CSE 566 Spring 2023

Analysis of the Ukkonen's Algorithm & Suffix Array

Instructor: Mingfu Shao

Correctness of Ukkonen's Algorithm

- **Fact 1**: once a leaf, always a leaf!
- Proof: by examining all 5 cases
- Consequence: leaf nodes can be extended automatically by the pointer!

Correctness of Ukkonen's Algorithm

- **Fact 2:** next phase can be started whereas we stopped at the current phase!
- Proof:
  - Current phase stops when case 2 or 4 occurs (or exhausting all suffixes).
  - Previous suffixes are handled by cases 1, 3 or 5. But all these suffixes are leaf nodes now, which will be extended automatically!

k+1

k

abaaba**b** case 1
baaba**b** case 3
aaba**b** case 5
aba**b** → stops here
ba**b**
a**b**
b

Analysis of Running Time

$n = |S|$

- #phases: O(n)
- #(case 2 and case 4): O(n)
- #(case 3): O(n), as it creates a new leaf
- #(case 5): O(n), as it creates an internal node and a leaf
- Each case takes constant time, except case 5, as we might "hop" multiple times
- *To estimate the total number of "hops" throughout the algorithm.*

Analysis of Running Time

- Idea: analyze the depth of the current node (or parent node in case of current edge)
- **Fact**: following a suffix link in an (implicit) suffix tree, the depth decreases **at most by 1**.
- #(follow suffix link): O(n)
- #(total decreases of depth): O(n)
- final-depth-of-current-node = total-increase - total-decrease
- #(hops): O(n)

O(n)

O(n)

total - inc = final - node + total-decrease

= O(n)

O(n)

Suffix Array: Motivation (G. Myers, 1995)

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.
- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.
- Suffix arrays are a more space efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

Suffix Array

S = banana$ $\quad$ SA(s) = (7, 6, 4, 2, 1, 5, 3)

$\Sigma = \{\$, a, b, n\}$

$\boxed{\$ < a < b < n}$

1	banana$
2	anana$
3	nana$
4	ana$
5	na$
6	a$
7	$

sort →

7	$
6	a$
4	ana$
2	anana$
1	banana$
5	na$
3	nana$

Longest Common Prefix (LCP) Array

S = banana$

SA(s) = (7, 6, 4, 2, 1, 5, 3)

LCP(s) = (0, 1, 3, 0, 0, 2)

$|LCP(s)| = |S| - 1$

1 banana$
2 anana$
3 nana$
4 ana$
5 na$
6 a$
7 $

$i \rightarrow$

$m = \frac{i+j}{2}$

$j \rightarrow$

LCP	SA	Suffix
	7	$
0	6	a$
1	4	ana$
3	2	anana$
0	1	banana$
0	5	na$
2	3	nana$

q = aaba

q = aabn

$LCS = \min\{LCP(q, 6), LCP(q, 4)\}$

$\boxed{?}$ starts from q[i]

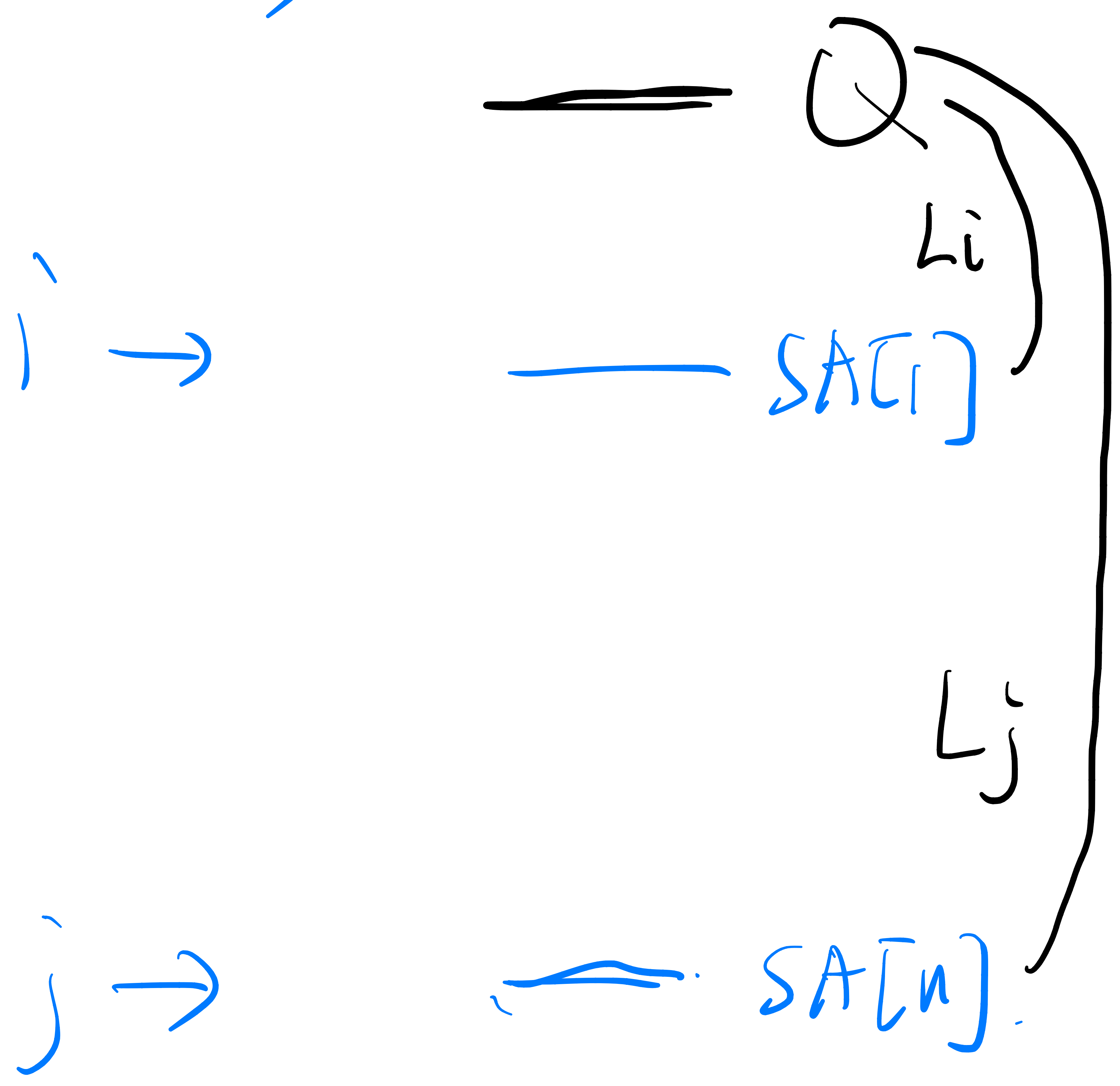
Searching for a query

- Searching questions:
  - Decide if query q is the substring of S
  - Find the longest common substring of query q and S, where the LCS starts from q[1].
- Essentially: find the position of query q in the sorted list of suffixes
- Key idea: binary search!
- Naive implementation: $O(|q| \log |S|)$.

$\Rightarrow O(|q| + \log |S|)$.

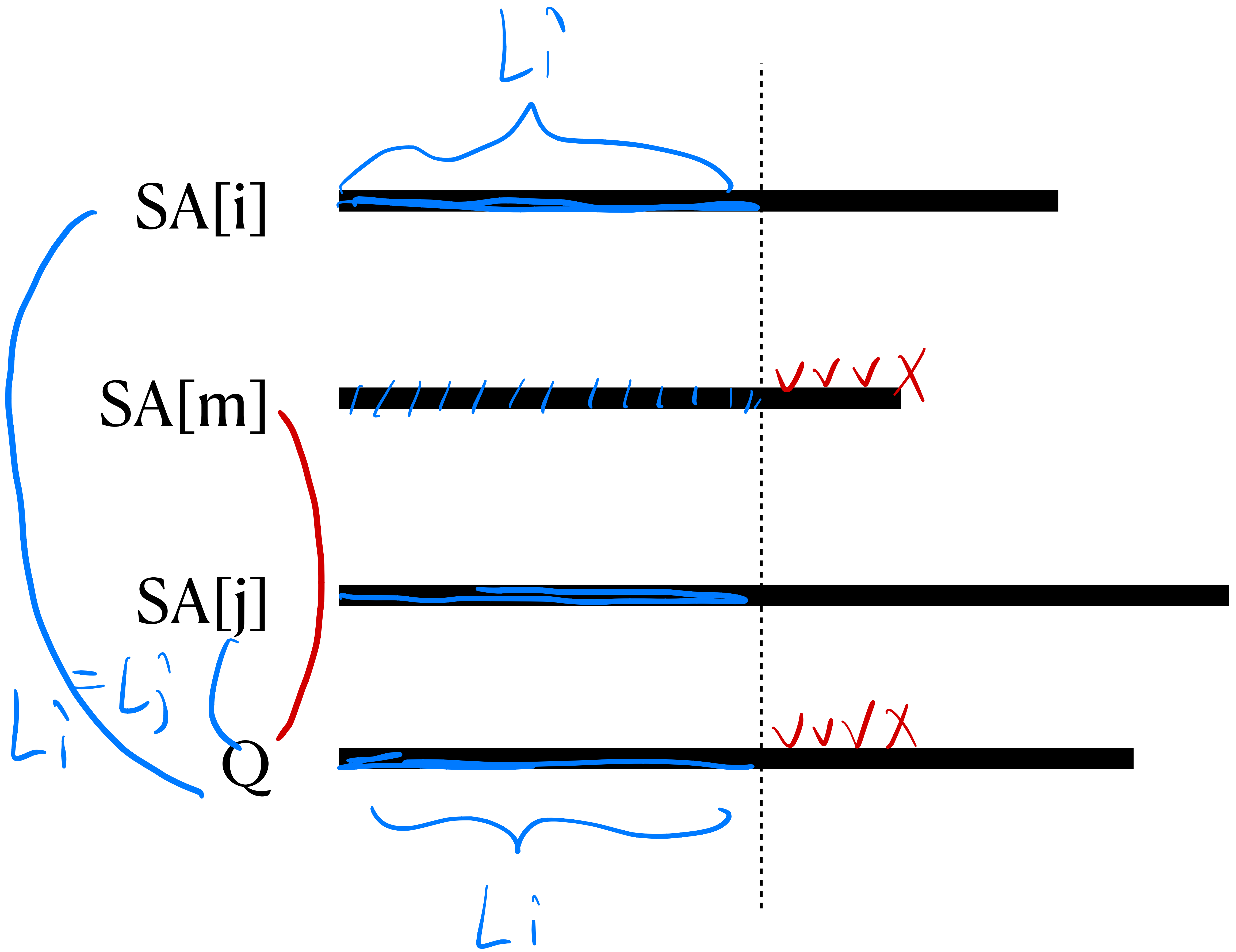
Faster Search Algorithm

- Check if Q is less than SA[1], or Q is larger than SA[n] $O(n)$
- Init: i = 1 and j = n
- Compute Li := LCP(SA[i], q) and Lj := LCP(SA[j], q) $O(n)$
- FUNCTION BS(i, j, Li, Lj)
  - Let m = (i + j) / 2
  - IF Li = Lj:
  - ELSE IF Li > Lj:
  - ELSE:
- END FUNCTION



Case 1: Li = Lj

- **Fact**: LCP(SA[m], Q) >= Li = Lj
- PROCEDURE:
  - Compare Q and SA[m] starting from position Li + 1 => LCP(SA[m], Q) Lm
  - If Q gets exhausted: return m
  - If Q < SA[m]: BS(i, m, Li, Lm)
  - If Q > SA[m]: BS(m, j, Lm, Lj)
- END PROCEDURE



Case 2: Li > Lj

- Find LCP(SA[i], SA[m])
  - Can be done in constant time!
- Case 2a: LCP(SA[i], SA[m]) = Li
- Case 2b: LCP(SA[i], SA[m]) < Li
- Case 2b: LCP(SA[i], SA[m]) > Li