r]	`lcp`[r]	T[`pos`[r] :]
0	13	-1	`$`
1	12	0	`i$`
2	11	1	`ii$`
3	1	2	`iississippii$`
4	8	1	`ippii$`
5	5	1	`issippii$`
6	2	4	`ississippii$`
7	0	0	`miississippii$`
8	10	0	`pii$`
9	9	1	`ppii$`
10	7	0	`sippii$`
11	4	2	`sissippii$`
12	6	1	`ssippii$`
13	3	3	`ssissippii$`
14		-1	

Range Minimum Queries

Naive Algorithms

Scan

Search whole query interval: $O(n)$ time for every query

```
def rmq_naive(A, i, j):
    best = None
    for k in range(i,j+1):
        if (best is None) or (A[k] < A[best]):
            best = k
    return best
```

Naive Algorithms

Scan

Search whole query interval: $O(n)$ time for every query

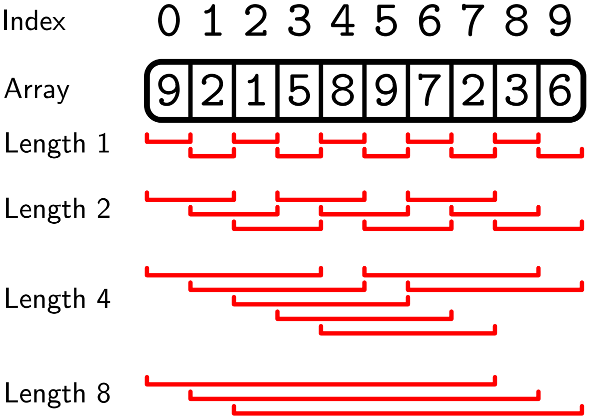
```
def rmq_naive(A, i, j):
    best = None
    for k in range(i,j+1):
        if (best is None) or (A[k] < A[best]):
            best = k
    return best
```

Table lookup

- **Preprocessing:**
  Create table with **all** intervals in $O(n^2)$ **space** and time once
- **Query:** Table lookup in $O(1)$ time for every query

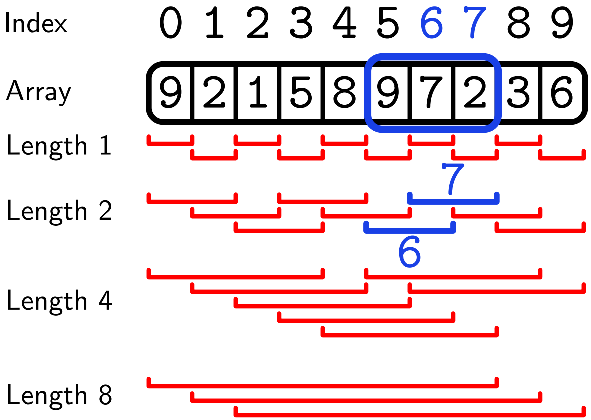
Sparse Table Approach

Preprocessing: create table with length-2^ℓ intervals in $O(n \log n)$ time



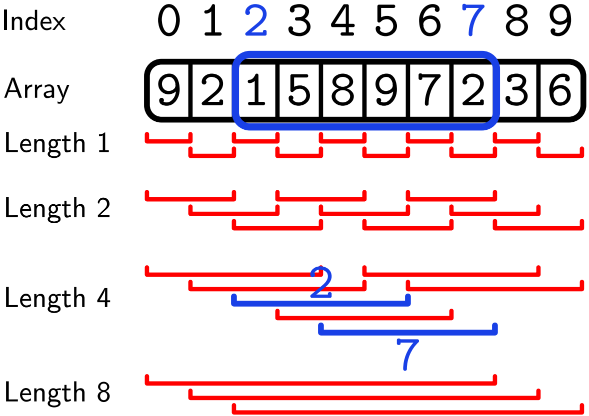
Sparse Table Approach

Querying: look up one or two (overlapping) 2^ℓ-intervals with $2^\ell \leq (j - i + 1)$



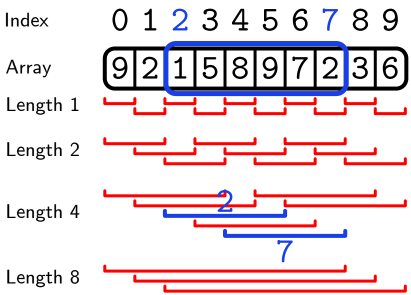
Sparse Table Approach

Querying: look up one or two (overlapping) 2^ℓ-intervals with $2^\ell \leq (j - i + 1)$



Analysis of Sparse RMQ Tables

- **Preprocessing:** $O(n \log_2 n)$ time and space:
  Compute minima locations for length $2^{\ell+1}$ from those of length 2^{ℓ};
  don't scan long intervals again!
- **Querying:** $O(1)$ time (minimum over at most two lookups)



Constant time RMQs with linear time preprocessing

Bender and Farach-Colton. "The LCA Problem Revisited", Proceedings of LATIN, 2000.

Cartesian Trees

Definition

For a given array `A`,
the **Cartesian tree** is a binary tree with exactly one node per entry
(i.e. nodes are labeled by array index), such that

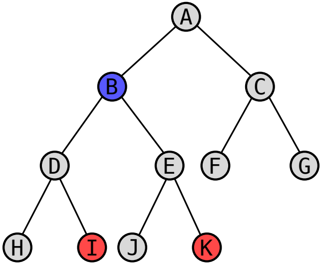
- the root corresponds to the index i of the (leftmost) minimum entry in $\mathtt{A}$,
- the left child of the root is a Cartesian tree of $\mathtt{A}[0 \ldots (i-1)]$,
- the right child of the root is a Cartesian tree of $\mathtt{A}[(i+1) \ldots (|\mathtt{A}|-1)]$,

Example: Cartesian Trees

```
      i:   0 1 2 3 4 5 6 7 8 9
      A:   9 2 1 8 5 9 3 7 2 6

depth 0:       2
               1
      1:     1             8
             2             2
      2:   0           6     9
           9           3     6
      3:         4     7
                 5     7
      4:       3   5
               8   9
```

Lowest Common Ancestor (LCA) Problem



Definitions

- A node u is called an **ancestor** of node v,
  if u lies on the (unique) path from the root to v.
- For a given rooted tree and two given nodes v_1 and v_2,
  the **lowest common ancestor (LCA)** is the node that
  is an ancestor of both v_1 and v_2 and has maximum distance from the root.

Solving RMQ using LCA

Idea

1. Build Cartesian tree `T` of input array `A`
2. Preprocess tree `T` for LCA queries
3. Each RMQ query on `A` now can be answered via an LCA query on `T`

Observation

Prove that $LCA_{\mathtt{T}}(i,j) = RMQ_{\mathtt{A}}(i,j)$ for all i, j.

Example: Solving RMQ using LCA: RMQ(3,7)

```
        i:    0 1 2 3 4 5 6 7 8 9
        A:    9 2 1 8 5 9 3 7 2 6

  depth 0:        2
                  1
        1:      1             8
                2             2
        2:    0           6     9
              9           3     6
        3:          4       7
                    5       7
        4:        3   5
                  8   9
```

Solving LCA using ± 1RMQ

Conversely, we can solve an LCA query on a tree using an RMQ query on an array.
The array has a special property: Consecutive entries differ by ± 1.

Idea

- Transform tree into array through an **Eulerian tour** (depth first search, DFS)
- For each visited node, keep track of its **depth** (distance from root)
- Now an **RMQ on this depth array** will solve LCA

Definition

An RMQ on an array $\mathtt{A}$ with $\mathtt{A}[i+1] - \mathtt{A}[i] \in \{-1, 1\}$ for all indices i is called **± 1RMQ**.

Example: Solving LCA using $\pm$1RMQ

```
      i:   0 1 2 3 4 5 6 7 8 9
      A:   9 2 1 8 5 9 3 7 2 6

depth 0:       2
      1:     1             8
      2:   0           6     9
      3:         4     7
      4:       3   5
```

```
DFS visits of i:   2 1 0 1 2 8 6 4 3 4 5 4 6 7 6 8 9 8 2
DFS depth array:   0 1 2 1 0 1 2 3 4 3 4 3 2 3 2 1 2 1 0
```

Idea: Constant-Time RMQ with Linear Preprocessing Time

Part I: Solve ± 1RMQ

- Partition input array into blocks of size $k = \lceil \frac{\log(n)}{2} \rceil$
- Key insights:
  - Normalizing a block by subtracting an offset does not change answers to an RMQ
    Example: [10,11,12,11,12] gives the same answers as [0,1,2,1,2].
  - After normalization, there are "only" 2^{k-1} different arrays in the $\pm$ setting.
    $\rightarrow$ opportunity to pre-compute all of them

Part II: Solve RMQ

RMQ $\rightarrow$ LCA $\rightarrow$ ± 1RMQ

Solving ±1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Solving ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks

Solving ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks

Solving ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks

Solving ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks

Solving ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array `A`	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks

Array `A'`	7	6	3	4	4	6	7	4	1	2

Solving ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks

Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27

Solving ±1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks										
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1

Solving ±1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks										
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1
Type	0	2	0	2	1	3	2	0	0	2

Solving ±1RMQ

Index	0 1 2	3 4 5	6 7 8	9 10 11	12 13 14	15 16 17	18 19 20	21 22 23	24 25 26	27 28 29
Array A	9 8 7	6 7 6	5 4 3	4 5 4	5 4 5	6 7 8	7 8 7	6 5 4	3 2 1	2 3 2
Blocks										
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1
Type	0	2	0	2	1	3	2	0	0	2

0	-1 -1	→	full table
1	-1 1	→	full table
2	1 -1	→	full table
3	1 1	→	full table

Solving ±1RMQ

Index	0 1 2	3 4 5	6 7 8	9 10 11	12 13 14	15 16 17	18 19 20	21 22 23	24 25 26	27 28 29
Array A	9 8 7	6 7 6	5 4 3	4 5 4	5 4 5	6 7 8	7 8 7	6 5 4	3 2 1	2 3 2
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1
Type	0	2	0	2	1	3	2	0	0	2

0	-1 -1	→	full table
1	-1 1	→	full table
2	1 -1	→	full table
3	1 1	→	full table

Preprocessing Time for ±1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks										
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1
Type	0	2	0	2	1	3	2	0	0	2

0	-1 -1	→ full table
1	-1 1	→ full table
2	1 -1	→ full table
3	1 1	→ full table

- We have $O(\frac{n}{\log n})$ blocks of size $s = \lceil \frac{\log n}{2} \rceil$.

Preprocessing Time for ± 1RMQ

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Array A	9	8	7	6	7	6	5	4	3	4	5	4	5	4	5	6	7	8	7	8	7	6	5	4	3	2	1	2	3	2

Blocks										
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1
Type	0	2	0	2	1	3	2	0	0	2

Type	Normalized block	
0	-1 -1	→ full table
1	-1 1	→ full table
2	1 -1	→ full table
3	1 1	→ full table

- We have $O(\frac{n}{\log n})$ blocks of size $s = \lceil \frac{\log n}{2} \rceil$.
- We need $O(n' \log n')$ time to build a **sparse table** for A', where n' is $O(\frac{n}{\log n})$, therefore $O(n)$ time.

Preprocessing Time for $\pm$1RMQ

Index	0 1 2	3 4 5	6 7 8	9 10 11	12 13 14	15 16 17	18 19 20	21 22 23	24 25 26	27 28 29
Array A	9 8 7	6 7 6	5 4 3	4 5 4	5 4 5	6 7 8	7 8 7	6 5 4	3 2 1	2 3 2
Array A'	7	6	3	4	4	6	7	4	1	2
Array P	2	3	8	9	13	15	18	23	26	27
Normalize	-1 -1	1 -1	-1 -1	1 -1	-1 1	1 1	1 -1	-1 -1	-1 -1	1 -1
Type	0	2	0	2	1	3	2	0	0	2

Type	Normalize	
0	-1 -1	$\longrightarrow$ full table
1	-1 1	$\longrightarrow$ full table
2	1 -1	$\longrightarrow$ full table
3	1 1	$\longrightarrow$ full table

- We have $O(\frac{n}{\log n})$ blocks of size $s = \lceil \frac{\log n}{2} \rceil$.
- We need $O(n' \log n')$ time to build a **sparse table** for A', where n' is $O(\frac{n}{\log n})$, therefore $O(n)$ time.
- We build 2^{s-1} small full tables (all pairs), for $O(2^{s-1} \cdot s^2)$ time, which is $O(n)$ for $s = \lceil \frac{\log n}{2} \rceil$.

Summary

- Relationship between suffix trees and suffix arrays
  - suffix tree nodes $\Leftrightarrow$ suffix array d-intervals
- Bottom-up traversals
- Top-down traversals: **Range Minimum Queries (RMQs)**
  - RMQ $\rightarrow$ LCA on Cartesian tree $\rightarrow$ $\pm$1RMQ
  - Linear-time construction of Cartesian tree from array (details not shown)
  - Linear-time construction of depth array from (details not shown)
  - Preprocessing in linear time and space for $\pm$1RMQ
- Application: **forward pattern search**
- Bottom line: Enhanced suffix arrays can be used as “virtual” suffix trees.

Possible Exam Questions

- How are suffix tree leafs related to suffix arrays?
- Which suffix tree operations can be simulated using suffix array plus LCP array?
- What is a range minimum query (RMQ)?
- How can RMQs be answered in constant time after at most $O(n \log n)$ preprocessing?
- Why are RMQs on the LCP useful?
- What is needed to do $O(|P|)$ time pattern matching with a suffix array?
- Explain how to achieve linear time/space preprocessing for constant time RMQs.
- What is the lowest common ancestor (LCA) problem?
- How is LCA connected to RMQ?
- What is a ± 1RMQ?