# 문자열 재구성 알고리즘 및 멱승문제 응용
# (A String Reconstruction Algorithm and Its Application to Exponentiation Problems)

심 정 섭 † 이 문 규 † 김 동 규 ††
(Jeong Seop Sim) (Mun-Kyu Lee) (Dong Kyue Kim)

**요 약** 대부분의 문자열 문제들과 이들에 대한 알고리즘들은 패턴 매칭, 데이타 압축, 생물정보학 등의 분야에 응용되어 왔다. 그러나 문자열 문제와 암호화 문제의 관련성에 대한 연구는 거의 진행되지 않았다. 본 논문에서는 다음과 같은 문자열 재구성 문제들에 대해 연구하고 이 결과들이 암호학에 응용될 수 있음을 보인다.

유한 알파벳으로 구성된 길이 $n$인 문자열 $x$와, 길이 $k(\le n)$ 이내의 문자열의 집합 $W$가 주어졌을 때, 첫 번째 문제는 $W$ 내의 문자열들 중 일부 문자열들을 최소의 회수로 연결하여 $x$를 재구성할 수 있는 연결 순서를 찾는 문제이다. 이 문제에 대해 $O(kn+L)$-시간 알고리즘을 제시한다. 이때, $L$은 $W$ 내의 모든 문자열들의 길이의 합을 표시한다. 두 번째 문제는 첫 번째 문제의 동적 버전이며 이에 대해 $O(k^3n+L)$-시간 알고리즘을 제시한다. 마지막으로 암호학과 관련된 멱승문제와 위에 제시된 재구성 문제들과의 관련성을 보이고 멱승문제를 해결하는 새로운 알고리즘을 제시한다.

**키워드** : 문자열재구성, 접미사트리, 멱승문제

***Abstract*** Most string problems and their solutions are relevant to diverse applications such as pattern matching, data compression, recently bioinformatics, and so on. However, there have been few works on the relations between string problems and cryptographic problems. In this paper, we consider the following string reconstruction problems and show how these problems can be applied to cryptography.

Given a string $x$ of length $n$ over a constant-sized alphabet $\Sigma$ and a set $W$ of strings of lengths at most an integer $k(\le n)$, the first problem is to find the sequence of strings in $W$ that reconstruct $x$ by the minimum number of concatenations. We propose an $O(kn+L)$-time algorithm for this problem, where $L$ is the sum of all lengths of strings in a given set, using suffix trees and a shortest path algorithm for directed acyclic graphs. The other is a dynamic version of the first problem and we propose an $O(k^3n+L)$-time algorithm. Finally, we show that exponentiation problems that arise in cryptography can be successfully reduced to these problems and propose a new solution for exponentiation.

**Key words** : string reconstruction, suffix trees, exponentiation problems

· 본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구 결과로 수행되었음(IITA-2008-C1090-0801-0003)
· 이 논문은 2007년도 정부(교육과학기술부)의 재원으로 국제과학기술협력재단의 지원을 받아 수행된 연구임(No. K20717000007070B010000710)
· 본 연구는 한국과학재단 특정기초연구(R01-2006-000-10957-0) 지원으로 수행되었음

† 종신회원 : 인하대학교 컴퓨터정보공학부 교수
jssim@inha.ac.kr
mklee@inha.ac.kr
†† 종신회원 : 한양대학교 전자컴퓨터공학부 교수
dqkim@hanyang.ac.kr

논문접수 : 2008년 5월 9일
심사완료 : 2008년 8월 23일

Copyright@2008 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 시스템 및 이론 제35권 제10호(2008.10)

## 1. Introduction

A string is a sequence of some characters from an alphabet. There are so many string related problems. Among them, we are interested in problems to reconstruct a string using one or more relatively short strings. This kind of problems and their solutions are applicable to diverse applications such as pattern matching, data compression, and recently molecular biology or bioinformatics.

There have been vigorous works on this field of problems. If a string $x$ can be reconstructed by one time repetition of a string $y$ where $y$ is a non empty string, that is, $x=y^2(=yy)$, we call $x$ a *square* (or *tandem repeat*). Also, if $x=y^3(=yyy)$, we call $x$ a *cube*. For example, aa, abab and abcabc are all squares and aaa and ababab are cubes. There are several $O(n\log n)$-time algorithms for finding all the repetitions in a string [1-3].

More generally, if a prefix of a string $x$ can be reconstructed by one or more concatenations of a string $y$, $y$ is called a *period* of a string $x$. In other words, $y$ is a period of a string $x$ if $x$ can be written as $x=y^k y'$ where $k \geq 1$ and $y'$ is a prefix of $y$. For example, if $x=$ abcabcab, then abc is a period of $x$. Using the preprocessing of the Knuth-Morris-Pratt algorithm [4], we can find all periods of $x$ in linear time.

If a string $x$ can be reconstructed by some concatenations and superpositions of a string $y$, $y$ is called a *cover* of $x$. For example, if $x=$ ababaaba, then aba is a cover of $x$. Apostolico, Farach and Iliopoulos [5] introduced the notion of covers and gave a linear-time algorithm for computing the shortest cover of a given string. Moore and Smyth [6] and Li and Smyth [7] gave linear-time algorithms for finding all covers of $x$.

If we can reconstruct a superstring of a string $x$ by some concatenations and superpositions of a string $y$, we call $y$ a *seed* of $x$. In other words, $y$ is a seed of $x$ if $y$ is a cover of any superstring of $x$. For example, if $x=$ababaab, then aba is a seed of $x$. Iliopoulos, Moore, and Park [8] introduced the notion of seeds and gave an $O(n\log n)$-time algorithm to compute all seeds of a given string of length $n$.

Iliopoulos and Smyth [9] considered $k$-covering problem. Given a string $x$ and an integer $k<|x|$, $k$-covering problem is to find a set $W=\{w_1,w_2,\ldots,w_m\}$ such that each $w_i$ ($|w_i|=k$, $1\leq i\leq m$) is a substring of $x$ and $W$ covers $x$ with minimum size $m$. Here, we say a set of strings $W$ covers another string $x$ when we can construct $x$ by concatenating and overlapping some strings in $W$. They presented an $O(n^2(n-k))$-time on-line algorithm for a minimum set of $k$-covers for all prefixes of a given string $x$ ($|x|=n$).

Skiena and Sundaram [10] studied a problem to determine an unknown string over a known alphabet using as few queries as possible. That is, when there is an unknown string $x$, we can ask questions of whether a string $y$ is a substring of $x$ or not, and the goal is to determine the exact contents of $x$ with the minimum number of queries as possible. This problem is highly related to sequencing by hybridization (SBH) [11-13].

In this paper, we consider the following two kinds of string reconstruction problems.

**Problem 1** *String reconstruction problem with a static set*

**Input:** *(i) A string $x$ of length $n$ over a constant-sized alphabet $\Sigma$, (ii) an integer $k$ ($\leq n$), (iii) a set $W$ of strings of lengths at most $k$.*

**Question:** *Can $x$ be reconstructed by concatenations of strings in $W$? If the answer is "YES," report the sequence of strings in $W$ that reconstruct $x$ by concatenations such that the number of concatenations in the sequence is minimal among all such sequences. Otherwise, the answer is "NO."*

**Problem 2** *String reconstruction problem with a dynamic set*

**Input:** *(i) A string $x$ of length $n$ over a constant-sized alphabet $\Sigma$, (ii) an integer $k$ ($\leq n$), (iii) a sequence of a set $W_0, W_1, \ldots, W_j$ of strings of lengths at most $k$ such that $W_{i+1}$ is made by inserting a string into $W_i$ or deleting a string from $W_i$.*

**Question:** *For each $i$ ($0\leq i\leq j$), answer the following question. Can $x$ be reconstructed by*

*concatenations of strings in $W_i$? If the answer is "YES," report the sequence of strings in $W_i$ that reconstruct $x$ by concatenations such that the number of concatenations in the sequence is minimal among all such sequences. Otherwise, the answer is "NO."*

Our contributions are as follows. First, we propose an algorithm that solves Problem 1, i.e., the string reconstruction with a static set, that takes $O(kn+L)$ time, where $L$ is the sum of all lengths of strings in a given set. We use suffix trees and a shortest path algorithm for directed acyclic graphs. Next, we propose an algorithm that solves Problem 2, i.e., a dynamic version of Problem 1. When we naively use the algorithm for Problem 1 to solve Problem 2, it takes $O(kn|\Sigma|^k+L)$ time since there can be $O(|\Sigma|^k)$ insertions in the worst case and it takes $O(kn)$ time for each insertion. In this paper we propose an $O(k^3n+L)$-time algorithm for Problem 2. Finally, we show that exponentiation problems that arise in cryptography can be successfully reduced to these problems and propose a new solution for exponentiation.

This paper is organized as follows. In Section 2, we define some notations and review some related works. In Section 3, we describe our algorithms and analyze them. In Section 4, we introduce the application of our algorithms to cryptography.

## 2. Preliminaries

We first give some definitions and notations. A *string* is a sequence of zero or more characters from an alphabet $\Sigma$. We denote the set of all strings over $\Sigma$ by $\Sigma^*$. The length of a string $x$ is denoted by $|x|$ and $x[i]$ denotes the $i$th character of $x$. When a string $y$ is $x[i]x[i+1]\cdots x[j]$ for $1 \le i \le j \le |x|$, we denote $y$ by $x[i,j]$ and $y$ is called a *substring* of $x$. Conversely, we call $x$ a *superstring* of $y$. For example, baa is a substring of abbaaba and abbaaba is a superstring of baa. A string $y$ is a *prefix* of $x$ if $x=yw$ for $w \in \Sigma^*$. Similarly, $y$ is a *suffix* of $x$ if $x=wy$ for $w \in \Sigma^*$.

The suffix tree due to McCreight [14] is a compacted trie that represents all suffixes of a string $x$. It was designed as a space-efficient alternative to Weiner's position tree [15]. A suffix tree $T_x$ for a string $x$ ($|x|=n$) is a rooted tree with exactly $n$ leaves labeled with 1 to $n$. It takes $O(n)$ time to construct $T_x$ when the size of the alphabet is constant [14,16,17]. Each leaf node labeled with $i$ represents the $i$th suffix of $x$, i.e., $x[i,n]$. Each internal node, except the root node, has at least two children and each edge is labeled with a nonempty substring of $x$. The locus of a string $y$ in $T_x$ is the node associated with $y$. We can construct $y$ by concatenations of each label of the edges on the path from the root to the locus of $y$, if it exists. The extended locus of $y$ is the locus of the longest prefix of $y$ whose locus exists.

Now we can find all the occurrences of $y$ in $x$ using $T_x$ as follows. First, we find the extended locus $E_y$ of $y$ in $T_x$. Next, we report all the labels of the leaves in the subtree rooted at $E_y$ in $O(occ)$ time where $occ$ is the number of occurrences of $y$ in $x$. Thus, using suffix trees, we can find all the occurrences of a pattern $y$ ($|y|=m$) in a text $x$ ($|x|=n$) in $O(m+occ)$ time with $O(n)$ preprocessing time to construct $T_x$. Figure 1 shows an example of the suffix tree when $x=$ aababc. When we want to find all the occurrences of ab in $x$, we first find the extended locus of ab and report all the labels of leaves below it, i.e., 2 and 4. Note that $x[2,3]=x[4,5]=$ ab.

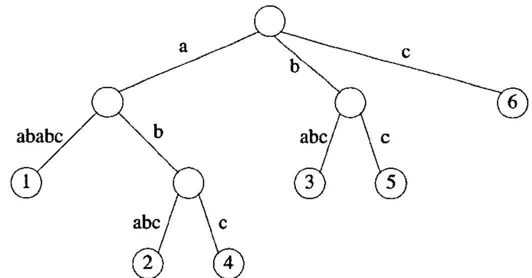


Figure 1 A suffix tree $T_x$ when $x=$ aababc

## 3. String Reconstruction Algorithms

### 3.1 Algorithm for a static set

First, we describe our algorithm for Problem 1. Our algorithm mainly consists of three steps.

1. **Preprocessing**

Assume $W=\{w_1, ..., w_m\}$ and let $L=\sum_{1\le i\le m}|w_i|$. First, we make a suffix tree $T_x$ of $x$ in $O(n)$ time. Next, we construct a directed acyclic graph (DAG for short) $G_x^0$ which corresponds to $x$. $G_x^r$ $(0\le r\le m)$ has following properties: (i) There are $n+1$ nodes in $G_x^r$ that are labeled from $N_0$ to $N_n$. (Each node $N_i$ of $G_x^r$ corresponds to $x[i]$ and $N_0$ is an initial node.) (ii) There is a string $w_i$ in $W$ that occurs in $j$th position of $x$ if and only if there exists an outgoing edge from $N_{j-1}$ to $N_{j+|w_i|-1}$ for $1\le i\le r$. (Note that $G_x^0$ has no edges.) Obviously, it takes $O(n)$ time to construct $T_x$ and $G_x^0$.

For each $w_i$ $(1\le i\le m)$, we perform Step 2. We make $G_x^i$ from $G_x^{i-1}$ and our goal of Step 2 is to make $G_x^m$.

2. **Find all occurrences**

(a) **Find the extended locus $E_{w_i}$ of $w_i$ in $T_x$.**

This can be done by the way of pattern search in a suffix tree and it takes $O(|w_i|)$ time to find each $E_{w_i}$ in $T_x$. If search fails, we go to $(i+1)$st iteration.

(b) **Report all the occurrences of $w_i$ in $x$.**

Report all the labels of the leaves in the subtree rooted at $E_{w_i}$. This step takes $O(occ(w_i))$ time where $occ(w_i)$ is the number of occurrences of $w_i$ in $x$.

(c) **Modify $G_x^{i-1}$ to $G_x^i$.**

We make an outgoing edge from $N_{j-1}$ to $N_{j+|w_i|-1}$ when $w_i$ occurs in $j$th position in $x$, i.e., $w_i=x[j, j+|w_i|-1]$. Since we can add a new edge into $G_x^{i-1}$ in constant time, it takes $O(occ(w_i))$ time for each $w_i$ to make $G_x^i$ from $G_x^{i-1}$. Figure 2 shows an example of $G_x^3$ when $x=$ aababc and $W=$ {a,b,ab}.

3. **Report a shortest sequence of strings if it exists**

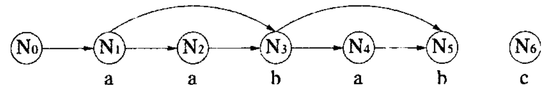


Figure 2 Initial $G_x^3$ when $x=$ aababc and $W=$ {a,b,ab}

Now, the problem which finds a shortest sequence of strings is transformed into the shortest path problem in a DAG if we assume the lengths (or weights) of all the edges are the same, for example, any positive integers such as 1. If we cannot find any shortest path from $N_0$ to $N_n$, we just output "No." (In this case, we cannot reconstruct $x$ with $W$.) If we find a shortest path from $N_0$ to $N_n$, we report all the corresponding strings on the path from $N_0$ to $N_n$, i.e., if an edge from $N_i$ to $N_j$ exists on the shortest path, report $x[i+1, j]$. It takes $O(|N|+|E|)$ time to find a shortest path from a DAG if any, where $|N|$ is the number of nodes and $|E|$ is the number of edges, respectively.

Now we analyze our algorithm for Problem 1. In Step 1, as mentioned above, preprocessing time to construct $T_x$ and $G_x^0$ takes $O(n)$ time. For Step 2-(a), finding each $E_{w_i}$ in $T_x$ takes $O(|w_i|)$ time and therefore performing this step for all $w_i$'s takes $O(L)$ time. (Recall that $L=\sum_{1\le i\le m}|w_i|$.) For each $w_i$, Step 2-(b) takes $O(occ(w_i))$ time. Consider the total number of occurrences of strings in $W$. There can be at most $n$ occurrences when $|w_i|=1$, $n-1$ occurrences when $|w_i|=2$, $n-2$ occurrences when $|w_i|=3$, and so on. Thus in total, there can be at most $kn$ occurrences in $x$ when $|w_i|\le k$ and Step 2-(b) takes $O(kn)$ time for all $w_i$'s in $W$. Also, Step 2-(c) takes $O(kn)$ time in total, by the same reason as Step 2-(b).

Consider Step 3. Obviously, there are $n+1$ nodes in $G_x^m$. Since the number of total occurrences cannot exceed $kn$, the number of newly added edges also cannot exceed $kn$. Thus, the total number of edges is at most $kn$ and therefore Step 3 takes $O(kn)$ time in total. Henceforth, our

algorithm for Problem 1 takes $O(L+kn)$ time in total.

### 3.2 Algorithm for a dynamic set

Now, we consider Problem 2 of the case for a dynamic set. All the procedures performed in this case are very similar to those of Problem 1 but their orders are different. We perform the procedures of Step 2 and Step 3 of the previous algorithm for each $W_i$. Recall that each $W_i$ can be made by either an insertion of a new string $w_i$ into $W_{i-1}$ or a deletion of a string $w_i$ from $W_{i-1}$.

**1. Preprocessing**

First, we make a suffix tree $T_x$ of $x$ in $O(n)$ time. Next, we construct a DAG $G_x^0$ which corresponds to $x$. As the previous algorithm, it takes $O(n)$ time to construct $T_x$ and $G_x^0$.

For each $W_i$, we perform Step 2 and Step 3.

**2. Find all occurrences.**

(a) **Find the extended locus $E_{w_i}$ of $w_i$ in $T_x$.**

It takes $O(|w_i|)$ time to find each $E_{w_i}$ in $T_x$. If search fails, we go to $(i+1)$st iteration.

(b) **Report all the occurrences of $w_i$ in $x$.**

Report all the labels of the leaves in the subtree rooted at $E_{w_i}$. As the previous algorithm, this step takes $O(occ(w_i))$ time.

(c) **Modify $G_x^{i-1}$ to $G_x^i$.**

In the case of insertions, we make an outgoing edge from $N_{j-1}$ to $N_{j+|w_i|-1}$ when $w_i$ occurs in $j$th position in $x$. In the case of deletions, we remove an outgoing edge from $N_{j-1}$ to $N_{j+|w_i|-1}$ when $w_i$ occurs in $j$th position in $x$. Since an insertion of an edge into $G_x^{i-1}$ or a deletion of an edge from $G_x^{i-1}$ takes constant time, it takes $O(occ(w_i))$ time for each $w_i$ to make $G_x^i$ from $G_x^{i-1}$.

### 3. Report a shortest sequence of strings if it exists

If we find a shortest path from $N_0$ to $N_n$, we report all the corresponding strings on the path from $N_0$ to $N_n$. If we cannot find any shortest path from $N_0$ to $N_n$, we just output "No."

For example, assume we are given $x=$ ababbaababbc, $k=3$, and the current set of strings $W_4=$ {a,b,c,ab}. Then, when a new string abb is inserted into $W_4$ to make $W_5$, we find all the occurrences of abb in $x$ using $T_x$. In this case, abb occurs at the position of 3 and 9 of $x$. See Figure 3. Then we add two new edges to $G_x^4$ to make $G_x^5$. Figure 4 shows the modification process. Before the insertion of abb, the shortest path was $N_0, N_2, N_4, N_5, N_6, N_8, N_{10}, N_{11}, N_{12}$, which means we can reconstruct $x$ by the following sequence of

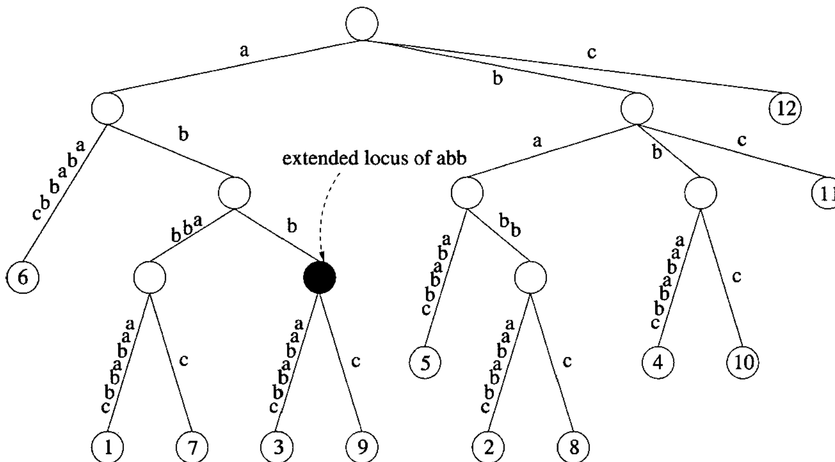


Figure 3 A suffix tree $T_x$ of $x=$ ababbaababbc and the extended locus of abb

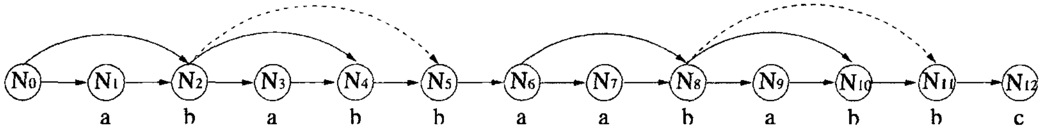


Figure 4 Modification of $G_x^1$ to $G_x^5$. Dotted lines represent newly inserted edges

strings in $W_4$, (ab)(ab)(b)(a)(ab)(ab)(b)(c). (Recall that we assumed all the lengths of edges are the same.) But after the insertion of abb, the shortest path has changed and we can reconstruct $x$ by the following sequence of strings in $W_5$, (ab)(abb)(a)(ab)(abb)(c).

For Problem 2, preprocessing time to construct $T_x$ and $G_x^0$ is the same as Problem 1, and thus, Step 1 takes $O(n)$ time. Consider the case when $W_i$ is made by an insertion of a new string $w_i$ into $W_{i-1}$. Step 2-(a) takes $O(|w_i|)$ time and Step 2-(b) takes $O(occ(w_i))$ time as Problem 1. Step 2-(c) also takes $O(occ(w_i))$ time for each $w_i$. (Note that if $w_i$ does not occur in $x$, Step 2-(b) and Step 2-(c) will not be performed.) Now consider the case when $W_i$ is made by a deletion of $w_i$ from $W_{i-1}$. As the case of insertions, Step 2-(a) takes $O(|w_i|)$ time. Also, Step 2-(b) and Step 2-(c) take $O(occ(w_i))$ time for each $w_i$ if $w_i$ occurs in $x$. Thus, for both cases of insertions and deletions, Step 2-(a) takes $O(L)$ time in total. Step 2-(b) and Step 2-(c) take $O(kn)$ time in total because there can be at most $O(kn)$ occurrences. Therefore, Step 2 can be done in $O(L+kn)$ time in total.

Note that we perform Step 3 only when $w_i$ occurs in $x$, i.e., $w_i$ is a substring of $x$. Since there can be $O(k^2)$ substrings, Step 3 takes $O(k^3n)$ time in total. Therefore, our algorithm for Problem 2 takes $O(L+k^3n)$ time in total.

## 4. Application to Cryptography

In this section, we show how our algorithm can be applied to the exponentiation problem, which is one of the most important problems in cryptography. Exponentiation over a group is to compute $X^e$ for a group element $X$ and a positive integer $e$. This is a fundamental operation for many modern cryptosystems, e.g. the RSA [18] and ElGamal public key schemes [19] and the DSA digital signature scheme [20]. To guarantee the security of these schemes, very large exponents $e$ should be used. Therefore, it is crucial to develop fast exponentiation methods that are suitable for large exponents, and there has been an extensive research for this problem [21]. There has also been some research to relate fast exponentiation to string problems, including Yacobi's work [22] where the Lempel-Ziv compression algorithm is used to speed up exponentiation. Since an exponentiation is composed of repeated multiplications, the study on the speed of exponentiation can be classified into two categories: to speed up multiplication itself, and to reduce the number of required multiplications. We are interested in the latter in this paper.

Recently, Park, Lee and Kim [23] showed that the problem of finding the minimum number of multiplications for an instance of exponentiation is equivalent to finding a shortest path in its corresponding graph, and they suggested two heuristic methods to reduce the path length dynamically. But they did not present any specific procedure to construct the graph, nor any complexity analyses. In this section, we give a complete procedure to solve exponentiation problems using our string reconstruction algorithm, which involves the shortest path problem. Hence our algorithm can be viewed as a generalization of [23]. We also give a precise analysis of time complexity.

First, we must define our problem more precisely. For the exponentiation $X^e$, we can assume that some powers of $X$ are computed at first, and then these values are used to construct a computation path for $X^e$. We call the table of these precomputed powers of $X$, a precomputation table. Note that this is a reasonable assumption since many efficient exponentiation algorithms such as the $M$ ary method

[24] and the sliding window method [25] use this technique. Hence we focus on the exponentiation problem that uses a precomputation table $\mathcal{T}$. We also assume that $\mathcal{T}$ contains $X^1=X$ and it does not contain $X^0=1$.

Now we show how we can solve this problem using a string reconstruction algorithm. As input to the exponentiation problem, we are given a base $X$, an exponent $e$ and a set $U$ of exponents that are supposed to be included in precomputation table $\mathcal{T}$ such that $U$ contains $u$ if and only if $\mathcal{T}$ contains $X^u$. Note that $1 \in U$ and $0 \notin U$ according to our assumption. Then we can reduce the exponentiation problem to a string reconstruction problem as follows:

1. Represent $e$ as a string $e_1e_2\ldots e_l$, where $l = \lceil \log_2(e+1) \rceil$, $e_i \in \{0,1\}$ and $e=\sum_{i=1}^{l} e_i 2^{l-i}$. Hence $e_1=1$.
2. Set $\Sigma=\{0,1\}$ and $x=e_2\ldots e_l$.
3. Set $k=\max_{u\in U}\{$ number of bits in $u\}$ and set $W \leftarrow U$. (Now $W$ is interpreted as a set of binary strings.)
4. Solve the string reconstruction problem using the algorithm given in the previous section. (When we solve the shortest path problem, we set the length of all edges to 1.) If the answer to the string reconstruction problem is "YES," then we obtain a sequence $w_1,\ldots,w_r$. If the answer is "NO," then this means that there is no path from the source node to the terminal node. However, this case can also be easily dealt with by inserting an edge $(N_{i-1},N_i)$ for each $N_i$ corresponding to '0'. Then assigning length 0 to each of these new edges and solving a shortest path problem always succeeds in producing an answer $w_1,\ldots,w_r$. In this case, however, we should modify our assumption on $\mathcal{T}$ so that $\mathcal{T}$ can contain $X^0=1$ since the case that $w_i=0$ is possible for some $i$'s.

Then an exponentiation will be done using the sequence $w_1,\ldots,w_r$ as follows:

1. Construct $\mathcal{T}=\{X^{w_1},\cdots,X^{w_r}\}$ using some multiplications.
2. $Y \leftarrow X$.
3. For $i=1$ to $r$ do
   $Y \leftarrow Y^{2^{|w_i|}} \times X^{w_i}$, where $X^{w_i}$ is fetched from $\mathcal{T}$.
4. return $Y$.

Therefore, the computational cost for exponentiation is

$$\text{(cost to construct } \mathcal{T}) + (\sum_{i=1}^{r}|w_i| \text{ squarings}) + ((r-s) \text{ multiplications}), \quad (1)$$

where $s$ is the number of $w_i$'s such that $w_i=$'0'. Note that $\sum_{i=1}^{r}|w_i|=|x|=l-1$ for any possible sequence $w_1,\ldots,w_r$. Thus we can see that if we fix the precomputation table $\mathcal{T}$, the first and second factors in (1) are always the same for any possible sequences. Then reducing $r$ is equivalent to finding a more efficient computation path for $X^e$.

We mention that the above algorithm for finding an efficient multiplication sequence can be improved further if we slightly change the string reconstruction algorithm. First, we observe that in Step 2 of the above exponentiation algorithm, if we can initialize $Y$ as one of the elements in $\mathcal{T}$, then we can reduce the number of required squarings so that it may be less than $l-1$. Our algorithm to find a short computation sequence can be modified easily so that this case may be covered. All we have to do is to feed $x=e_1\ldots e_l$ instead of $x=e_2\ldots e_l$ and assign weight 0 to all the outgoing edges from $N_0$ when we solve the shortest path problem.

An attractive feature of our approach is that we can construct $\mathcal{T}$ adaptively according to the characteristics of a specific $e$, which differs from the $M$-ary method and the sliding window method that use only the same tables for given parameters. To be precise, we proceed as follows:

1. Start with some initial $U$ and find a sequence $w_1,\ldots,w_r$ using the string reconstruction algorithm.
2. Evaluate (1).
3. Repeat the following for reasonable times: insert an element to $U$ and if this new $U$ gives a smaller value of (1), then set this $U$ as a candidate.

4. Compute the exponentiation using this final $U$.

Note that the adaption procedure (Steps 1-3) can be implemented using the string reconstruction algorithm with a dynamic set, and it performs much faster than exponentiation itself (Step 4) in a practical situation:

- *String reconstruction cost.* In a practical setting of an exponentiation, $k$ is chosen so that $k \ll n = l-1$. A typical choice is $k = O(\log_2 n)$. Note that since $L < k \cdot 2^k$, the cost for dynamic string reconstruction is $O(k \cdot 2^k + k^3 \cdot n) = O(n(\log_2 n)^3)$.
- *Exponentiation cost.* In (1), the costs for one squaring and one multiplication are virtually the same. Then (1) becomes $\Omega(n)$ multiplications. Moreover, practical multiplication algorithms use $\Theta(n^2)$ bit operations for $n$-bit operands. Hence we require $\Omega(n^3)$ bit operations for one exponentiation.

## 5. Conclusion

In this paper, we define two string reconstruction problems and show how exponentiation problems that arise in cryptography can be successfully reduced to these problems. Most of the algorithms for repetitive strings can be applied to well-known areas such as data compressions and bioinformatics. However, string algorithms are rarely used in the field of cryptography, which is one of the important research areas.

## References

[1] M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Information Processing Letters* 12, 5 (1981), 244-250.

[2] A. Apostolico and F.P. Preparata, Optimal off-line detection of repetitions in a string, *Theoretical Computer Science* 22 (1983), 297-315.

[3] M.G. Main and R.J. Lorentz, An algorithm for finding all repetitions in a string, *Journal of Algorithms* 5 (1984), 422-432.

[4] D.E. Knuth, J.H. Morris, and V.R. Pratt, Fast pattern matching in strings, *SIAM Journal on Computing* 6, 1 (1977), 323-350.

[5] A. Apostolico, M. Farach, and C.S. Iliopoulos, Optimal superprimitivity testing for strings, *Information Processing Letters* 39, 1 (1991), 17-20.

[6] D. Moore and W.F. Smyth, A correction to "An optimal algorithm to compute all the cover of a string," *Information Processing Letters* 54, 2 (1995), 101-103.

[7] Y. Li and W.F. Smyth, An optimal on-line algorithm to compute all the covers of a string, preprint.

[8] C.S. Iliopoulos, D.W.G. Moore and K. Park, Covering a string, *Algorithmica* 16 (1996), 288-297.

[9] C.S. Iliopoulos and W.F. Smyth, On-line algorithms for $k$-covering, in: *Proc. 9th Australasian Workshop on Combinatorial Algorithms* (1998), 97-106.

[10] S.S. Skiena and G. Sundaram, Reconstructing Strings from Substrings, *Journal of Computational Biology* 2 (1995), 333-353.

[11] W. Bains and G. Smith, A novel method for nucleic acid sequence determination, *Journal of Theoretical Biology* 135, 3 (1988), 303-307.

[12] S. Fodor, J. Read, M. Pirrung, L. Stryer, A. Lu, and D. Solas, Light-directed, spatially addressable parallel chemical synthesis, *Science* 251 (1991), 767-773.

[13] P.A. Pevzner and R.J. Lipshutz, Towards DNA sequencing by hybridization, in: *19th Symp. on Mathem. Found. of Comp. Sci.*, LNCS 841 (1994), 143-258.

[14] E.M. McCreight, A space-economical suffix tree construction algorithm, *Journal of ACM* 23 (1976), 262-272.

[15] P. Weiner, Linear pattern matching algorithms, *Proc. 14th IEEE Symp. Switching and Automata Theory* (1973), 1-11.

[16] E. Ukkonen, On-Line Construction of Suffix Trees, *Algorithmica* 14, 3 (1995), 249-260.

[17] D. Gusfield, Algorithms on Strings, Trees, and Sequences, *Cambridge Univ. Press* 1997.

[18] R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21 ,2 (1978), 120-126.

[19] T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Transactions on Information Theory* 31, 4 (1985), 469-472.

[20] National Institute of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186 (1994).

[21] D.M. Gordon, A Survey of Fast Exponentiation Methods, *Journal of Algorithms* 27, 1 (1998), 129-146.

[22] Y. Yacobi, Fast exponentiation using data compresseio, *SIAM Journal on Computing* 28, 2 (1998), 700-703.

[23] C.S. Park, M.-K. Lee and D.K. Kim, New computation paradigm for modular exponentiation using a graph model, in: *Stochastic Algorithms: Foundations and Applications (SAGA 2005)*, LNCS 3777 (2005), 170-179.

[24] D. E. Knuth, The art of computer programming: Seminumerical algorithms, Volumn 2, 2nd edition, *Addison-Wesley, Reading, MA* (1981) 461-485.
[25] C. K. Koç, Analysis of sliding window techniques for exponentiation, *Computers and Mathematics with Application* 30, 10 (1995) 17-24.

심 정 섭
1995년 서울대학교 컴퓨터공학과 학사 1997년 서울대학교 컴퓨터공학과 석사 2002년 서울대학교 전기컴퓨터공학부 박사. 2002년~2004년 한국전자통신연구원(선임연구원). 2004년 9월~현재 인하대학교 컴퓨터정보공학부(조교수). 관심분야는 알고리즘, 최적화이론, 바이오인포매틱스

이 문 규
정보과학회논문지 : 시스템 및 이론 제 35 권 제 7 호 참조

김 동 규
1992년 2월 서울대학교 컴퓨터공학과(공학사). 1994년 2월 서울대학교 컴퓨터공학과(공학석사). 1999년 2월 서울대학교 컴퓨터공학과(공학박사). 1999년 9월~2006년 2월 부산대학교 컴퓨터공학과(조교수). 2006년 3월~현재 한양대학교 전자통신컴퓨터공학부(부교수). 관심분야는 Cryptology, 암호화 모듈 칩 설계, String processing