가변길이 분할 기법을 적용한 모듈러 지수연산법

Modular Exponentiation Using a Variable-Length Partition Method

이상운* Sang-Un Lee*

요 약 암호학의 암호 생성과 해독의 곱셈 횟수는 대부분 $d^e \pmod{m}$ 모듈러 지수연산의 효율적 구현여부로 결정된다. 표준 모듈러 지수연산법으로는 $1-ary$법이 이전법이 있으며, $n-ary(2 \leq n \leq 6)$법이 많이 적용되고 있다. $n-ary(1 \leq n \leq 6)$법은 $b = b_kb_{k-1} \cdots b_{h(2)}$에 대해 R-L 방향으로 $n$비트로 고정된 분할을 하고, $n$회 제곱과 비트값 곱셈을 수행하는 제곱-곱셈법이다. 본 논문에서는 $b_{k-1} \cdots b_{h(2)}$에 대해 L-R 방향으로 가변길이로 분할하는 방법을 적용한다. 또한, 가변길이의 제곱과 곱셈 또는 나눗셈을 적용한다. 제안된 가변길이 분할법은 고정길이 분할법인 $n-ary$법에 비해 곱셈 수행횟수를 감소시킬 수 있었다.

Abstract The times of multiplication for encryption and decryption of cryptosystem is primarily determined by implementation efficiency of the modular exponentiation of $d^e \pmod{m}$. The most frequently used among standard modular exponentiation methods is a standard binary method, of which $n-ary(2 \leq n \leq 6)$ is the most popular. The $n-ary(2 \leq n \leq 6)$ is a square-and-multiply method which partitions $b = b_kb_{k-1} \cdots b_{h(2)}$ into $n$ fixed bits from right to left and squares $n$ times and multiplies bit values. This paper proposes a variable-length partition algorithm that partitions $b_{k-1} \cdots b_{h(2)}$ from left to right. The proposed algorithm has proved to reduce the multiplication frequency of the fixed-length partition $n-ary$ method.

Key Words : Modular exponentiation, Binary method, $n-ary$ method, Fixed-length, Variable-length

Ⅰ. 서 론

$a^b \pmod{m}$은 $a^b$의 값을 $m$으로 나눈 나머지를 구하는 모듈러 지수연산 (modular exponentiation)이라 하며 암호학 분야에 널리 사용되고 있다. [1,2] 대표적인 모듈러 지수 연산법에는 $a^b \pmod{m}$의 $b = b_kb_{k-1} \cdots b_{h(2)}$를 R-L (right-to-left)로, $n$비트 값의 값을 초기치 $d^0$로 설정하고, $n-1$부터 $n$까지 $n$회 제곱 후 비트 값을 지수항 덧셈 (곱셈)을 수행하는 $n$항 법 ($n-ary$ method)이 있다. $n-ary$법은 각 비트 값이 가질 수 있는 가능한 경우수인 $2^n$개 중에서 0,1을 제외한 $2^n-2$개를 사전 처리 (preprocessing)하여 저장한 후 비트 값 곱셈에 사용한다. 이외에도 몽고메리 감소법 (Montgomery reduction)과 추가사슬 (addition chain)법 등이 있다. [3-8] RN-ary법은 $b = b_kb_{k-1} \cdots b_{h(2)}$에 대해 항상 고정된 길이 $n$비트값 분할한다. 이를 동신 분야에서 데이터
압축시 적용되는 기법인 고정길이 부호 기법 (fixed-length coding method)으로 설명될 수 있다. 데이터 압축의 효율성을 증대시키는 방법으로 가변길이 부호 기법 (variable-length coding method)을 일반적으로 적용하고 있다. 본 논문은 4-2의 곱셈 횟수를 감소시키는 방법으로 가변길이 부호 기법을 적용한다. 알고리즘의 차이점은 첫 번째로, \( b = b_k b_{k-1} \cdots b_0 \) 에 대해 \( n-\text{ary} \) 법은 R-L로 \( n \) 비트씩 일정하게 분할하는 기법으로 \( n \)-부록부호 (block code) 라 할 수 있다. 반면에, 제안된 방법은 \( b_k \) 를 제외한 \( b_{k-1} b_{k-2} \cdots b_0 \) 에 대해 L-R (left-to-right) 가변길이로 비트를 분할하는 기법을 적용한다. 따라서 제안된 방법은 고정 길이 기법과 상반되는 개념이다. \( n-\text{ary} \) 법을 "고정길이 분할법"이라고, 제안된 방법은 "가변길이 분할법"이라 한다. 두 번째로, 고정길이 분할법은 사전처리를 수행하는데 반해 가변길이 분할법은 사전처리를 하지 않고, 비트 값 계산은 알고리즘 수행과정에서 얻은 값을 활용한다. 세 번째로, 고정길이 분할법은 비트 값 계산은 항상 곱셈으로 얻는다. 반면에 가변길이 분할법에서는 곱셈 또는 나눗셈을 적용한다.

2장에서는 고정길이 분할법을 고찰한다. 3장에서는 가변길이 분할법을 제안하고, 고정길이 분할법과 가변길이 분할법의 수행 횟수를 비교하여 본다.

II. 고정길이 분할법

일반적으로 알려진 \( n-\text{ary} \) 법은 그림 1과 같이 수행 된다.\(^6\) 그림 1의 \( a \)-\(\text{ary} \) 법과 \( n-\text{ary} \) 법을 통합하여 일반화시켜 표현하면 그림 2와 같이 수행되며, 그림 1 방법에 비해 곱셈을 1회 감소시킬 수 있다.

\( b = 1933 = 1111001101_2 \)인 경우의 \( a \)-\(\text{ary} \) 법, \( 2-\text{ary} \) 법과 \( 3-\text{ary} \) 법의 계산 방법은 그림 3에 제시하였다. 이진법은 사전처리 횠수(0)+제곱횟수(10)+비트 값 곱셈횟수(6)=16회를 수행하였다. \( 2-\text{ary} \) 법은 2\( \times 5\times 2\times 4\)로 16회, \( 3-\text{ary} \) 법은 6\( \times 3\times 3\times 3\times 3\)으로 18회를 수행한다. \( b = 127 = 111111111_2 \)인 경우, 이진법은 0\( \times 2\times 2\times 2\times 2 \)\( \times 2\times 2\times 2 \)\( \times 2\)로 12회 계산을 수행한다. \( 2-\text{ary} \) 법은 01111111 = 1333\( \times a_4 \)에 대해 2\( \times 3\times 3\times 3\times 3 \)으로 11회 수행한다. \( 3-\text{ary} \) 법은 00111111 = 177\( \times a_6 \)에 대해 6\( \times 4\times 4\times 14 \)회를 수행한다. 결국, \( b = 127 \)인 경우 \( 2-\text{ary} \)법이 최선의 방법임을 알 수 있다.

\[
a^b \pmod{n}
\]

\[\text{Modular-Exponentiation } (a, b, n)\]
\[c = 1, \quad b = b_k b_{k-1} \cdots b_0 \]
\[\text{for } i = k \text{ down to 0 do}\]
\[c = (c \times c) \pmod{n} \quad /\ast \text{ square}\]
\[\text{if } b_i = 1 \text{ then } c = (c \times a) \pmod{n}\]
\[\text{return } c\]

\[
(a) \text{ Binary method}
\]

\[
a^b \pmod{n} \quad b = b_k b_{k-1} \cdots b_0 \]
\[\text{Preprocessing (사전 처리)}\]
\[a_0 = 1\]
\[\text{for } i = 1 \text{ to } (2^k - 1) \text{ do}\]
\[a_i = (a_{i-1} \times a) \pmod{n}\]
\[c = 1\]
\[\text{Modular-Exponentiation } (a, b, n)\]
\[\text{for } i = n \text{ down to 0 do}\]
\[\text{for } j = 0 \text{ to } k - 1 \text{ do}\]
\[c = (c \times c) \pmod{n} \quad /\ast \text{ power}\]
\[\text{if } b_i = 0 \text{ then } c = (c \times a_k) \pmod{n}\]
\[\text{return } c\]

\[
(b) \text{ n-ary } (n > 1) \text{ method}
\]

그림 1. 전형적인 \( n-\text{ary} \) 지수연산법

Fig. 1. Typical \( n-\text{ary} \) modular exponentiation

\[
a^b \pmod{m} \quad b = n_k n_{k-1} \cdots n_0 n_{0}(n)\]
\[\text{Preprocessing (사전 처리)}\]
\[a_i = a\]
\[\text{if } n \geq 3 \text{ then } /\ast \text{ } 1-\text{ary} \text{와 } 2-\text{ary} \text{는 수행 없음.}\]
\[\text{for } i = 2 \text{ to } (2^n - 1) \text{ do}\]
\[a_i = (a_{i-1} \times a) \pmod{m}\]
\[\text{end}\]
\[\text{end}\]
\[\text{Modular-Exponentiation } (a, b, m)\]
\[c = a_k /\ast n_k\text{값의 } a_k \text{ 값. 만약, } n_k = 10\text{면 } a.\]
\[\text{for } i = k - 1 \text{ down to 0 do}\]
\[\text{for } j = 1 \text{ to } n \text{ do}\]
\[c = (c \times c) \pmod{m}\]
\[\text{end}\]
\[c = (c \times a_k) \pmod{m} /\ast n_k\text{값의 } a_k \text{ 값.}\]
\[\text{end}\]
\[\text{return } c\]

그림 2. 일반화된 \( n-\text{ary} \) 지수연산법

Fig. 2. Generalized \( n-\text{ary} \) modular exponentiation
이진법 곱셈

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
\text{구분} & \text{이진법} & \text{2-ary} & \text{3-ary} & \text{레기} & \text{결과} & \text{레기} & \text{결과} \\
\hline
\text{사전 처리} & - & - & - & 10 & - & 11 & - \\
& & & & a \times a & & a_2 \times a & \\
& & & & 100 & a_4 \times a & & 101 \times a & \\
& & & & 101 & a_4 \times a & & 111 & a_6 \times a & \\
\hline
b_0, b_1, b_2 & 1 & - & - & - & - & - & - \\
& (a_1)^2 & (a_2)^2 & (a_3)^2 & a_2^{1+3} & a_3^{2+3} & & & \\
& (a_1)^2 & (a_2)^2 & (a_3)^2 & a_4^{1+3} & a_2^{1+3} & & & \\
& (a_1)^2 & (a_2)^2 & (a_3)^2 & a_4^{1+3} & a_2^{1+3} & & & \\
& (a_1)^2 & (a_2)^2 & (a_3)^2 & a_4^{1+3} & a_2^{1+3} & & & \\
& (a_1)^2 & (a_2)^2 & (a_3)^2 & a_4^{1+3} & a_2^{1+3} & & & \\
& (a_1)^2 & (a_2)^2 & (a_3)^2 & a_4^{1+3} & a_2^{1+3} & & & \\
& & & & & & & & \\
\text{그림 3. } a^{1933} \text{ 계산 } n-\text{ary법} \\
\text{Fig. 3. } a^{1933} \text{ Computation using } n-\text{ary method} \\
\end{array}
\]

\[
b = b_k b_{k-1} \cdots b_0 \text{에 대해 } n-\text{ary법은 } 2^n - 2\text{회 사전처리, } n\left(\frac{k+1}{n} - 1\right)\text{회 계산과 } \left(\frac{k+1}{n} - 1\right) \text{회 곱셈을 수행합니다.}
\]

\[
(2^n - 1)\text{회 곱셈은 수행할 수 있다. 이 공식을 적용하면 이론적으로 최적의 } n-\text{ary법을 결정할 수 있다. RSA는 일반적으로 이진수 512, 1024 또는 2,048 비트를 적용한다. 따라서 } 2 \leq k \leq 2048\text{에 대해 계산한 결과 } k \text{ 범위별 최적의 } n-\text{ary법은 그림 4에 제시하였다. 그러나 실제로 } b_i = 1 \text{의 개수에 따라 최적의 } n-\text{ary법이 달라질 수 있다.}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
\text{범위} & \text{1-ary} & \text{2-ary} & \text{3-ary} & \text{4-ary} & \text{5-ary} & \text{6-ary} & \text{7-ary} & \text{8-ary} \\
\hline
\text{범위별 최적의 } n-\text{ary법의 이진자리수 } k \text{ 범위} \\
\hline
\text{그림 4. } k \text{ range of binary digits number for optimal } n-\text{ary method} \\
\end{array}
\]

\section{III. 가변길이 분할법}

모듈러 지수 연산법에서 가변길이로 분할하는 방법은 연구되지 않고 있다. 본 장에서 제안하는 가변길이

\[
\text{분할법은 다음의 2가지 기법을 추가적으로 접목시킨다.}
\]

\[
(1) n-\text{ary법은 } 2^n - 2\text{회의 곱셈 사전처리를 수행하는데 반해 가변길이 분할법은 사전 처리를 수행하지 않고 알고리즘 수행 과정에서 얻는 지수 값을 활용한다.}
\]

\[
b = 48 = 110000_2 \text{인 경우, 사전처리를 하지 않으면 } 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow (48 = 32 + 16) \rightarrow 48 \text{의 7회 곱셈을 수행한다. 만약, } 2-\text{ary법을 적용한다면, 사전에 2\rightarrow 3\text{을 구한 후 } 1.2 \rightarrow 3 \rightarrow 3 \rightarrow 3 \rightarrow 3 \text{을 수행하여 8회의 곱셈을 수행하는데 반해 사전 처리된 값을 곱셈에 전혀 사용하지 않는다.}
\]

\[
(2) n-\text{ary법은 비트 값에 대해 곱셈 (지수 항 덧셈)만을 수행한다. 반면에 가변길이 분할법은 곱셈 또는 나눗셈을 적용한다.}
\]

\[
b = 127 = 1111111_2 \text{에 대한 곱셈 수행 횟수는 이진법은 } 12\text{회, } 2-\text{ary법은 } 11\text{회, } 3-\text{ary법은 } 14\text{회임을 } 2\text{장에서 보았다. 만약, } a^{127} = a^{128-1} \text{으로 모듈러 지수 나눗셈을 계산할 수 있다면 } a^{127} = a^{126} \times a^{128-1} \rightarrow a^{128-1} = a^{127} \text{로 } 7\text{회의 곱셈과 } 1\text{회의 나눗셈으로 } 8\text{회로 단축시킬 수 있다. 모듈러 지수 나눗셈 } a^{b-c} \text{을 적용하여 계산하는 방법은 제안되지 않고 있다. 모듈러 지수 나눗셈은 } n \text{의 임의의 배수를 더한 값 을 나누어 정수가 되어야 되기 때문에 본 장에서는 모듈러 나눗셈 } a^{b} \text{을 } a^{b} \text{를 } a^{b} \text{의 } a^{c} \text{으로 계산할 수 있으며, 이 경우 } j \text{ 값을 찾아야 한다.}
\]

\[
\text{while } j = 1 \text{ to } e = \text{정수} \\
\quad e = a^{b \times j} \mod (mod(m)) \text{, } e = \text{정수 or} \\
\quad e = a^{b \times j} \mod (mod(m)) \text{, } e = \text{정수} \\
\text{end}
\]

\section{IV. 모듈러 나눗셈 방법}

\[
\text{Fig. 5. Modular divide method}
\]
가변길이 분할법의 핵심은 b의 이진수에 대해 비트 수를 가변적으로 분할하는 기준이다. 

### IV. 적용 결과 및 분석

32 ≤ b ≤ 64범위에 대해 이진법과 가변길이 분할법의 횟수를 비교한 결과는 표 3에 제시되어 있다.
다음으로, (2⁺²⁻⁶⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻¹⁻²⁻��

\[ a^k \mod n, b = (b_k b_{k-1} \cdots b_1) \]

**Modular-Exponentiation** \((a, b, m)\)

\[ c = a \]

\[ b = b_{k-1} b_{k-2} \cdots b_0 \text{ 을 } l \text{개로 분할, } /* \text{ 분할 기준 참조} \]

for \( i = 1 \) to \( l \)

for \( j = 1 \) to \( p_i \) do /\* \( p_i \) 각 분할의 비트 수

\[ c = (c \times c_i) \mod (m) \]

end if

비트 값 = 이진 \( a^{k+b} \) then

\[ c_i = (c \times c_i) \mod (m) \]

\[ c = (c \times c_i) \mod (m) \]

else if 비트 값 = \( 2^k - 1 \) then

while \( j \) to \( c_i \) 정수 do /\* \( c_i \) \( a^k \mod (m) \)

\[ c_i = \frac{a^k \times (m \times j)}{a} \mod (m) \]

\[ j = j + 1 \]

end if

\[ c = (c \times c_i) \mod (m) \]

end if

return \( c \)

그림 6. 가변길이 분할법

Fig. 6. Variable-length partition method

**표 3. \( 32 \leq b \leq 64 \) 길이의 수행횟수**

Table 3. Trial number of multiplication for

\( 32 \leq b \leq 64 \)
추가적으로 64 ≤ b ≤ 128에 대해 몇 개의 숫자에 대해 이진법, 2−ary법과 가변길이 분할법을 적용한 사례는 표 4에 제시되어 있다.

표 4에는 이진 자리수 모 두 7개 이상으로 이론상으로는 2−ary법이 이진법에 비해 효율적인 것으로 알려져 있지만 127를 제외하면 실제로는 이진법이 더 효과적임을 알 수 있다. 또한, 가변길이 분할법은 표 3에서의 결과와 동일하게 모든 b값에 대해 n−ary법(1 ≤ n)법과 동일하거나 보다 좋은 결과를 나타내고 있다.

<table>
<thead>
<tr>
<th>b</th>
<th>2진수</th>
<th>수행 횟수 (이진법/ 2−ary/가변길이 분할법)</th>
<th>64</th>
<th>1000 0000 00</th>
<th>111 0000 00</th>
<th>1111 0110</th>
<th>127 0000 00</th>
<th>1000 0000 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>01 01 01 01 01 01</td>
<td>2−31, 1, 1−1/2−1 +16 32 64 (64회)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>01 01 01 01 01</td>
<td>2, 1−2−1+1/2−1 +16 32 64 (64회)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>01 01 01 01 01</td>
<td>2, 1−2−1+1/2−1 +16 32 64 (64회)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V. 결론

본 논문은 가변길이 분할법을 적용한 모듈러 지수연산법을 제안하였다. n−ary법은 2 = \( b_0 + b_{n-1} + \ldots + b_{2} \)에 대해 \( R = L \)로 n 비트의 동일한 비트로 분할하는 고정길이 분할법이라 할 수 있다. 제안된 가변길이 분할법은 \( b_{n-1} = \ldots = b_{2} \)에 대해 \( L = R \)로 가변길이 분할 기준을 적용하여 분할하는 방법을 적용하였다. 또한, n−ary법의 \( 2^k = 2 \)개의 가능한 비트 집에 대해 갑족 사전처리를 수행하지 않으며, 비트 갯에 대해 갑종 또는 나눗셈을 적용하는 특징이 있다.
가변길이 분할법은 고정길이의 $n-ary$법에 비해 수행 횟수를 보다 감소시키는 효과를 얻었다.

References


