Lecture 7: 문자열을 위한 고급 자료구조 (II)*

7.1 접미사 배열 (Suffix array)

정의 7.1.1 길이 n의 문자열 T의 접미사 배열 SA는 $T[\mathtt{SA[i]} : \mathtt{n}] < T[\mathtt{SA[i+1]} : \mathtt{n}]$ (단, $1 \leq i < n$)를 만족하는 길이 n의 *permutation*이다. ■

접미사 트리 (suffix tree)는 고정된 문자열 상에서의 문자열 매칭 문제를 이론적으로 다루기에는 매우 훌륭한 자료구조이지만, $\mathcal{O}(n)$개의 노드가 필요하기 때문에 실제로 구현을 해서 사용하게 되면 메모리 사용량이 지나치게 많다는 단점이 있다. 접미사 배열 (suffix array)은 이 문제를 해결하기 위하여 제안된 자료구조로, 주어진 문자열의 모든 접미사를 사전순으로 정렬했을 때, 각 순위의 접미사 시작 위치를 순서대로 나열한 길이 n의 정수 배열이다.

문제 7.1.1 문자열 T=*aababaaa$*의 접미사 배열을 구해보자.

1. T의 모든 접미사를 시작 위치와 함께 나열한다.

2. 접미사들을 사전순으로 정렬한다.

3. 각 순위의 접미사들의 시작 위치를 순서대로 나열한다.

i	1	2	3	4	5	6	7	8	9
SA$[i]$									

*이 자료는 김성환 연구원이 작성했습니다 (2016년 4월). 2020년 6월 1일 마지막으로 수정됨.

Table 1: 문자열 T =baabaabbbabaabaabb$의 접미사 배열과 LCP 배열

i	SA$[i]$	LCP$[i]$	T[SA$[i]:n$]
1	19	0	$
2	12	7	aabaabb$
3	2	3	aabaabbbabaabaabb$
4	15	4	aabb$
5	5	1	aabbbabaabaabb$
6	10	5	abaabaabb$
7	13	6	abaabb$
8	3	2	abaabbbabaabaabb$
9	16	3	abb$
10	6	0	abbbabaabaabb$
11	18	1	b$
12	11	8	baabaabb$
13	1	4	baabaabbbabaabaabb$
14	14	5	baabb$
15	4	2	baabbbabaabaabb$
16	9	1	babaabaabb$
17	17	2	bb$
18	8	2	bbabaabaabb$
19	7	0	bbbabaabaabb$

7.1.1 접미사 트리와 접미사 배열, LCP 배열

접미사 트리의 각 내부 노드의 branch를 알파벳 순으로 배치한 후 깊이우선탐색으로 노드를 차례대로 순회하면서 말단 노드에 대응하는 접미사 번호를 순서대로 나열한 결과는 접미사 배열과 같다. 접미사 트리를 생성하고 순회하는데 각각 $\mathcal{O}(n)$ 시간이 소요되므로 접미사 배열 역시 $\mathcal{O}(n)$ 시간에 생성할 수 있다.

접미사 배열로부터도 접미사 트리를 구할 수 있다. 이를 위해서는 LCP배열이라는 구조에 대해서 설명할 필요가 있다. LCP 배열은 각 인접한 순위의 접미사 간의 최장공동접두사(Longest Common Prefix) 길이를 나열한 것으로 이는 해당 접미사에 대응되는 접미사 트리의 말단 노드의 최소공통조상(Lowest Common Ancestor)에 대응되는 부분문자열의 길이와 일치한다. 표 1에 LCP 배열의 예가 나타나있다. 예를 들어 LCP[3]은 3번째 작은 접미사인 aabaabbbabaabaabb$와 4번째 작은 접미사인 aabb$의 최장공통접두사가 aab이므로 LCP[3] = 3이다.

정의 7.1.2 길이 n의 문자열 T의 *LCP* 배열은 길이 $n-1$의 배열로 각 원소가 다음과 같이 정의된다: LCP$[i]$ = lcp(T[SA$[i]:n$], T[SA$[i+1]:n$]). *(*단, $1 \le i < n$*)*■

이 LCP 배열은 접미사 배열로부터 $\mathcal{O}(n)$ 시간 내에 계산할 수 있다. 문자열 T와 그 접미사 배열 SA, 접미사 배열의 역함수 SA^{-1}이 주어져있다고 하자. 여기서 SA$^{-1}[i]$가 의미하는 것은 접미사 $T[i:n]$의 순위를 의미한다. 어떤 접미사 $T[i:n]$에 대하여 j = SA$^{-1}[i]$라고 하자. LCP$[j]$는 T[SA[j] $: n]$과 T[SA$[j+1]: n]$의 최장공통접두사

Table 2: 예제 문자열 T와 SA 및 $\texttt{SA}^{-1}$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$T[i]$	b	a	a	b	a	a	b	b	b	a	b	a	a	b	a	a	b	b	$
$\texttt{SA}[i]$	19	12	2	15	5	10	13	3	16	6	18	11	1	14	4	9	17	8	7
$\texttt{SA}^{-1}[i]$	13	3	8	15	5	10	19	18	16	6	12	2	7	14	4	9	17	11	1

길이로 정의할 수 있다. 이 길이를 k라고 하자. 그렇다면 $i' = i+1$, $j' = \texttt{SA}^{-1}[i']$이라고 할 때, $\texttt{LCP}[j']$은 반드시 $k-1$ 이상이다.

이를 이용하면 $\texttt{LCP}[j']$을 계산할 때 $T[i':n]$과 그 다음 순위의 접미사의 공통 접두사를 처음부터 비교해서 구하는 것이 아니라 각 접미사의 k번째 문자부터 비교를 시작할 수 있다. 따라서 i를 1부터 n까지 증가시키면서 이 방법을 이용하여 최장공통접두사 길이를 구할 수 있는데, 최장공통접두사는 문자열 길이인 n을 넘을 수 없으며, i가 증가할 때마다 최장공통접두사 길이는 최대 1씩 감소하므로 전체 문자 비교 횟수는 $\mathcal{O}(n)$이다.

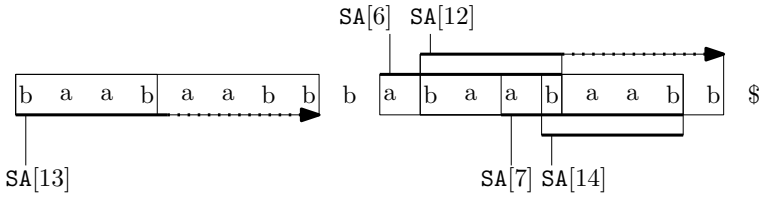

Figure 1: LCP 배열 계산 과정

LCP 배열을 이용하면 접미사 배열을 접미사 트리로 $\mathcal{O}(n)$ 시간에 변환할 수 있다. i번째 작은 접미사까지 각 접미사에 대응하는 말단 노드가 만들어진 상태라고 가정해보자. i번째 작은 접미사와 $i+1$번째 작은 접미사가 갈라지는 지점은, 루트 노드에서 i번째 작은 접미사의 말단 노드까지의 경로 상에서 있으며, 보다 정확하게는 해당 경로 상에서 루트 노드로부터의 거리가 $\texttt{LCP}[i]$인 지점이다. 이 지점을 i번째 작은 접미사의 말단 노드에서 거슬러 올라가면서 찾으면 된다. 그림 2는 이러한 방식으로 접미사 트리에 노드를 추가하는 과정을 나타낸다.

하나의 말단 노드를 추가할 때마다 최악의 경우 이전에 추가한 말단 노드의 깊이만큼 타고 올라가야 한다. 그러나 한번 타고 올라가면 그 다음 단계에서는 해당 경로는 더 이상 사용하지 않기 때문에 전체 복잡도는 선형시간이 된다. i번째 접미사의 말단 노드의 깊이가 h라고 해보자. 만일 찾으려는 $\texttt{LCP}[i]$ 지점이 말단 노드와 그 직전 branch 지점 사이에 있다면 다음 말단 노드의 깊이는 1 증가하게 된다. 그러나 만약 해당 지점을 찾기 위해 Δh만큼의 branch 노드를 거쳐서 올라갔다면 다음에 추가될 $i+1$번째 접미사에 대응되는 말단 노드의 깊이는 $h+1-\Delta h$가 된다. 따라서 잠재비용을 마지막 추가된 말단 노드의 깊이라고 두면 다음 말단 노드를 추가하기 위한 비용은 $\mathcal{O}(1)$ amortized time임을 알 수 있다.

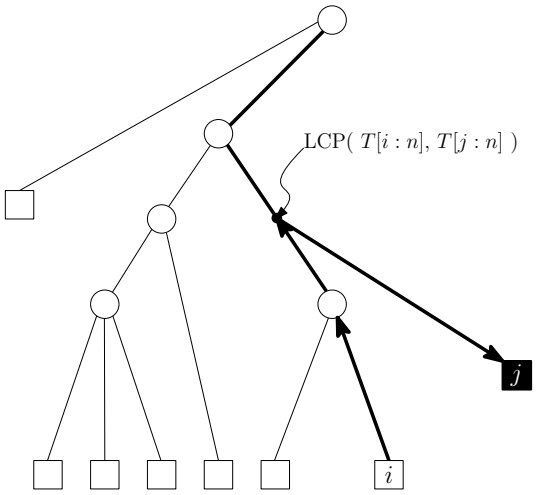

Figure 2: 접미사 배열과 LCP 배열을 이용하여 접미사 트리를 만드는 과정.

7.1.2 탐색 방법 : $\mathcal{O}(m \lg n)$

문자열 P가 문자열 T상에 나타난다면 P를 접두사로 가지는 접미사가 존재해야 한다. 접미사 배열은 사전순으로 정렬된 접미사들의 위치를 차례대로 나열한 배열로 정의되므로, P를 접두사로 가지는 접미사는 항상 연속되게 위치한다. 이러한 성질을 이용하여 접미사 배열 상에서의 문자열 매칭 결과는 하나의 인덱스 구간으로 나타낼 수 있다. 이러한 구간을 패턴 문자열 P에 대한 suffix range라고 부른다. 그림 3는 suffix range의 예를 보여준다.

	i	SA[i]	T[SA[i] : n]
	1	11	$
	2	10	a $
	3	9	a a $
i_s ➧	4	2	a a b a a b b a a $
i_e ➧	5	5	a a b b a a $
	6	3	a b a a b b a a $
	7	6	a b b a a $
	8	8	b a a $
	9	1	b a a b a a b b a a $
	10	4	b a a b b a a $
	11	7	b b a a $

Figure 3: T =baabaabbaa$의 접미사 배열 상에서 패턴문자열 P =aab 에 대한 매칭 결과. 구간 $[i_s, i_e] = [4, 5]$으로 나타낼 수 있으며, 패턴문자열의 출현위치는 SA[4] = 2, SA[5] = 5 임을 알 수 있다.

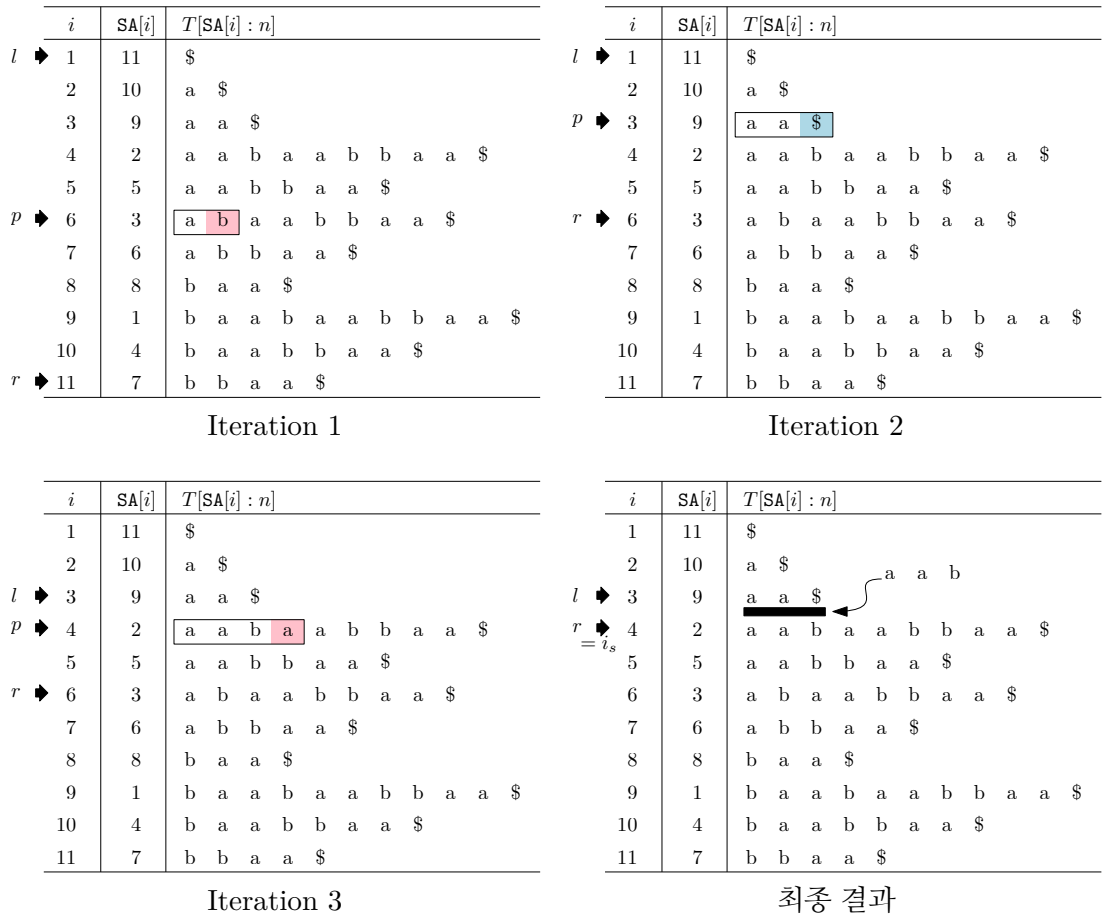

Figure 4: 이진 탐색을 이용하여 suffix range의 상단부인 i_s를 구하는 과정

길이 m의 패턴 문자열 P에 대하여 길이 n의 문자열 T의 접미사 배열 상에서 문자열 매칭을 수행하는 가장 간단한 방법은 이진 탐색을 이용하는 것이다. $T[\texttt{SA}[i_s] : n] \leq P \leq T[\texttt{SA}[i_e] : n]$를 만족하는 두 접미사 $T[\texttt{SA}[i_s] : n]$, $T[\texttt{SA}[i_e] : n]$를 접미사 배열 상에서 이진 탐색으로 찾는다. 예를 들어 i_s를 찾는다고 하자. 구간 $[1, n]$으로 시작하여, 현재 구간 $[l, r]$에 대하여 $p = \lfloor \frac{i+j}{2} \rfloor$ 번째 접미사 $T[\texttt{SA}[p] : n]$과 P를 비교한 후, P가 작다면 $[l, p]$ 구간에 대해서 반복하고, P가 크다면 $[p, r]$ 구간에 대해서 탐색을 반복하면 된다. 이러한 과정에서 불변량은 $T[\texttt{SA}[l] : n] < P < T[\texttt{SA}[r] : n]$이다. 따라서 $l + 1 = r$이 되는 시점에서 $T[\texttt{SA}[r] : n$이 P를 접두사로 가지고 있다면 $i_s = r$이라고 할 수 있다. 반대편 경계인 i_e에 대해서도 마찬가지 방법으로 탐색을 수행하면 된다.

그림 4는 이러한 과정을 통하여 문자열 T =baabaabbaa$ 상에서 P=aab에 대한 suffix range의 한쪽 경계인 i_s를 구하는 과정을 나타낸다. 각 단계에서 사각형으로 표시된 부분은 P와의 문자열 비교에 사용된 부분을 나타낸다. 각 단계에서 패턴문자열과 접미사 간의 문자열 비교가 수행되는데, 비교를 위해 사용되는 문자의 수는 최대

패턴문자열 P의 길이만큼 수행하게 되므로 $\mathcal{O}(m)$ 시간이 걸린다. 또한 각각 양 끝단의 접미사를 찾는데 $\mathcal{O}(\lg n)$ 단계가 소요되므로 탐색에 걸리는 총 시간은 $\mathcal{O}(m \lg n)$ 이다.

탐색 과정을 살펴보면 매번 이분 탐색을 수행할 때마다 패턴문자열 P와 중앙점 $p = \lfloor \frac{i+j}{2} \rfloor$에 해당하는 접미사 $T[\texttt{SA}[p] : n]$와 처음부터 비교를 수행하는 것이 불필요한 비교를 수반한다는 사실을 알 수 있다. 예를 들어서 Iteration 3에서 $T[\texttt{SA}[l] : n]$=aa$이고, $T[\texttt{SA}[r] : n]$=abaabbaa$이다. 이 두 접미사는 각 aa$는 Iteration 2에서, abaabbaa$는 Iteration 1에서 P와 한번씩 비교된 적이 있다. 그리고 이 비교를 통해 P는 aa$와 앞에서부터 2개 문자가 같고, abaabbaa$와는 앞에서 1개 문자가 같다는 것을 알 수 있었다. 다시 말해 이 두 접미사는 앞에서부터 비교했을 때 $\min\{2, 1\} = 1$개 문자인 a를 공통접두사로 가지고 있으며, 접미사들은 사전순으로 정렬되어 있으므로 구간 $[l, r]$ 내의 모든 접미사들은 a로 시작한다. 따라서 패턴문자열 P와 현재의 중앙점에 있는 접미사 aabaabbaa$를 비교할 때 1번째 문자는 건너뛰어도 무방하다는 사실을 알 수 있다.

$\mathsf{lcp}_l = lcp(P, T[\texttt{SA}[l] : n])$, $\mathsf{lcp}_r = lcp(P, T[\texttt{SA}[r] : n])$라고 하자. $k = \min\{\mathsf{lcp}_l, \mathsf{lcp}_r\}$라고 할 때, $lcp(P, T[\texttt{SA}[p] : n]) \geq k$ 임이 보장되므로 $k+1$번째 문자부터 비교를 수행하면 된다. 만약 P가 현재 중앙점인 접미사보다 작다면 $r \leftarrow p$로 업데이트되므로 lcp_r을 현재 단계에서 비교한 문자의 수에 따라 업데이트 해주고, 반대의 경우 lcp_l를 같은 방식으로 갱신해주면 된다. 이 기법은 탐색 속도를 비약적으로 상승시켜주지만 이진 탐색이 어느 한 방향으로만 진행되는 경우에는 $k = \min\{\mathsf{lcp}_l, \mathsf{lcp}_r\}$이 감소하지 않으므로 각 단계에서 비교하는 문자 수는 여전히 $\mathcal{O}(m)$이므로, 전체 탐색의 시간 복잡도는 $\mathcal{O}(m \lg n)$ 이다.

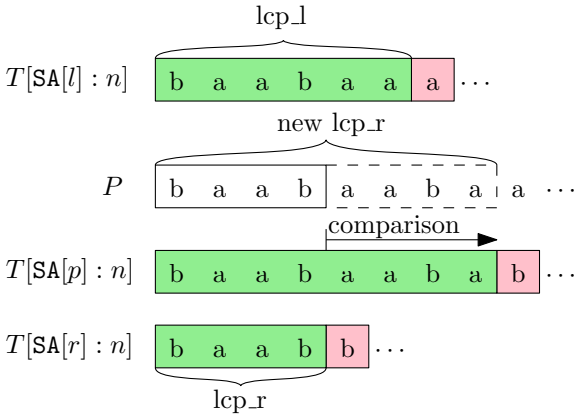

Figure 5: 현재 탐색 구간 $[l, r]$에 대해 P 와 $T[\texttt{SA}[l] : n]$, $T[\texttt{SA}[r] : n]$ 간의 최장공통접두사 길이 lcp_l, lcp_r를 각각 갖고 있다면 $\min\{lcp\_l, lcp\_r\}$ 만큼의 접두사에 대한 불필요한 비교를 줄일 수 있다.

7.1.3 탐색 방법 : $\mathcal{O}(m + \lg n)$

만약 이전 단계에서 P와 비교했던 T의 접미사 $T[\texttt{SA}[p] : n]$의 최장공통접두사의 길이를 알고 있다고 해보자. 이 때, $T[\texttt{SA}[p] : n]$와 현재 단계에서 비교할 접미사 $T[\texttt{SA}[p'] : n]$의

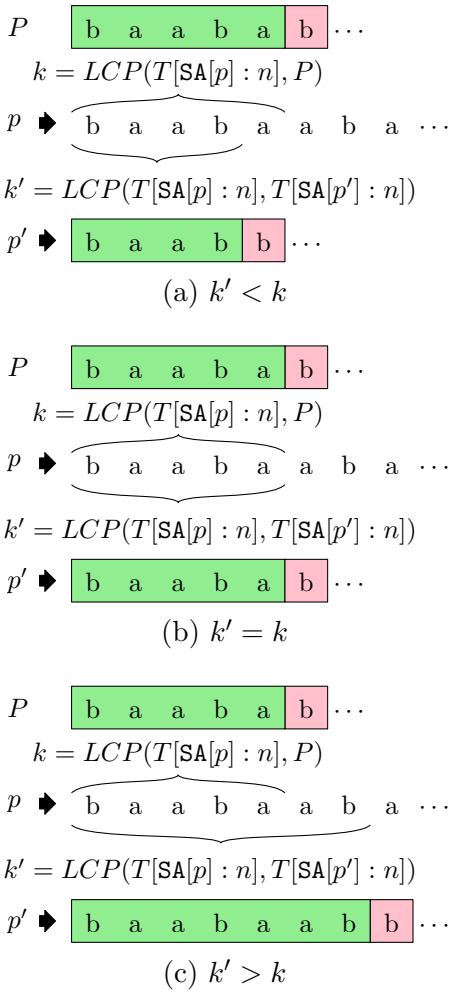

Figure 6: $T[\texttt{SA}[p] : n]$와 P 및 $T[\texttt{SA}[p'] : n])$ 각각에 대한 최장공통접두사 길이의 관계에 따른 경우의 수

최장공통접두사의 길이를 알 수 있다면 P와 $T[\texttt{SA}[p'] : n]$를 문자열의 처음부터 비교할 필요가 없다. 우선 $P > T[\texttt{SA}[p] : n])$였다고 가정하자. 그리고 $k = LCP(P, T[\texttt{SA}[p] : n])$, $k' = LCP(T[\texttt{SA}[p] : n], T[\texttt{SA}[p'] : n])$이라고 하자. 그렇다면 다음 3가지 경우의 수가 생긴다.

1. $k' < k$: P는 $T[\texttt{SA}[p'] : n]$보다 작다.
2. $k' = k$: P와 $T[\texttt{SA}[p'] : n]$의 최장공통접두사를 $k + 1$번째 문자부터 차례로 비교하여 구한 후 결과에 따라 분기한다.
3. $k' > k$: P는 $T[\texttt{SA}[p'] : n]$보다 크다.

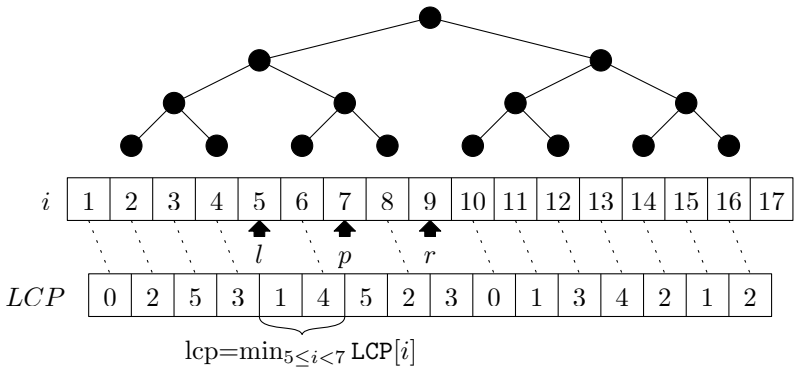

Figure 7: 이진 탐색 과정에서 중앙점 p가 주어지면 그에 대응하는 l, r은 유일하다

문제 7.1.2 만약 임의의 두 접미사 간의 최장공통접두사의 길이를 $\mathcal{O}(1)$ 시간에 구할 수 있다면 접미사 배열을 이용하여 $\mathcal{O}(m + \lg n)$ 시간에 문자열 매칭을 수행할 수 있음을 보이세요.

탐색 시간을 줄이기 위한 기본적인 아이디어는 이진 탐색 대상이 되는 구간 $[l, r]$이 접미사 배열 상의 임의의 구간이 아니라는 점이다. 또한 $p = \lfloor \frac{i+j}{2} \rfloor$가 주어지면 그에 대응하는 탐색 구간 $[l, r]$은 유일하게 주어진다. 그림 7은 이러한 예를 나타낸다. 예를 들어 $p = 7$로 주어졌다면 이전 단계에서는 반드시 중앙점이 5였고, 패턴문자열이 이 접미사보다 컸기 때문에 $l \leftarrow 5$로 업데이트 된 것이다. 이 두 접미사 $T[\texttt{SA}[l]]$과 $T[\texttt{SA}[p]]$의 공통접두사의 길이는 LCP배열에서 해당되는 구간의 최소값이므로 이 예제에서는 그 값은 1이다.

기본적인 원리는 이진 탐색 과정을 트리로 나타내는 경우 어떤 노드의 자식노드는 부모노드의 구간의 한쪽 끝을 공유하는 부분구간에 대응되고, 또한 어떤 구간 $[l, r]$이 두 구간 $[l, c]$와 $[c, r]$로 분리되는 경우 $[l, r]$ 내의 LCP 최소값은 각 구간의 최소값 중 작은 값에 해당한다는 사실을 이용하는 것이다. 각각의 지점 $1 < p < n$에 대응하는 탐색구간 $[l, r]$이 유일하므로 이 경계값을 각각 $L(p), R(p)$라고 하자. 여기서 두 개의 배열 $LCP\_L[p]$, $LCP_R[p]$를 다음과 같이 정의한다.

$$\begin{aligned} LCP\_L[p] &= lcp(T[\texttt{SA}[p] : n], T[\texttt{SA}[L(p)] : n]) \\ LCP\_R[p] &= lcp(T[\texttt{SA}[p] : n], T[\texttt{SA}[R(p)] : n]) \end{aligned} \tag{1}$$

두 배열 LCP_L, LCP_R은 다음과 같이 선형시간에 계산할 수 있다. 먼저 이진 탐색 트리 상에서 말단노드부터 시작한다. 말단노드의 경우에는 $l + 1 = p = r - 1$인 경우인데, $LCP\_L[p] = LCP[l] = LCP[p - 1]$, $LCP_L[p] = LCP[p]$임을 쉽게 확인할 수 있다. p의 왼쪽과 오른쪽 자식에 해당하는 위치가 각각 x, y인 경우 $LCP\_L[p] = \min\{LCP\_L[x], LCP\_R[x]\}$, $LCP_R[p] = \min\{LCP_L[y], LCP_R[y]\}$와 같이 채워나간다.

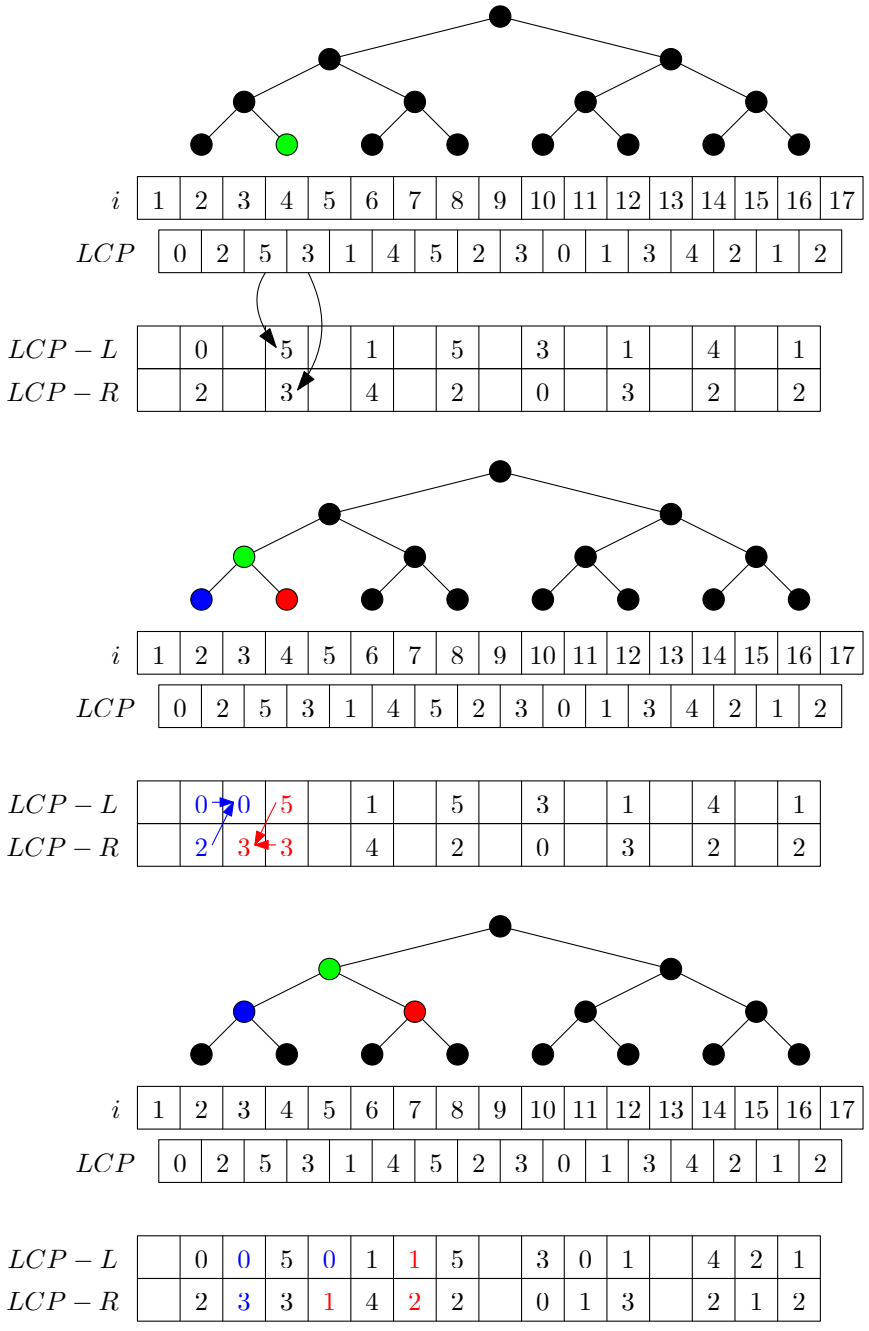

Figure 8: LCP_L, LCP_R 배열을 계산하는 과정

문제 7.1.3 위에서 계산한 LCP_L, LCP_R 배열을 이용하여 $\mathcal{O}(m + \lg n)$ 시간에 문자열 매칭을 수행할 수 있음을 보이세요.

7.1.4 접미사 배열 $\mathcal{O}(n)$ 구축 알고리즘: DC3 Algorithm

앞서 살펴본 것과 같이 접미사 트리를 이용하면 $\mathcal{O}(n)$ 시간에 접미사 배열을 계산할 수 있다. 그러나 인간 유전체 서열과 같이 대용량의 문자열을 다루는 경우 접미사 트리를 구성하기 위한 메모리 자체가 충분하지 않을 경우가 있다. 이러한 경우 접미사 트리를 거치지 않고 바로 접미사 배열을 계산해야 할 필요가 있다. 본 절에서는 접미사 트리를 거치지 않고 재귀적인 방법을 통해 접미사 배열을 바로 계산하는 방법을 알아본다.

기본적인 아이디어는 문자열의 사전 순 비교의 특성을 이용하여 전체 접미사 순위를 결정하기 위해서 점차 길이가 축소된 변환 문자열을 이용하여 재귀적으로 비교를 수행한다는 점이다. 두 접미사 $T[i:n]$와 $T[j:n]$의 크기를 비교할 때 나타날 수 있는 경우의 수는 다음과 같다.

1. $T[i] < T[j]$: 접미사 $T[i:n]$가 더 작다.
2. $T[i] = T[j]$: 두 접미사 $T[i:n]$과 $T[j:n]$의 순서는 $T[i+1:n]$과 $T[j+1:n]$의 순서와 일치한다.
3. $T[i] > T[j]$: 접미사 $T[j:n]$가 더 작다.

먼저 주어진 문자열 T를 구성하는 모든 문자들을 크기 순으로 정렬하여 1부터 차례대로 번호를 부여한다. 만약 중복되는 번호가 없다면 정렬이 완료된 것이다. 중복되는 번호가 있다면, 변환된 T의 연속된 3개의 문자를 묶어서 하나의 문자로 취급하는 새로운 문자열을 구성한다. 시작 위치에 따라 $T_1 = (T[1]T[2]T[3])(T[4]T[5]T[6])\cdots$, $T_2 = (T[2]T[3]T[4])(T[5]T[6]T[7])\cdots$, $T_3 = (T[3]T[4]T[5])(T[6]T[7]T[8])\cdots$로 나눌 수 있다. 각 문자열의 마지막 문자가 3의 배수가 아닌 경우 0을 패딩해준다.

예를 들어 $T = 112121110$이라면, $T_1 = (112)\ (121)\ (110)$, $T_2 = (121)\ (211)\ (100)$, $T_3 = (212)\ (111)\ (000)$이다. T_2와 T_3을 이어 붙여서 하나의 문자열을 만든 후, 재귀적으로 반복하면 T_2와 T_3를 구성하는 각 문자의 순위가 완전히 정해진다. 이제 T_1의 문자와 T_2, T_3의 문자를 병합 정렬한다. T_1의 각 문자 $t_{1,i}$는 $t_{1,i}[1]T_2[i]$ 또는 $t_{1,i}[1]t_{1,i}[2]T_3[i]$로 표현할 수 있고, T_2의 각 문자 $t_{2,i}$는 $t_{2,i}[1]T_3[i]$, T_3의 각 문자 $t_{3,i}$는 $t_{3,i}[1]t_{3,i}[2]T_2[i+1]$로 표현할 수 있으므로 이를 이용하여 병합 정렬을 수행하여 T_1, T_2, T_3를 구성하는 모든 문자들의 순위를 구할 수 있다.

이제 T의 각 $3i+j$번째 문자를 $T[3i+j]T_{j+1}[i+1]$ 꼴로 표현을 하여 정렬을 하면 각 위치로부터 시작하는 접미사의 순위를 구할 수 있다. 구해진 순위를 이용하여 접미사 배열을 구성한다.

문제 7.1.4 *DC3 Algorithm*의 시간 복잡도가 $\mathcal{O}(n)$ 임을 보이세요.

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

7.2 버로우즈-휠러 변환과 FM-index

FM-index는 문자열의 버로우즈-휠러 변환의 특성을 이용한 색인 방법으로 접미사 배열보다 더욱 공간효율적인 자료구조이다. 특히 FM-index는 압축된 공간 상에서도 문자열 매칭을 수행할 수 있도록 변형할 수 있기 때문에 메모리 공간이 한정되어 있거나, 주기억장치와 보조기억장치 간의 성능 차이가 매우 큰 경우 유용하게 사용된다.

7.2.1 버로우즈-휠러 행렬과 버로우즈-휠러 변환

주어진 문자열 T를 회전-쉬프트하여 얻을 수 있는 모든 문자열을 나열한 후 사전순으로 정렬을 하여 순서대로 나열한 후, 각 문자열의 마지막 글자를 차례대로 취한 것을 T의 버로우즈-휠러 변환 $\texttt{BWT}(T)$ 라고 한다. (앞으로 문맥이 확실한 경우 T를 표기에서 생략하도록 한다) 이 때, 문자열들을 정렬하여 순서대로 나열한 것을 $n \times n$ 행렬과 비슷하다고 하여 버로우즈-휠러 행렬이라 한다. 버로우즈-휠러 변환은 버로우즈-휠러 행렬의 마지막 열(column)이다.

i	$\texttt{SA}[i]$	Burrows-Wheeler Matrix
1	19	$baabaabbbabaabaabb
2	12	aabaabb$baabaabbbab
3	2	aabaabbbabaabaabb$b
4	15	aabb$baabaabbbabaab
5	5	aabbbabaabaabb$baab
6	10	abaabaabb$baabaabbb
7	13	abaabb$baabaabbbaba
8	3	abaabbbabaabaabb$ba
9	16	abb$baabaabbbabaaba
10	6	abbbabaabaabb$baaba
11	18	b$baabaabbbabaabaab
12	11	baabaabb$baabaabbba
13	1	baabaabbbabaabaabb$
14	14	baabb$baabaabbbabaa
15	4	baabbbabaabaabb$baa
16	9	babaabaabb$baabaabb
17	17	bb$baabaabbbabaabaa
18	8	bbabaabaabb$baabaab
19	7	bbbabaabaabb$baabaa

Figure 9: 문자열 T =baabaabbbabaabaabb$의 버로우즈-휠러 행렬

주어진 문자열이 종료문자 \$로 끝나며, 이 종료문자가 문자열 상의 다른 어느 곳에도 출현하지 않는다면 버로우즈-휠러 변환으로부터 다시 원래 문자열을 복원할 수 있다. 버로우즈-휠러 변환 $\texttt{BWT}(T)$로부터 T를 복원하는 과정은 버로우즈-휠러 행렬을 다음과 같이 재구성함으로써 얻을 수 있다.

$n \times n$ 행렬의 마지막 열(n번째 열)에 $\texttt{BWT}(T)$가 있다고 가정하자. 이 행렬의 각 행을 독립적인 문자열로 보고 정렬을 한다. 그리고 $n-1$번째 열에 버로우즈-휠러 변환을 덧붙이고 다시 정렬을 수행한다. 이 과정을 마지막까지 반복하면 버로우즈-휠러 행렬을 얻을 수 있고, 원 문자열 T는 종료문자 \$가 가장 첫 문자가 되도록 회전-쉬프트된 형태로 첫 행에 존재하게 된다.

위의 버로우즈-휠러 변환에 관한 설명은 $\mathcal{O}(n^2)$ 공간이 사용되는 방법으로 개념만을 직관적으로 설명하기 위한 방법이다. 종료문자 \$가 유일하다면 버로우즈-휠러 행렬의 각 행은 문자열 T의 접미사를 순서대로 나열한 것과 대응된다. 버로우즈-휠러 변환은 각 순위 접미사의 앞 글자를 차례대로 나열한 것과 같다.

정의 7.2.1 종료문자 \$를 가지는 길이 n의 문자열 T의 버로우즈 휠러 변환 $\texttt{BWT}$은 길이 n의 문자열로, 다음과 같이 정의된다:

$$\texttt{BWT}[i] = T[(\texttt{SA}[i] + n - 2 \mod n) + 1]$$

■

인덱스를 0-based로 표현되는 경우(즉, $T[0:n-1]$로 표현하는 경우)에는 다음과 같이 표현할 수 있다.

$$\texttt{BWT}[i] = T[\texttt{SA}[i] + n - 1 \mod n] \tag{3}$$

접미사 배열을 $\mathcal{O}(n)$ 시간에 계산할 수 있으므로 버로우즈-휠러 변환도 $\mathcal{O}(n)$ 시간에 계산 가능하다.

7.2.2 버로우즈-휠러 변환을 이용한 문자열 매칭: FM-index

원 문자열 T 상에서 어느 한 위치 j에 출현한 문자열 $T[j]$는 버로우즈-휠러 행렬의 각 열마다 정확히 하나씩 대응된다. 버로우즈-휠러 행렬의 마지막 열의 $T[j]$와 정확히 대응되는 첫 열의 $T[j]$를 함수로 표현한 것을 LF-mapping이라 부른다.

버로우즈-휠러 행렬의 마지막 행에서 i번째 출현한 α의 원 문자열 T 상에서의 위치를 j라고 하자. 버로우즈-휠러 행렬의 첫 행에서 i번째 출현한 α의 원 문자열 T 상에서의 위치 역시 정확히 j 임을 확인할 수 있다.

이를 보다 형식적으로 표현해보자. 먼저 $T[j] = \alpha$라고 하자. 접미사 $T[j:n]$가 α로 시작하는 접미사 중 i번째 작다고 하자. 이것이 의미하는 바는 버로우즈-휠러 행렬 상에서 α로 시작하는 행 중 i번째 행에 대응되는 문자열이 $T_j = T[j]T[j+1]\cdots T[n-1]\$T[1]\cdots T[j-1]$이라는 것이다. $T[j'] = \alpha$인 어떤 j'에 대하여 $T_{j'} = T[j']T[j'+1]\cdots T[n-1]\$T[1]\cdots T[j'-1]$과 T_j의 순서는 $T[j] = T[j'] = \alpha$이기 때문에 $T_{j'+1}$와 T_{j+1}간의 순서 관계와 정확히 일치한다.

즉, α로 시작하면서 T_j보다 작은 문자열들의 개수는 마지막 문자가 α이면서 T_{j+1}보다 작은 문자열의 개수와 같다. 이를 이용하면 마지막 열의 i번째 행과 대응되는 첫 열의 행을 나타내는 LF-맵핑 $\texttt{LF}(i)$을 다음과 같이 기술할 수 있다.

$$\texttt{LF}(i) = \texttt{Occ}(c_i) + \texttt{rank}_{c_i}(i) \tag{4}$$

단, $c_i = \texttt{BWT}[i]$, $\texttt{Occ}(c)$는 c보다 작은 문자들 개수의 합, $\texttt{rank}_c(i)$는 버로우즈-휠러 변환의 접두사 $\texttt{BWT}[1:i]$에서의 문자 c의 출현 횟수이다.

버로우즈-휠러 행렬은 접미사 배열과 직접적으로 대응이 되므로, 버로우즈-휠러 변환 상에서의 문자열 매칭 탐색 결과는 접미사 배열의 경우와 마찬가지로 하나의 구간 $[i_s, i_e]$로 표현할 수 있다. 어떤 패턴 P에 대한 탐색 구간이 $[i_s, i_e]$로 주어졌다면, αP에 대한 탐색 구간 $[i'_s, i'_e]$ 은 버로우즈-휠러 변환 상의 구간 $[i_s, i_e]$에서 가장 처음, 그리고 가장 마지막에 출현하는 α의 LF-mapping과 같으며, 이는 다음과 같이 계산할 수 있다.

$$\begin{aligned} i'_s &= \texttt{Occ}(\alpha) + \texttt{rank}_\alpha(i_s - 1) + 1 \\ i'_e &= \texttt{Occ}(\alpha) + \texttt{rank}_\alpha(i_e) \end{aligned} \tag{5}$$

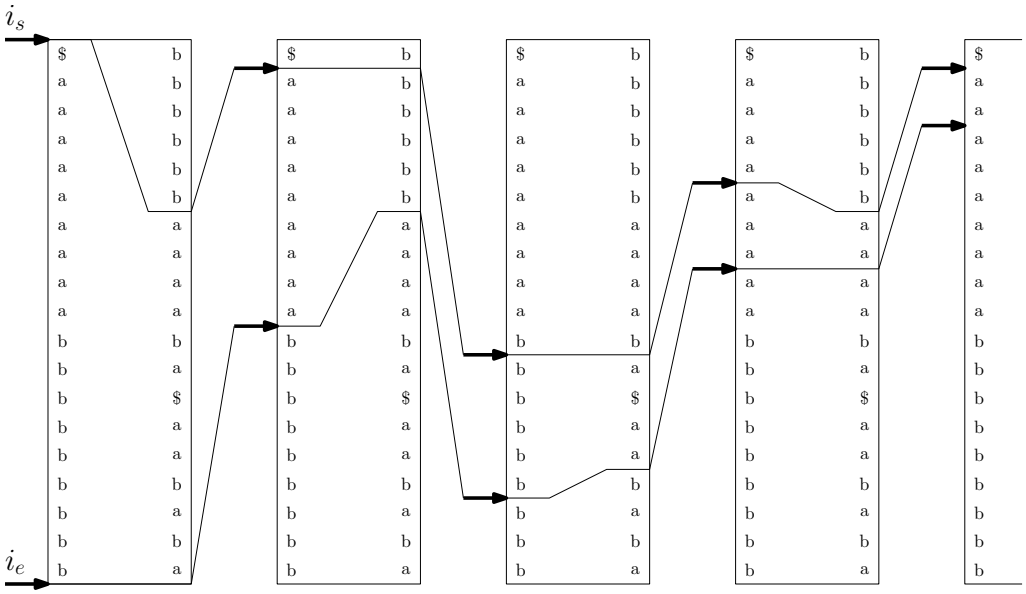

Figure 10: FM-index로 문자열 매칭을 수행하는 과정

두 함수 `Occ`와 `rank`만 구현할 수 있다면 문자열 매칭을 위 식을 이용하여 효율적으로 수행할 수 있다. `Occ`는 배열로 간단히 구현할 수 있으며, `rank`는 문자열을 구성하는 알파벳의 개수나 분포에 따라 다소 구현에 차이가 있을 수 있다. 위 식의 `Occ`와 `rank`을 이용한 문자열 색인을 FM-index라고 한다.

7.3 웨이블릿 트리 (Wavelet Tree)

고정된 비트열 상에서의 Rank 질의는 $n + o(n)$ bit 공간을 사용해서 $O(1)$ 시간에 수행할 수 있다. 이를 이용하면 임의 알파벳 Σ으로 구성된 어떤 문자열 T 상에서의 Rank 질의를 다음과 같이 구현할 수 있다. 알파벳 집합 Σ를 서로소인 두 부분집합 Σ_0와 Σ_1로 분할한다. 이 때, $\Sigma_0 \cup \Sigma_1 = \Sigma$, $\Sigma_0 \cap \Sigma_1 = \phi$를 만족하도록 한다. 주어진 T의 각 문자 $T[i]$에 대하여, $T[i] \in \Sigma_0$이면 $B[i] = 0$, 아니면 $B[i] = 1$이 되도록 비트열 B를 구성한다. 알파벳 부분집합 Σ_0, Σ_1로 이루어진 T의 Subsequence T_0, T_1를 각각 현재 노드의 왼쪽 자식, 오른쪽 자식으로 만든 후 각 자식 노드에 대하여 같은 작업을 알파벳 크기가 1이 될 때까지 반복한다.

T 상에서의 Rank 질의 $\texttt{rank}_c(i; T)$는 다음과 같이 수행한다. 먼저 현재 노드에서 $c \in \Sigma_j$라고 하자. 먼저 $i' = \texttt{rank}_j(i; B)$를 구한다. 만약 현재 노드에서 $|\Sigma| = 1$이라면 i'이 정답이다. 그렇지 않다면 $j = 0$인 경우 왼쪽 노드로, $j = 1$인 경우는 오른쪽 노드로 이동한다. 이동한 노드에서 $\texttt{rank}_c(i', T_j)$를 수행한다.

문제 7.3.1 웨이블릿 트리를 이용하면 Σ 상의 문자열 T에 대한 `rank` 질의를 $\mathcal{O}(n \lg \Sigma)$ *bit* 공간을 이용하여 $\mathcal{O}(\lg n)$ 시간에 수행할 수 있음을 보이세요.

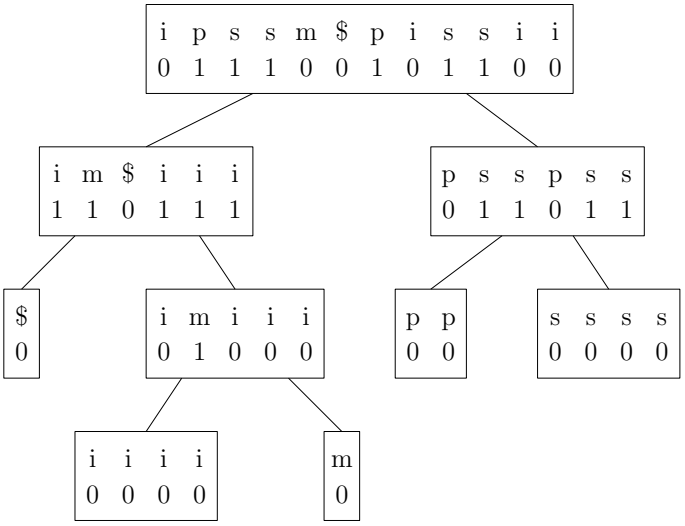

Figure 11: 문자열 $T =$`ipssm$pissii`의 웨이블릿 트리