Lecture 6: 문자열을 위한 자료구조 (I)*

6.1 문자열 탐색과 Trie 구조

문자열은 전산에서 가장 기본적인 자료 형태 중 하나이다. 문자열 탐색에 있어서 가장 기본이 되는 자료구조 형태는 Trie이다. 본 장에서는 Trie 및 이와 관련된 몇 가지 응용 문제를 알아보고자 한다.

특히 엄청난 길이의 생물학 서열을 다루는 Bioinformatics에서 이 문자열 검색은 아주 중요한 문제이다. 그리고 요즘의 hot issue인 Big Data 연구는 주로 문서 기반의 연구가 주를 이루고 있는데 문서는 전형적인 서열(sequence) 자료의 일종이다.

앞으로 다룰 주제는 주로 주어진 문자열이 거의 변화가 없는 고정적인 상황에서, 예를 들어 생물체의 유전정보와 같이, 다양한 검색작업을 돕기 위하여 매우 효율적인 전처리 자료구조에 관한 것들이다. 따라서 이런 자료구조는 문자열의 구성성분이 시간에 따라 자주 바뀐다든지, 즉 어떤 부분구간에 대하여 다른 문자열의 삽입, 또는 삭제가 번번히 일어나는 경우에는 매우 비효율적인 자료구조가 될 수 있다.

문제 6.1.1 문자열 집합 $S \subset \Sigma^*$가 있다. 주어지는 질의 문자열 $\mathbf{q} \in \Sigma^*$에 대하여 $\mathbf{q} \in S$인지를 확인하는 방법은? 예) $S = \{app, apple, apply, tomato, tomas\}$, $\mathbf{q}_1 = application$, $\mathbf{q}_2 = tomato$.

문제 6.1.2 문자열 집합 $S = \{S_i\}$가 있다 (단 $1 \le i \le k$). S_i의 길이가 n_i일 때, 다음 질문에 답하시오.

1. *Trie*의 높이는?
2. *Trie*의 최대 노드 수는?
3. *Trie*의 말단 노드 수는?

6.2 Trie 구현에서 노드 표현 구조

Trie의 각 노드는 다음 노드로 가는 링크들을 가지고 있다. 이러한 링크들을 어떻게 구현할 것인지 생각해보자.

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

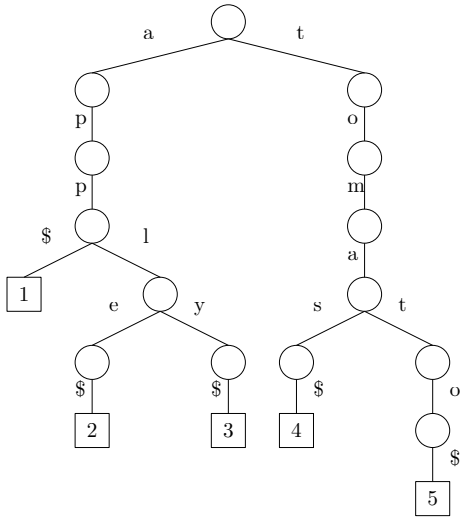


Figure 1: 문자열 집합 S ={`app`,`apple`,`apply`,`tomato`,`tomas`}에 대한 Trie.

구현 방법	Time for `lookup()`	Space
Array		
Sorted Array		
Linked List		
Binary Search Tree		
Hash Table		
Weighted BST		

6.2.1 Weighted Binary Search Tree

branch를 이진 탐색 트리(binary search tree)로 구성하되 각 노드들 아래에 있는 말단 노드의 개수를 가중치로 하여 구성할 수 있다.

문제 6.2.1 시간 복잡도가 $O(|P| + \lg k)$가 됨을 증명하시오. *(hint: BST*에서 자식 노드로 내려올 때마다 감소되는 말단 노드의 개수를 살펴보자*)*

6.2.2 Array + wBST

말단 노드가 $|\Sigma|$개 미만이 되기 직전까지 상단 트리로 구성하여 배열로 표현하고, 각각의 하위 노드는 $|\Sigma|$ 이하 개의 말단노드를 가지도록 트리를 나누어 Weighted BST로 표현할 수 있다.

문제 6.2.2 이 경우 공간 및 시간 복잡도는?

6.2.3 Ternary Trie

Ternary Trie는 이진 탐색 트리와 Trie의 특성을 섞은 자료구조이다. 각 노드는 기준 문자와 함께 최대 3개의 자식노드를 가진다. 왼쪽과 오른쪽 자식노드를 루트로 가지는 부분트리는 각각 현재 문자가 기준 문자보다 작거나 큰 문자열들의 Ternary Trie이다. 가운데 자식노드를 루트로 가지는 부분트리는 기준문자로 시작하는 문자열들의 집합에서 첫 문자를 제외한 나머지 문자열을 취했을 때의 문자열 집합에 대한 Ternary Trie이다.

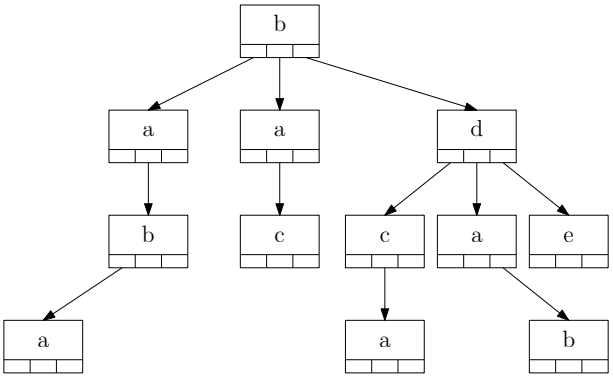


Figure 2: 문자열 집합 S ={aa,ab,bac,ca,da,db,e}에 대한 Ternary Trie

문제 6.2.3 총 n개의 문자열이 색인된 *Well-balanced Ternary Trie*에서 길이 m의 질의 문자열에 대한 탐색 시간은 $O(m + \lg n)$임을 보이시오. *(Hint:* 탐색은 왼쪽/오른쪽 자식노드로 이동하여 탐색대상을 줄이는 것과 아래로 이동하여 질의 문자열을 처리하는 연산의 조합으로 이루어진다*)*

6.2.4 Double Array Trie

만약 문자열 집합에 삽입, 삭제 등의 연산이 거의 일어나지 않고 탐색을 주로 수행하는 경우라면 Trie를 보다 공간 효율적으로 표현하기 위한 방법으로 double array를 이용하는 방법이 있다. 배열을 이용하면 $O(1)$ 시간에 현재 노드로부터 주어진 문자에 해당하는 다음 다음 노드로 이동할 수 있지만 현재 문자열 다음에 나타나지 않는 문자가 많을수록 사용되지 않고 낭비되는 공간들이 많다. Double Array 표현법의 기본적인 아이디어는 서로 다른 노드들이 링크를 표현하는 배열을 공유함으로써 낭비되는 배열의 빈 공간들을 최대한 활용하는 것이다.

우선 3개의 배열을 이용하여 표현하는 방법을 살펴보자. 그림 3은 어떤 문자열에 대한 Trie를 3개의 배열로 나타내는 원리를 설명하고 있다. v_2는 다음에 오는 문자로 $와 b를 가지며, 각각에 해당하는 노드는 $v_1$2와 $v_1$7이다. v_3는 다음 문자로 $와 c를 가지며, 각각 $v_1$5와 $v_1$6에 대응된다. 아래 그림은 이러한 구조를 3개의 배열로 나타낸 예를 보여준다. 다음 노드에 대한 링크는 Next 배열에 저장되는데, v_2는 Next[7]부터 Next[10]까지의 구간 중 Next[7]과 Next[9]를 사용하고, v_3는 Next[8]부터 Next[11]까지의 구간 중 두 개의 요소인 Next[8]과 Next[11]을 사용한다. Base 배열은 각 노드가 사용하는 Next 배열 상의 구간의 시작점을 나타낸다. 예를 들어, Base[2]는 v_2가 사용하는 구간 {7,8,9,10}의 시작점 7을 나타내고 있다. Check는 Next 배열의 각 요소가 어느 노드에

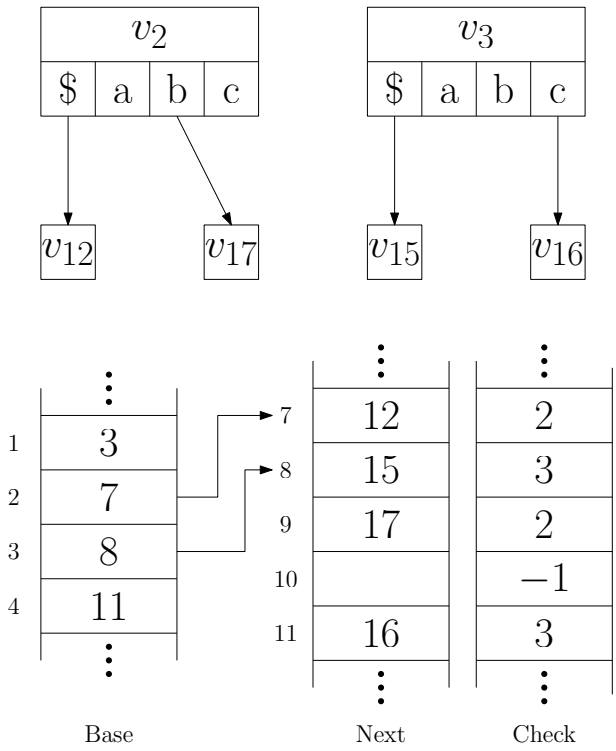


Figure 3: Trie의 Triple Array 표현 방법

의해 사용되고 있는지를 나타낸다. 예를 들어 Check[8]=3인데 Next[8]이 v_3에 의해 사용되고 있다는 것을 의미한다.

이를 정리하면 다음과 같다. 배열 `Base`, `Next`, `Check`를 이용하여 각 노드 s에서 문자 c가 들어왔을 때 다음 노드 t로의 매핑을 다음과 같이 구성할 수 있다.

$$t = \texttt{Next}[\texttt{Base}[s] + c] \text{ if } \texttt{Check}[\texttt{Base}[s] + c] == s$$

탐색을 수행할 때에는 현재 노드 v에서 다음 문자 α가 들어왔을 때 $\texttt{Check}[\texttt{Base}[v] + \alpha] = v$인지를 확인한다. 만일 그렇다면 α에 해당하는 분기는 노드 $\texttt{Next}[\texttt{Base}[v] + \alpha]$를 가리킨다. 만약 $\texttt{Check}[\texttt{Base}[v] + \alpha] \neq v$이라면 v에는 α에 해당하는 분기가 없다는 것을 의미한다.

또한 위 관계를 잘 살펴보면 $\texttt{Check}[v]$는 노드 $\texttt{Next}[v]$의 부모 노드를 가리키고 있음을 알 수 있다. 따라서 만일 Triple array로 표현된 Trie를 재구성하고자 하는 경우 이러한 관계를 이용하면 보다 효율적으로 Trie 구조를 복원할 수 있다. 물론, 루트노드에서부터 재구성할 수도 있지만 별도의 추가적인 정보가 없다면 $\texttt{Base}[v]$부터 $\texttt{Base}[v] + \sigma - 1$까지의 `Check` 배열의 값을 확인해야 한다는 점을 생각해보자.

문제 6.2.4 문자열 집합 $S = \{a, ab, ac, ba, bd\}$에 대한 *trie*를 그리고, *triple array* 구조로 표현해보자. 말단 노드를 표현하는 방법에는 여러 가지가 있는데, 여기서는 종료문자 $\$$가 주어지면 말단노드라고 가정하고, 해당 위치에는 그 문자열의 고유번호를 적도록 한다. 고유번호는 문자열 집합을 사전순으로 나열했을 때 각 문자열의 순위를 사용하도록 하자. 예를 들어 ab의 고유번호는 2이다.

i	Base	Next	Check
1	0	2	1
2	3	3	1
3		4	2
4			
5	9	5	2
6		6	2
7			
8			
9		2	5
10			
11			
12			

여기서 Next 배열 없이 다음과 같이 표현할 수도 있다.

$$t = \texttt{Base}[s] + c \text{ if } \texttt{Check}[\texttt{Base}[s] + c] == s$$

Double array 표현에서는 $\texttt{Check}[v]$가 v의 부모를 그대로 가리킨다. Double array로 표현된 Trie 상에서 탐색을 수행하는 경우, 노드 v에서 문자 α에 해당하는 노드로 이동할 때 $\texttt{Check}[\texttt{Base}[v] + \alpha] = v$인지를 확인해야 하는데, 이는 다시 말하면 노드 $u = \texttt{Base}[v] + \alpha$의 부모가 v인지를 확인하는 것이다. 또한 부모 노드로 이동하고자 하는 경우에는 곧바로 $w = \texttt{Check}[v]$로 이동하면 된다. Double array 표현은 Triple array 표현에 비하여 배열을 하나 적게 쓴다는 이점 뿐만 아니라 부모노드로의 이동 역시 $O(1)$ 시간에 수행할 수 있다는 장점이 있다.

문제 6.2.5 문자열 집합 $S = \{a, ab, ac, ba, bd\}$를 *double array* 구조로 표현해보자.

i	Base	Check
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

6.3 압축 트라이 (Compressed Trie)

Trie에서 자식노드로 가는 link가 하나 뿐인 내부 노드는 사실상 의미가 없다. 이들을 제거하여 내부 노드가 항상 2개 이상의 branch를 가지도록 만든 트리구조를 Compressed Trie라고 한다.

문제 6.3.1 문자열 집합 $S = \{app, apple, apply, tomato, tomas\}$에 대한 *Compressed Trie*를 그려보자.

문제 6.3.2 문자열 개수가 n일 때 *compressed trie*에서 *internal node*의 개수는?

우리는 앞서 정의한 Trie의 다양한 응용의 예에 대하여 살펴보기로 하자.

문제 6.3.3 *(자동 완성)* $S = \{app, apple, apply, tomato, tomas\}$가 있다. $\mathbf{q} = toma$로 시작하는 문자열은 무엇인가?

문제 6.3.4 *(Predecessor)* $S = \{app, apple, apply, tomato, tomas\}$가 있다. 이 때, $\mathbf{q} = application$보다 크지 않은 최대의 문자열은 무엇인가?

문제 6.3.5 *(문자열 해싱)* $S = \{app, apple, apply, tomato, tomas\}$가 있다. 이들에게 번호 $1, \cdots |S|$를 중복되지 않게 부여하는 방법은?

문제 6.3.6 *(문자열 정렬)* $S = \{app, apple, apply, tomato, tomas\}$가 있다. 이들에게 번호 $1, \cdots |S|$를 사전순으로 부여하는 방법은?

문제 6.3.7 *(LCP)* 문자열 집합 $S = \{app, apple, apply, tomato, tomas\}$가 있다. 이 때, $\mathbf{q} = tomatsauce$와 가장 긴 접두사 *(prefix)*를 공유하는 문자열은 무엇이며, 최장 공통 접두사의 길이는?

6.4 다중 패턴 매칭과 Aho-Corasick 알고리즘

어떤 텍스트 문자열 T에 다른 패턴 문자열 P가 출현했는지는 찾는 문제는 문자열 매칭 또는 패턴 매칭 문제라고 한다. 만약 찾으려는 문자열의 수가 여러 개인 경우, 즉 문자열 집합 $\mathbf{P} = \{P_1, P_2, \cdots, P_m\}$에 속한 문자열들 중 어떤 문자열이 T의 어느 위치에 출현했는지를 찾고자 하는 경우 이를 다중 패턴 매칭 문제라고 한다. 욕설 필터링 시스템과 같이 미리 정의된 욕설 단어 집합이 있을 때 사용자가 입력하는 대화에 대하여 욕설이 존재하는지를 찾아내는 문제가 전형적인 다중 패턴 매칭 문제의 예시라고 할 수 있다.

길이가 $n+m$인 두 문자열의 문자열 매칭 문제는 $O(n+m)$ 시간에 찾을 수 있지만 각각의 $P_i \in \mathbf{P}$에 대하여 독립적인 문자열 매칭을 수행하는 것은 매우 비효율적이다. 이 때 패턴 문자열 집합 $\mathbf{P}$의 문자열 길이 합이 $M = \sum_i |P_i|$일 때, 이를 $O(M)$ 시간에 전처리하여 적절한 자료구조를 구성한 다음 매칭 대상이 되는 길이 n의 텍스트 문자열이 입력되면 $O(n+M+k)$ 시간에 매칭을 수행할 수 있다(단, k는 패턴 문자열의 출현 횟수).

Aho-Corasick 알고리즘은 다중 패턴 매칭 문제를 위한 대표적인 알고리즘으로 Trie를 기반으로 하여 각 노드에 대하여 추가적인 간선을 연결하여 오토마타 형태의 자료구조를 구성하여 매칭을 효율적으로 수행할 수 있도록 한다. 이 때, Trie는 종료문자 \$를 추가하지 않고 구성한다. 문자열 X에 해당하는 노드는 다음과 같은 링크를 추가적으로 지닌다.

1. Failure Link: 자신을 제외한 X의 접미사 중 Trie 내에 대응하는 노드가 존재하는 가장 긴 접미사에 대응하는 노드
2. Output Link: 자신을 제외한 X의 접미사 중 패턴 집합에 속한 가장 긴 접미사에 대응하는 노드. 즉 패턴 집합에 속하는 노드 중 현재 노드로부터 Failure link를 계속 타고 갔을 때 처음으로 만나는 노드.

그림 4에 패턴 문자열 집합 $\mathbf{P} = \{\texttt{ab}, \texttt{abcc}, \texttt{bca}, \texttt{bccc}, \texttt{cab}, \texttt{cc}, \texttt{ccb}\}$에 대한 오토마타인데, $\mathbf{P}$에 속하는 문자열에 대응되는 노드들은 짙은 회색으로 표시되어 있고, failure link는 빨간색, output link는 파란색 화살표로 표시되어 있다.

Failure link는 다음 문자에 대한 분기가 없을 때, 문자열을 처음부터 새로 탐색하지 않고 중간부터 재개할 수 있도록 해준다. 예를 들어 Link가 없는 Trie 구조를 이용하여 문자열 `abcab` 상에서 패턴 문자열들을 탐색해야 한다면 `abcab`의 모든 접미사, 즉 `abcab`, `bcab`, `cab`, `ab`, `b`에 대하여 루트 노드부터 탐색을 수행해야 하므로 $O(n^2)$ 시간이 소요된다. 그런데 만약 `abcab`를 탐색하는 도중에 `abc`까지 탐색하고 그 다음 문자인 `a`에 대한 분기가 없을 때, `bc`에 해당하는 노드로 이동할 수 있어서 자연스럽게 `bca`에 대한 탐색을 수행할 수 있다면 매번 처음부터 수행하지 않아도 된다면 많은 시간을 절약할 수 있다. 이러한 역할을 Failure link가 하는 것이다.

Output link는 `abcc`, `cc`와 같이 한 패턴 문자열이 다른 패턴 문자열의 부분 문자열인 경우 이를 모두 report할 수 있도록 하는 노드를 연결해주는 간선이다.

6.4.1 탐색 알고리즘

텍스트 문자열이 주어지면, 일반적인 Trie 탐색과 같이 각 문자에 대하여 분기를 타고 다음 노드로 이동한다. 앞서 말했듯이 Failure Link는 다음 문자에 대한 분기가 없을 때

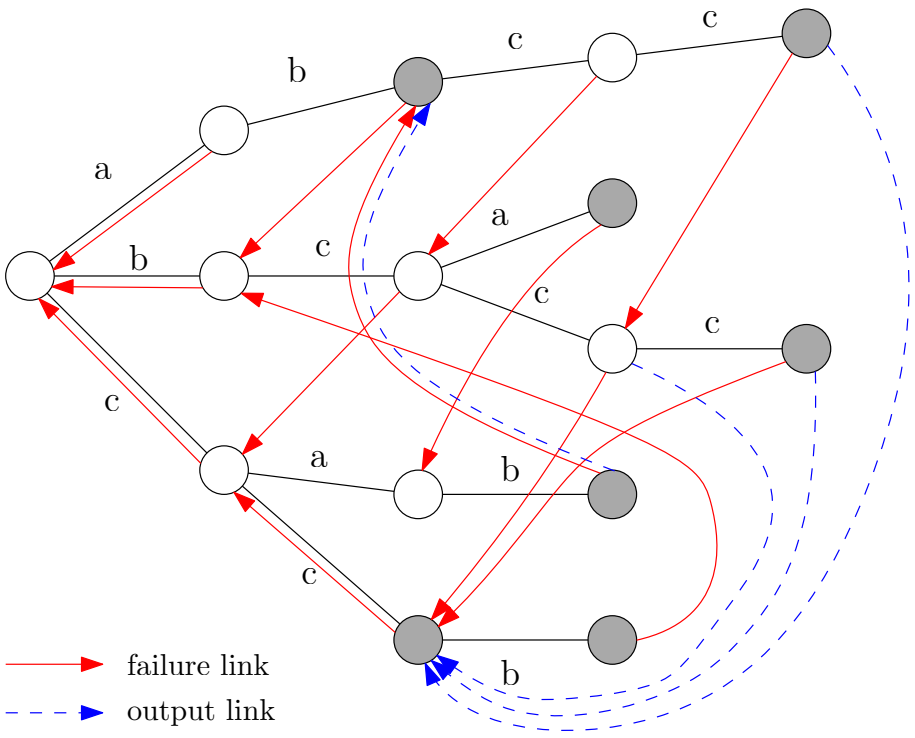


Figure 4: 패턴 집합 $\mathbf{P} = \{\texttt{ab}, \texttt{abcc}, \texttt{bca}, \texttt{bccc}, \texttt{cab}, \texttt{cc}, \texttt{ccb}\}$에 대한 Aho-Corasick 오토마타

사용하는 링크이며, Output link는 이동 후에 현재 위치에 존재하는 패턴을 report하기 위한 링크이다. 탐색 과정을 서술하면 다음과 같다.

1. 만약 현재 노드에서 $T[i]$에 해당하는 분기가 없다면, $T[i]$에 대한 분기가 있을 때까지 Failure Link를 타고 이동한다. 만약 루트 노드에서도 분기가 없으면 다음 문자로 넘어간다.
2. 만약 분기가 있다면 해당하는 노드로 이동한다.
3. 이동한 노드가 $\mathbf{P}$에 속한 노드이면 이 노드를 report한다.
4. 만약 output link가 있다면 output link를 따라 연결된 노드들을 차례대로 report한다.

그림 5는 패턴 집합 $\mathbf{P} = \{\texttt{aabc}, \texttt{abc}, \texttt{bba}, \texttt{bca}, \texttt{c}, \texttt{cb}\}$에 대한 Aho-Corasick 오토마타 상에서 텍스트 문자열 $T =$aabca에 대한 다중 패턴 매칭을 수행하는 과정을 나타낸다.

정리 6.4.1 이미 구축된 *Aho-Corasick* 오토마타를 이용하면 길이 n의 문자열에 대하여 $O(n + k)$ 시간에 다중 패턴 매칭 탐색을 수행할 수 있다. *(단, k는 패턴 문자열의 총 출현 횟수)*

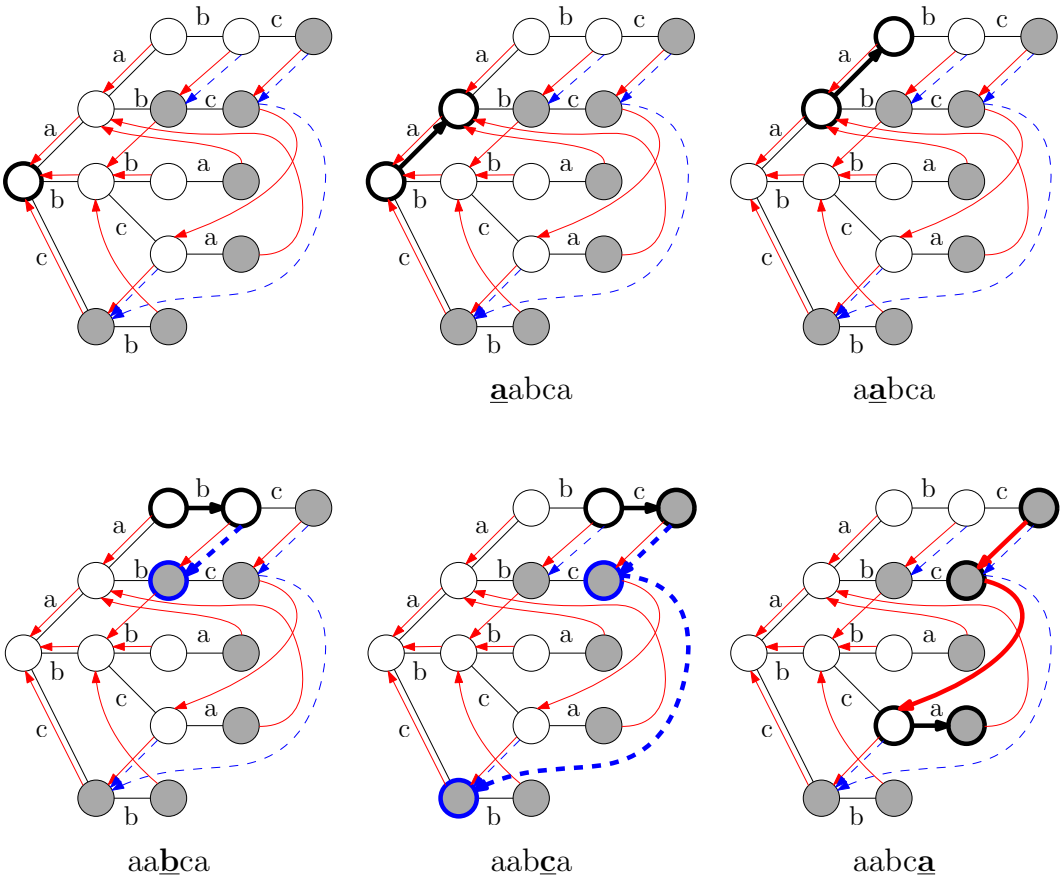


Figure 5: 패턴 집합 $\mathbf{P} = \{\texttt{aabc}, \texttt{abc}, \texttt{bba}, \texttt{bca}, \texttt{c}, \texttt{cb}\}$에 대한 Aho-Corasick 오토마타 상에서 텍스트 문자열 $T =$aabca에 대한 매칭을 수행하는 과정

6.4.2 Link 구축 알고리즘

이제 Aho-Corasick 오토마타를 구축하는 방법에 대하여 살펴보도록 한다.

먼저 패턴 문자열 집합에 해당하는 Trie를 구성한 후, BFS 탐색을 통해 노드들을 순차적으로 순회한다.

각 노드 v에 대하여 α를 노드 v에 해당하는 문자열의 마지막 문자라고 하자. 즉, α는 v의 부모노드에서 v로 오는 분기에 해당되는 문자이다. v를 방문하는 시점에서 v의 부모 노드는 루트 노드이거나 Failure link가 이미 만들어져 있는 상태이다. v의 부모가 루트노드라면 v의 Failure link는 루트노드이고, 만약 그렇지 않다면 부모의 Failure link를 타고간 후 α에 대한 분기를 타고 간 노드가 v의 Failure link가 된다. 만약 해당 노드에 α에 대한 분기가 없다면 다시 Failure link를 α에 대한 분기가 존재할 때까지 타고 간다.

현재 노드 v에 대해 Failure link를 만들었다면 Output link를 만든다. 노드 v의 Failure link가 노드 u를 가리킨다고 하자. 만일 노드 u가 패턴 집합 내에 속한다면 v의 Output link는 u이다. 만일 그렇지 않다면 v의 Output link는 u의 Output link로 설정한다. 다음 그림에 이러한 과정이 설명되어 있다.

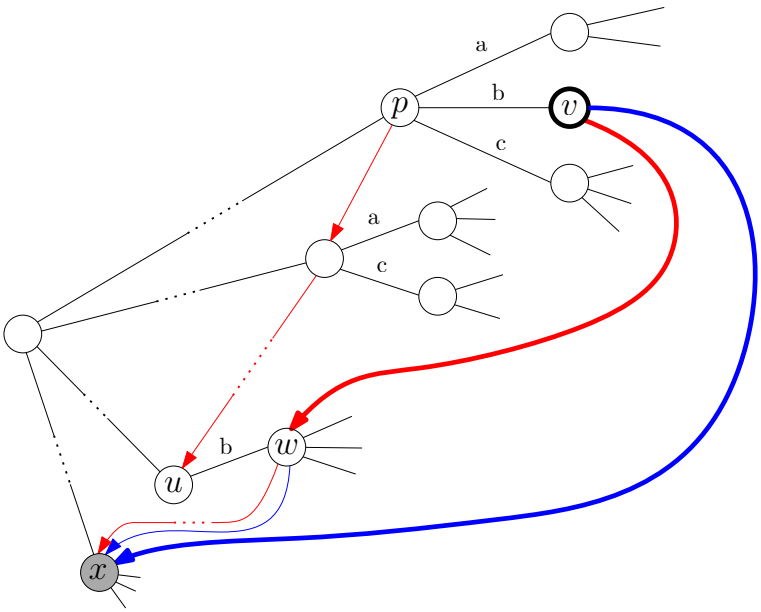


Figure 6: Failure link와 Output link를 생성하는 과정.

BFS를 통해 노드 v에 도달할 차례가 되면, 우선 v의 부모노드인 p의 Failure link를 타고 문자 b에 대한 분기가 나올 때까지 반복해서 이동한다. 그림에서는 처음으로 b에 대한 분기가 나오는 노드는 u인데, u에서 b에 대한 자식노드인 w가 v의 Failure link가 된다. 이 시점에서 w는 x를 향해 Output link를 가지고 있는데, v에도 마찬가지로 x를 향한 Output link를 생성해주면 된다.

정리 6.4.2 패턴 집합 $\mathbf{P} = \{P_i\}$에 대한 *Aho-Corasick* 오토마타는 $O(M)$ 시간에 구축할 수 있다. 단, $M = \sum_i |P_i|$.

6.5 문자열 매칭과 접미사 트리(Suffix Tree)

먼저 문자열 매칭 문제에 대하여 formal하게 정의하고자 한다. 길이가 각각 n, m인 두 문자열 $T, P \in \Sigma^*$에 대하여 P가 T 상에 등장하는 위치를 찾는 문제를 문자열 매칭이라고 한다. 문자열 매칭 문제는 $O(n+m)$ 시간에 해결할 수 있음은 이미 오래 전에 증명되었다. 응용 분야에 따라서 텍스트 T가 고정되어 있는 상태에서 다수의 패턴에 대한 문자열 매칭을 수행하는 일이 잦은 경우가 있다. 이 때, 매번 T를 스캔하는 것보다는 T를 전처리를 해놓은 후 P에 의존적인 시간에 문자열 매칭을 수행하는 것이 효율적이다. 접미사 트리는 이를 위한 대표적인 자료구조 중의 하나이다.

접미사 트리는 고정된 텍스트 T에 대한 문자열 매칭을 위한 아주 효율적인 자료구조이다. T의 모든 접미사에 대한 compressed trie를 T의 **접미사트리**라고 한다.

문제 6.5.1 문자열 T=*baabab*의 접미사를 모두 나열하시오.

문제 6.5.2 문자열 T=*baabab*의 접미사 트리를 그리시오.

6.6 접미사 트리 구성을 위한 선형 우코넨(Ukkonen) 알고리즘

접미사 트리를 위에서 설명한 방식에 따라 구축을 하게 되면 접미사 길이의 합 만큼의 시간이 필요하므로 시간 복잡도는 $O(n^2)$이다. 본 절에서는 접미사 트리를 선형시간에 구축하는 방법을 알아본다.

Ukkonen의 알고리즘에서는 접미사 구축 과정을 n개의 단계로 구성해 놓았다. i번째 단계에서는 $T[1:i]$ 의 모든 접미사들을 트리에 추가하는 작업을 수행한다. 즉, i번째 단계에는 $1 \leq j \leq i$에 대해서 접미사 $T[j:i]$의 추가 작업을 수행한다. 예를 들어 T=baabab에 대한 접미사 트리는 다음과 같은 과정을 거쳐 구축된다.

1. 1단계의 1번째 접미사 $T[1:1]$=b 를 추가한다.
2. 2단계의 1번째 접미사 $T[1:2]$=ba 를 추가한다.
3. 2단계의 2번째 접미사 $T[2:2]$= a 를 추가한다.
4. 3단계의 1번째 접미사 $T[1:3]$=baa 를 추가한다.
5. 3단계의 2번째 접미사 $T[2:3]$= aa 를 추가한다
6. 3단계의 3번째 접미사 $T[3:3]$= a 를 추가한다.
7. $\cdots$

Ukkonen 알고리즘의 중간 과정에서 구성되는 트리는 종료문자(\$)가 없는 implicit suffix tree이다. 각 단계의 세부단계를 구성하는 접미사 추가 작업을 자세히 설명하려고 한다. 우선 현재까지 완성된 implicit suffix tree에 i번째 단계의 j번째 접미사 $T[j:i] = x\alpha$를 추가한다고 가정해보자. 그럼 다음 3가지 경우로 나뉘어 진다.

1. x가 말단 노드인 경우: 해당 말단 노드로 가는 에지에 α를 추가한다.
2. $x\alpha$에 대응하는 경로가 없는 경우: x에 해당하는 내부 노드에서 α로 가는 branch를 추가한다. (이 때, x에 해당하는 지점이 내부 노드가 아니라 에지 위라면 해당 에지를 분할하여 내부 노드를 새로 만든다.)
3. $x\alpha$에 대응하는 경로가 있는 경우: 아무 작업도 하지 않는다.

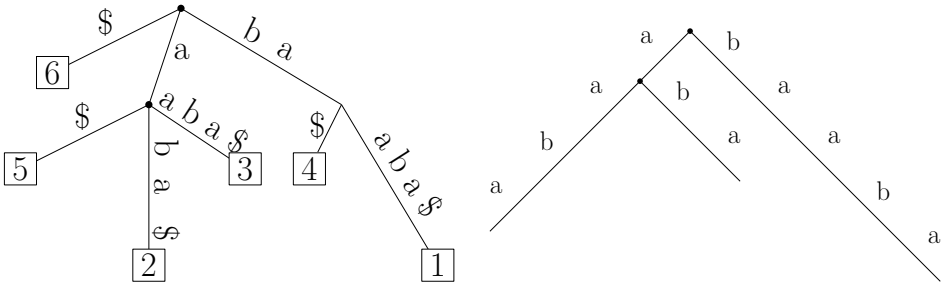


Figure 7: 문자열 T=baaba의 suffix tree와 implicit suffix tree.

문제 6.6.1 문자열 T=*baabab*의 *implicit suffix tree*를 그리고 T=*baaba*의 *implicit suffix tree*와의 차이점을 설명해보시오.

Ukkonen 알고리즘에서 i단계가 끝나면 $T[1:i]$의 implicit suffix tree가 만들어진다. Ukknoen이 제시한 선형 시간 알고리즘의 핵심은 $T[1:i]$의 implicit suffix tree로부터 $T[1:i+1]$의 implicit suffix tree 를 $O(1)$ amortized time에 구하는 것에 있다.

6.6.1 Suffix Link

정의 6.6.1 *(접미사 링크)* 두 문자열 αX, X에 대응하는 내부 노드가 각각 v, u일 때, v의 접미사 링크 $s(v)$는 u로 향하는 포인터이다.

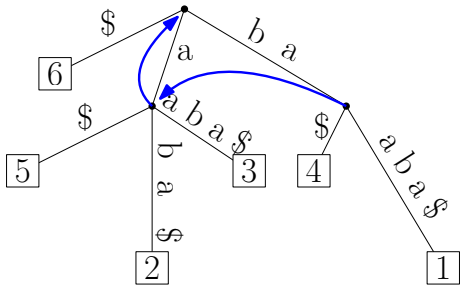
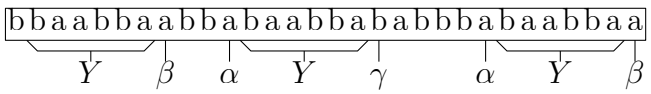
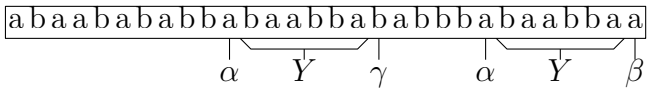


Figure 8: 접미사 링크의 예



(a) Case 1



(b) Case 2

Figure 9: 접미사 αX를 추가할 때 내부 노드가 생성되었을 때, 접미사 X를 추가할 때 발생할 수 있는 두 가지 경우

문제 6.6.2 앞에서 그린 문자열 T=`baabab`의 *implicit suffix tree*에서 접미사 링크를 표시하시오.

소정리 6.6.1 만약 i단계의 j번째 접미사 αX를 추가하는 과정에서 새로운 내부 노드 v가 생성되었다면, *(i)* X*(=*같은 단계의 $j+1$번째 접미사*)*에 대응하는 지점에 이미 내부노드가 있거나 *(ii)* 접미사 X를 추가할 때 반드시 생성된다.

$X = Y\beta$라고 하자. 만약 αX를 추가할 때 새로운 내부 노드가 생성되었다면 $\alpha Y\beta$가 현재까지 추가된 문자열 중 처음 등장했다는 것이다. 한편, αY와 Y는 $i-1$번째 단계에서 이미 추가가 되었기 때문에 내부 노드 또는 에지 상에 반드시 존재한다는 점을 상기하자. 여기서 말단 노드에 β가 추가되는 것이 아니라 내부노드가 생성되었다는 것은 이전에 어떤 문자열 $\alpha Y\gamma$가 존재해서 αY 지점에서 β와 γ의 분기가 새로이 생성되었다는 것이다. 이로부터 현재 단계의 implicit 접미사 트리 상에 $Y\gamma$에 대응되는 지점이 존재한다는 것을 알 수 있다. 따라서 만약 $Y\beta$가 이전에 출현한 적이 있었다면 이미 Y에서 β와 γ로 분기되는 내부노드가 이미 존재하고, $Y\beta$가 출현한 적이 없었다면 다음 단계에서 새로이 생성되게 된다.

따름정리 6.6.1 i단계의 j번째 접미사가 추가되는 과정에서 생성된 내부 노드 v는 같은 단계 $j+1$번째 접미사에 대응되는 지점으로의 접미사 링크를 가진다.

따름정리 6.6.2 $T[1:i]$의 *implicit suffix tree*에서 어떤 문자열 $x\alpha$에 대응하는 내부 노드 v가 있다면 x에 대응하는 내부 노드도 반드시 존재한다.

이러한 사실들을 이용하면 i번째 단계의 j번째 접미사를 추가한 후에 같은 단계의 $j+1$번째 접미사를 추가하는 작업을 수행할 때, $T[j+1:i]$를 루트노드부터 따라 내려가는 것이 아니라 접미사 링크를 이용하여 좀 더 효율적으로 수행할 수 있다. 구체적인 과정은 다음과 같다.

1. 현재 지점에서 가장 가까운 상위 노드 v로 이동한다. 이 때 노드 v부터 현재지점까지의 레이블을 y라고 하자.
2. v에서 접미사 링크를 타고 $s(v)$로 이동한다.
3. $s(v)$에서 문자열 y에 해당하는 에지를 타고 내려온다.

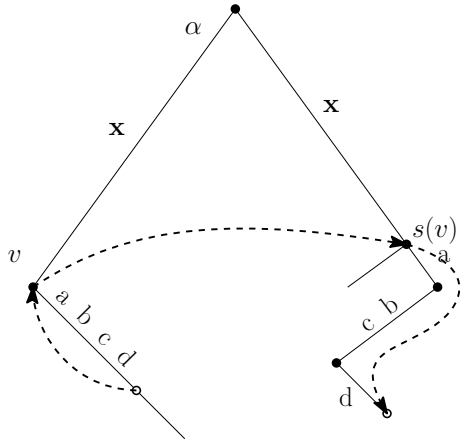


Figure 10: 접미사 링크를 이용하여 트리에 $j+1$번째 접미사를 추가하는 과정

6.6.2 Skip/Count 트릭

접미사를 추가하는 과정에서 에지를 타고 올라가거나 내려가는 작업을 해야 한다. 이 때, 각 문자를 하나씩 확인하면서 움직이는 대신 노드 단위로 이동해도 무방하다. 노드간에 이동을 할 때 에지 레이블 길이만큼의 시간이 아니라 상수 시간이 소요되도록 하는 것이다. 에지 레이블을 문자열로 표현하는 대신 T 상의 인덱스로 표현하고 이동 시에는 branch를 제외하고는 인덱스의 증감으로만 나타낼 수 있다.

문제 6.6.3 앞에서 그린 문자열 T=`baabab`의 *implicit suffix tree*에서 에지 레이블을 T 상의 인덱스 번호로 나타내시오.

문제 6.6.4 T=`baabab`의 *implicit suffix tree*에서 루트 노드에서 `baab`를 타고 내려가는 과정을 나타내보자. 또한 이 상태에서 접미사 링크를 이용하여 `aab`로 이동하는 과정을 설명해보자.

문제 6.6.5 *Skip/Count* 트릭을 이용할 때 i번째 단계를 전체를 수행하는데 걸리는 시간은?

이를 계산할 때 가장 관건이 되는 것은 접미사 링크를 타고 건너간 후에 에지를 타고 내려가는 작업을 몇 번이나 해야 하는지이다. 같은 단계의 각 접미사를 추가할 때마다 에지를 타고 올라가는 작업과 접미사 링크를 타고 이동하는 작업은 1 번씩 수행된다. 이 작업들이 수행될 때마다 현재 위치의 depth는 그대로이거나 각각 1 감소한다. 한 단계 전체를 아울러 살펴보면 depth가 최대 $2i$만큼 감소하게 된다. 접미사 링크를 타고 $s(v)$로 건너가서 에지를 타고 내려가는 작업을 한 번 수행할 때마다 depth는 1씩 증가한다. 그러나 i번째 단계에서 트리의 최대 depth는 i를 넘을 수가 없으므로, 에지를 타고 내려가는 횟수는 $3i$을 넘지 않는다는 것을 알 수 있다.

6.6.3 불필요한 작업 줄이기 기법

먼저 i번째 단계에서의 말단 노드 v를 살펴보도록 하자. 이 말단 노드 v는 어떤 j에 대하여 $T[j:i]$와 대응됨은 자명하다. $i+1$번째 단계의 j번째 접미사 $T[j:i+1]$이 추가될 때 말단 노드 v에서는 규칙 1번에 따라 말단 에지 레이블에 $T[i+1]$이 추가되는 작업 밖에 이루어 지지 않는다. 즉, 한번 말단노드가 된 노드는 접미사 트리가 완성되는 순간까지 말단 노드로 남아있으면서 에지 레이블에 각 단계별로 문자가 하나씩 추가되는 양상을 보인다. i번째 단계에서 $T[j:i]$에 대한 말단 노드가 새로 생성이 되었다면 $i+1$번째 단계에서는 $T[j:i]$에 대한 처리는 각 말단 에지마다 별도로 할 필요 없이 모든 말단 에지가 레이블 인덱스의 마지막 값을 공유하도록 하여 한번에 처리하면 된다.

또 다른 케이스로는 i단계의 j번째 접미사에 대응하는 경로가 이미 트리 상에 존재하는 경우이다 (접미사 추가 규칙 3번). i번째 단계에서 $T[j:i+1]$이 트리 상에 이미 존재한다면, $T[j+1:i+1]$도 반드시 존재한다. 따라서 한번 이러한 케이스가 발견되면 같은 단계에서 남은 접미사들에 대한 처리는 더 이상 할 필요가 없다.

6.6.4 우코넨 알고리즘의 시간 복잡도

1번 규칙에 따른 말단 에지의 업데이트는 단계마다 $O(1)$ 시간에 처리가 되므로 전체 단계에 걸쳐 총 $O(n)$ 시간이 걸린다. 2번 규칙이 적용될 때마다 내부노드가 1개씩 증가하므로 총 $O(n)$번 적용된다. 따라서 2번 규칙에 의한 소요 시간은 전체 단계에 걸쳐 총 $O(n)$ 이다. 3번 규칙은 한 단계마다 최대 1번만 수행되므로 전체 알고리즘에 걸쳐 총 $O(n)$ 시간이 소요된다.

정리 6.6.1 *Ukkonen* 알고리즘을 이용하여 길이가 n인 문자열의 접미사 트리를 $O(n)$ 시간에 구성할 수 있다.

문제 6.6.6 앞서 설명한 규칙들을 이용하여 T=*mississippi*의 접미사 트리를 직접 만들어보자.

6.7 접미사 트리 응용

6.7.1 String Matching Problem

길이 n, m의 문자열 T, P에 대하여 P의 출현 횟수는 $O(n)$ 시간의 전처리를 거쳐 $O(m)$ 시간에 가능하다. 모든 출현 위치를 찾는 것은 $O(m+k)$ 시간이 걸린다 (k는 출현 횟수).

문제 6.7.1 T=*baababa*에서 패턴 P =*aba*를 접미사 트리를 이용하여 찾아보자.

6.7.2 Generalized Suffix Tree and its Application

접미사 트리는 하나의 문자열에 대한 트리이지만 문자열 집합으로 쉽게 확장할 수 있다. 문자열 집합 $\mathbf{T} = \{T_1, \cdots, T_k\}$가 주어졌을 때, 문자열 $T_1\$_1T_2\$_2 \cdots T_k\$_k$에 대한 접미사 트리를 구성할 수 있다. 그리고 $\$_i$ 이후에는 무조건 말단 노드가 위치하도록 한다. 이를 일반화된 접미사 트리(generalized suffix tree)라고 한다. 원래의 접미사 트리에서 말단 노드가 단일 문자열 상의 위치였다면 일반화된 접미사 트리에서는 해당 접미사가 속한 문자열 번호와 그 문자열 상의 위치가 같이 저장이 된다.

문제 6.7.2 두 문자열 T_1 = *banana*, T_2 = *sabana*에 대한 접미사 트리를 그려보자.

문제 6.7.3 앞의 일반화된 접미사 트리 상에서 *Longest Common Substring*을 찾는 방법은?

6.7.3 Matching Statistics

문자열 T의 각 $1 \leq i \leq n$에 대하여 $T[i:n]$의 prefix 중 패턴 P의 일부와 일치하는 가장 긴 문자열의 길이를 $M(i)$라고 하자.

예를 들어 T =ababaabaa, P =aba라고 하면,

i	1	2	3	4	5	6	7	8	9
$T[i]$	a	b	a	b	a	a	b	a	a
$M(i)$	3	2							

T =abcxabcde, P =iabciabcdxcde인 경우에는,

i	1	2	3	4	5	6	7	8	9
$T[i]$	a	b	c	x	a	b	c	d	e
$M(i)$	3								

문제 6.7.4 P의 접미사 트리가 주어졌을 때 $M(i)$를 구하는 방법과 그 시간 복잡도를 설명하시오. *[Hint: Suffix Link*를 이용하자*]*

6.8 Lowest Common Ancestor 문제

어떤 두 노드에 대하여 최저 공통 조상 (lowest common ancestor)를 찾는 문제는 트리 구조의 자료 형태에서 매우 빈번하게 사용된다. 특히 접미사 트리에서의 최저 공통

조상은 공통 문자열을 찾기 위해 매우 유용하게 활용할 수 있다. 본 절에서는 최저 공통 조상 문제를 찾는 문제를 범위 최소값(Range minimum)을 찾는 문제로 전환시켜 이를 $O(1)$ 시간에 푸는 방법을 알아보도록 한다.

6.8.1 Range Minimum Query

배열 $A = x_1, x_2, \cdots, x_n$ 이 있다. 두 인덱스 l, r 이 주어졌을 때

$$\arg\min_{l \le i \le r} A[i]$$

를 찾고자 한다.

문제 6.8.1 다음 RMQ 질의에 답해보자.

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

l	r	$RMQ_A(l, r)$
2	6	5
1	2	
2	4	
4	8	
6	8	

6.8.2 $O(1)$ solution for RMQ

RMQ 질의는 $O(n \lg n)$ 공간을 이용하여 $O(1)$ 시간에 계산할 수 있다. 각 지점 i 에 대하여 i 에서 $i + 2^j - 1$ 까지를 포함하는 길이 2^j 인 구간에서의 RMQ 질의 결과를 모두 미리 계산하여 lookup table L 에 저장한다.

문제 6.8.2 위 배열 A 에 대한 테이블을 완성하시오.

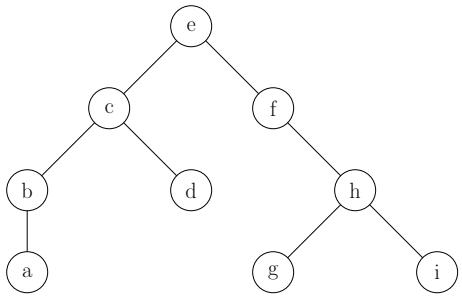
$L[2^j][i]$	1	2	3	4	5	6	7	8
2^0								
2^1								
2^2								
2^3								

이 테이블을 이용하면 어떤 l, r이 입력되더라도 서로 다른 2개의 미리 계산된 구간으로 나타낼 수 있으며, 두 값 중 더 작은 원소를 가리키는 인덱스를 반환하면 된다.

6.8.3 LCA 문제와 RMQ 문제의 연관 관계

먼저 이진트리(binary tree)에 대하여 LCA 문제를 RMQ 문제로 전환하여 살펴보도록 하자. 먼저 대상이 되는 트리의 각 노드에 루트로부터의 거리를 적어넣는다. 이후 트리를 중위 탐색을 하면서 각 노드로부터 루트까지의 거리를 방문하는 순서대로 적어넣는다.

문제 6.8.3 아래 트리에 대하여 루트와의 거리를 중위 탐색 순서대로 나열하시오.



node									
dist									

위 결과를 살펴보면 두 노드의 LCA는 각 노드에 대응하는 인덱스에 대한 dist배열에서의 RMQ 질의 결과와 대응된다는 점을 확인할 수 있다.

문제 6.8.4 트리의 *euler tour representation*을 이용하면 LCA를 위한 RMQ 질의를 $O(n)$ 공간에 수행할 수 있다. 그 방법에 대하여 생각해보자.

문제 6.8.5 접미사 트리에서 두 말단 노드의 LCA가 의미하는 점을 생각해보자.