# 모듈라 멱승 연산의 빠른 수행을 위한 새로운 모듈라 곱셈 알고리즘†

홍 성민°, 오 상엽, 윤 현수
한국과학기술원 전산학과

# A New Modular Multiplication Algorithm for Fast Modular Exponentiation

Hong Seong-Min°, Oh Sang-Yeop, Yoon Hyun-Soo
Dept. of Computer Science, KAIST

**요약**

모듈라 멱승(modular exponentiation) 연산은 암호학에서 기본적이고 중요한 연산이다. 그러나, 이는 다정도 정수(multiple precision integer)들을 다루기때문에 그 연산시간이 무척 많이 걸리므로 이를 단축시킬 필요가 있다. 모듈라 멱승 연산은 모듈라 곱셈(modular multiplication)의 반복으로서, 전체 연산시간을 단축시키기 위해서는 모듈라 곱셈의 수행시간을 단축시키거나, 모듈라 곱셈의 반복횟수를 줄이는 것이 필요하다. 본 논문에서는 모듈라 곱셈을 빠르게 수행하기 위한 알고리즘 두 개를 제안한다. 하나는 서로다른 두 수의 모듈라 곱셈 알고리즘이고, 다른 하나는 모듈라 제곱을 빠르게 수행하는 알고리즘이다. 이 둘은 기존의 모듈라 곱셈 알고리즘들에 비해 각각 절반과, 1/3가량의 단정도 곱셈(single-precision multiplication)만을 필요로 한다. 실제로 PC상에서 구현한 결과 각각 100%와 30%의 속도향상을 보인다.

## 1 서론

1976년 Diffie와 Hellman이 공개키 개념을 제안한 이래로 많은 공개키 암호시스템들이 개발되었다[1, 2]. 그 중 많은 암호시스템들이 모듈라 멱승 연산을 필요로 한다[3, 1, 4]. 따라서, 모듈라 멱승 연산은 암호학에서 기본적이고 중요한 연산이 되었다. 그런데, 이는 매우 큰 수들을 다루기때문에 그 연산시간이 무척 오래 걸리므로 이를 단축시키 위한 연구들이 많이 수행되어 왔다[5, 6, 7, 8, 9, 10, 11].

모듈라 멱승 연산은 모듈라 곱셈의 반복으로 이루어진다. 따라서, 전체 연산시간을 단축시키기 위해서는 모듈라 곱셈의 수행시간을 단축시키거나, 모듈라 곱셈의 반복횟수를 줄이는 것이 필요하다. 예를 들면, $7^9 \bmod 8$을 수행하고자 할 때, 이를 빠르게 하는 방법은 다음 두 가지가 있다.

1. $(\cdots((((7 \times 7) \bmod 8) \times 7) \bmod 8)\cdots) \times 7) \bmod 8$과 같이 8번의 모듈라 곱셈을 수행하는 대신에, $((((7^2 \bmod 8)^2 \bmod 8)^2 \bmod 8) \times 7) \bmod 8$처럼 4번의 모듈라 곱셈을 수행한다.

2. $(7 \times 7) \bmod 8$과 같은 개개의 모듈라 곱셈 연산을 빠르게 수행한다.

위의 첫 번째 경우가 덧셈사슬(addition-chain)을 이용해서 필요한 모듈라 곱셈의 횟수를 줄이는 방법으로서 이에 대해 이루어진 연구들이 [5, 6, 7, 8, 9, 10, 11] 등에 나타나 있다. 두

†본 연구는 국민은행과 (주)한글과컴퓨터의 지원에 의해 수행중임

번째 경우가 모듈라 곱셈을 빠르게 수행하는 방법으로, 이에 대해 이루어진 연구들도 많이 수행되어 왔다 [6, 12, 13, 14, 15, 16, 17, 18].

두번째 방법은 다시 두 가지로 나뉘어진다. 하나는 다정도 곱셈(multiple-precision multiplication)과 모듈라 감소(modular reduction)를 나누어 생각하는 접근방법이다[6, 17, 12, 13]. 다른하나는 모듈라 곱셈을 하나의 연산으로 생각하고 계산하는 방식이다[16, 15, 18]. 본 논문에서 제안하는 알고리즘들도 이러한 방식의 알고리즘들이다. 기본 아이디어는 자주 계산되는 값들을 미리 저장하여 그 정보들을 이용함으로써 전체 연산시간을 단축시키는 것이다.

모듈라 멱승 연산에 필요한 모듈라 곱셈의 반복횟수를 줄이는 방법 중에서, 분석가능한 가장 좋은 방법은 작은윈도우(small-window) 기법이다. 이러한 작은윈도우 기법으로 모듈라 멱승 연산을 수행할 경우에 필요한 모듈라 곱셈은 두 가지 종류로 나누어진다. 하나는 서로다른 두 수의 모듈라 곱셈이고, 다른 하나는 모듈라 제곱(modular squaring)이다. 본 논문에서는 두 개의 알고리즘을 제안한다. 하나는 서로다른 두 수의 모듈라 곱셈을 빠르게 수행하는 알고리즘이다. 이는 [15]에서 Kawamura등이 제안한 방법을 작은윈도우 기법에 적용가능하도록 확장한 것이다. 다른하나는 모듈라 제곱시에 모듈라 감소를 빠르게 수행할 수 있도록 모듈라 곱셈의 연산순서(modular multiplication sequence)를 수정한 알고리즘이다. 첫번째 알고리즘이 같은 수를 반복적으로 곱하는 모듈라 곱셈에 대해서는 매우 좋은 성능을 보이는 반면, 모듈라 제곱에는 사용할 수가 없다. 따라서, 모듈라 제곱을 빠르게 수행하는 두번째 알고리즘과 결합하여 사용할 경우, 최종목적인 모듈라 멱승 연산을 빠르게 수행할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 작은윈도우 기법을 사용해서 모듈라 멱승 연산을 수행하는 과정을 간략하게 기술한다. 3장에서는 기본적인 모듈라 곱셈 알고리즘과 본 논문에서 제안하는 알고리즘들을 설명한다. 4장에서는 기존의 모듈라 곱셈 알고리즘과 본 논문에서 제안하는 알고리즘들의 성능을 분석한다. 그리고, 5장에서는 본 논문에서 제안하는 알고리즘들의 성능을 기존의 알고리즘들과 비교한다. 6장에서는 구현결과를 제시하고 이에 대해 논한다. 머지막으로 7장에서는결론을 내리고 앞으로의 연구방향을 제시함으로써 끝맺는다.

## 2 모듈라 멱승 연산과정

RSA와 같은 암호시스템들에서 사용되는 모듈라 멱승 연산은 다음과 같이 정의된다.

**정의 2.1** $C = M^E \bmod N \quad ( b^{k-1} \le E < N < b^k, 0 \le M < N )$.

$M$은 메세지를 의미하고, $E$는 키(key)를 의미한다. 따라서, $M$은 항상 다른 것이 되지만 $E$는 같은 것이 반복되어 사용되기때문에 빠르게 연산할 수 있는 순서를 구해서 사용하는 것이 효과적이다. 이 때에 사용되는 순서를 덧셈사슬이라고 하며, 그 정의는 다음과 같다.

**정의 2.2** 임의의 양의 정수 $E$에 대한 덧셈사슬(*addition-chain*)은 다음과 같은 성질을 지닌 일련의 수열 $a_0, a_1, ..., a_l$이다.

1. $a_0 = 1$, $a_l = E$.
2. $a_i = a_j + a_k, 0 \le j \le k < i \le l$.

위의 정의 2.1과 2.2에 의하면, 모듈라 멱승 연산과정은 다음과 같이 나열될 수 있다.

$$\begin{aligned} C_0 &= M^{a_0} \bmod N = M, \\ C_1 &= M^{a_1} \bmod N, \end{aligned} \tag{1}$$

$$\begin{aligned} C_2 &= M^{a_2} \bmod N, \\ &\vdots \\ C_{l-1} &= M^{a_{l-1}} \bmod N, \\ C_l &= M^{a_l} \bmod N \quad = M^E \bmod N (= C). \end{aligned}$$

덧셈사슬의 정의에 의하면, 위 식(1)의 각 단계는 다음과 같은 관계에 의해서 연결된다.

$$\begin{aligned} C_i &= (C_j \times C_k) \bmod N, \\ & 0 \le j \le k < i \le l. \end{aligned} \tag{2}$$

위의 식들 (1)과 (2)에서 알 수 있듯이, 덧셈사슬의 길이가 짧을 수록 모듈라 멱승 연산을 수행하는 데에 필요한 모듈라 곱셈의 횟수가 줄어든다. 따라서, 보다 짧은 덧셈사슬을 구할 수 있는 알고리즘을 개발하기 위한 노력들이 있어왔다[5, 6, 7, 8, 9, 10, 11]. 그러한 알고리즘들 중에서 작은윈도우 기법[6]이 분석가능한 가장 짧은 덧셈사슬을 구한다[7, 5]. 사실, 가장 짧은 덧셈사슬을 구하는 알고리즘은 [5]에서 Bos 등이 제안한 휴리스틱 알고리즘이다. 그러나, [5]의 휴리스틱 알고리즘을 사용해서 구할 수 있는 덧셈사슬의 길이와 작은윈도우 기법을 사용해서 구해지는 덧셈사슬의 길이의 차이는 극히 작은데 반해, 휴리스틱 알고리즘은 매우 복잡하다. 또한 본 논문에서 제안하는 알고리즘들은 [5]의 휴리스틱 알고리즘과 작은윈도우 기법 모두에 적용이 가능하다. 따라서, 본 논문에서는 설명의 편의를 위해 작은윈도우 기법을 가정한다. 이러한 작은윈도우 기법에 의해서 구해지는 덧셈사슬을 이용하면, 다음과 같은 두 가지 종류의 연산을 반복함으로써 모듈라 멱승 연산이 수행된다. 다음 식에서 $w$는 작은윈도우 기법에 사용되는 윈도우의 크기를 나타낸다.

$$C_i = C_{i-1} \times C_\alpha \bmod N, 0 < i \le l, 0 \le \alpha < 2^{w-1}. \tag{3}$$

$$C_i = C_{i-1}^2 \bmod N, 0 < i \le l. \tag{4}$$

## 3 모듈라 곱셈 알고리즘

이번 절에서는 앞에서 기술한 식(3)과 (4)를 계산하는 기본적인 모듈라 곱셈 방법을 설명하고, 두 개의 새로운 알고리즘들을 제안한다.본 논문에서는 앞으로 편의상 식(3)을 윈도우 모듈라 곱셈(window modular multiplication)이라 하고, 식(4)를 모듈라 제곱(modular squaring)으로 부른다. 그리고 다정도 정수(multiple-precision integer) $C_i$를 다음과 같이 $b$-진법으로 표현한다.

$$C_i = \sum_{j=0}^{k-1} c_{i,j} b^j. \tag{5}$$

### 3.1 기본적인 방법(Multiply & Reduction)

모듈라 곱셈을 수행하는 기본적인 방법은 곱셈을 하고나서, 그 결과에 모듈라 감소를 수행함으로써 나머지(residue)를 구하는 것이다. 식(3)과 (4)는 각각 식(6)과 (7)에 의해 계산된다.

$$\begin{aligned} C_i &= C_{i-1} \times M^{a_\alpha} \bmod N \\ &= (C_{i-1} \times (M^{a_\alpha} \bmod N)) \bmod N \\ &= (C_{i-1} \times T[\alpha]) \bmod N, \\ & T[\alpha] = M^{a_\alpha} \bmod N, 0 \le \alpha < 2^{w-1}. \end{aligned} \tag{6}$$

$$\begin{aligned} C_i &= C_{i-1}^2 \bmod N \\ &= (C_{i-1}^2) \bmod N. \end{aligned} \tag{7}$$

### 3.2 알고리즘 1

앞에서 언급한 바대로, 본 논문에서 제안하는 첫번째 알고리즘은 모듈라 멱승시 작은윈도우 기법을 사용하는 것으로 가정한다. 작은 윈도우 기법의 특징은, 식(3)에서 알 수 있듯이, 윈도우 모듈라 곱셈의 경우 두 피연산자 중 하나는 윈도우에 들어있는 제한된 갯수의 수들 중 하나라는 점이다. 이러한 특성을 이용하면, 식(3)은 다음 식(8)과 같이 전개될 수 있고, 또한 식(8)을 통해서 계산될 수 있다.

$$\begin{aligned} C_i &= C_{i-1} \times M^{\alpha} \bmod N \\ &= (\textstyle\sum_{j=0}^{k-1} c_{i-1,j} b^j) \times M^{\alpha} \bmod N \\ &= (\textstyle\sum_{j=0}^{k-1} c_{i-1,j} \times (b^j \times M^{\alpha} \bmod N)) \bmod N \\ &= (\textstyle\sum_{j=0}^{k-1} c_{i-1,j} \times T[\alpha][j]) \bmod N, \\ & \quad T[\alpha][j] = b^j \times M^{\alpha} \bmod N, 0 \le \alpha < 2^{w-1}. \end{aligned} \tag{8}$$

위의 식(8)에서 테이블 $T$는 다음과 같은 식을 이용해서, 간단하게 구해질 수 있다.

$$\begin{aligned} T[\alpha][0] &= M^{\alpha} \bmod N, \\ T[\alpha][j] &= (T[\alpha][j-1] \times b) \bmod N, \\ & \quad 0 \le \alpha < 2^{w-1}, 0 < j \le k-1. \end{aligned} \tag{9}$$

### 3.3 알고리즘 2

모듈라 멱승 연산을 수행하기 위해서는 일반적으로 모듈라 제곱이 서로다른 두 수의 모듈라 곱셈보다 많이 필요하다. 게다가, 작은윈도우 기법을 이용하게 되면 모듈라 제곱이 윈도우 모듈라 곱셈에 비해 월등히 많이 필요하다. 그런데, 이전 절에서 설명했던 알고리즘 1이 매우 빠르기는 하지만 모듈라 제곱에는 사용될 수 없다. 따라서, 모듈라 멱승 연산을 빠르게 수행하기 위해서는 모듈라 제곱을 빠르게 수행할 수 있는 알고리즘이 필요하다.

이번 절에서는 모듈라 제곱을 빠르게 수행할 수 있는 알고리즘을 제안한다. 이번 절의 알고리즘은 비교적 작은 수에 대해서는 모듈라 감소가 보다 빠르게 수행될 수 있다는 점을 이용한다. 식(4)는 다음과 같이 전개될 수 있다.

$$\begin{aligned} C_i &= C_{i-1}^2 \bmod N \\ &= (c_{i-1,k-1}b^{k-1} + c_{i-1,k-1}b^{k-1} + \cdots + c_{i-1,1}b^1 + c_{i-1,0})^2 \bmod N \\ &= ((\cdots(c_{i-1,k-1}b + c_{i-1,k-2})b + \cdots + c_{i-1,1})b + c_{i-1,0})^2 \bmod N \\ &= (C_{i-1}^{(1)}b + c_{i-1,0})^2 \bmod N \\ &= ((C_{i-1}^{(1)})^2 b^2 + 2c_{i-1,0}C_{i-1}^{(1)}b + c_{i-1,0}^2) \bmod N \\ &= (((C_{i-1}^{(1)})^2 \bmod N)b^2 + 2c_{i-1,0}C_{i-1}^{(1)}b + c_{i-1,0}^2) \bmod N, \\ & \quad C_{i-1}^{(1)} = (\cdots(c_{i-1,k-1}b + c_{i-1,k-2})b + \cdots + c_2)b + c_1. \end{aligned} \tag{10}$$

위의 식(10)의 우변항들 중에서 처음과 마지막 항을 살펴보면, $C_{i-1}^2 \bmod N$과 $(C_{i-1}^{(1)})^2 \bmod N$이 각각 들어 있는데, 이들은 같은 형태를 가진다. 따라서, 재귀적으로(recursively) 계산할 수 있다. 게다가, 위의 식(10)에서 $C_{i-1}^{(1)}$은 $C_{i-1}$을 오른쪽으로 한 자릿수 만큼 이동(shift)시킨 값이다. 따라서, 재귀함수호출(recursive function call) 을 $\frac{k}{2}$번 반복하여 생기는 최종의

$C_{i-1}^{(\frac{k}{2})}$의 값은 다음과 같다.

$$C_{i-1}^{(\frac{k}{2})} = \sum_{j=0}^{\frac{k}{2}-1} c_{i-1,j+\frac{k}{2}} b^j. \tag{11}$$

위의 식(11)에서 알 수 있듯이, $\frac{k}{2}$번을 재귀호출(recursive call)한 결과인 $C_{i-1}^{(\frac{k}{2})}$는 $\frac{k}{2}$자릿수로서, 제곱을 해도 $k$자리를 넘지 않음을 알 수 있다. 따라서, 다음과 같은 연산을 $\frac{k}{2}$번 수행하면 모듈라 곱셈이 이루어진다.

$$\begin{aligned} &(C_{i-1}^{(j-1)})^2 \bmod N \\ &= (((C_{i-1}^{(j)})^2 \bmod N)b^2 + 2c_{i-1,0}C_{i-1}^{(j)}b + c_{i-1,0}^2) \bmod N, \\ &\quad 1 \le j \le \tfrac{k}{2}, C_{i-1}^{(0)} = C_{i-1}. \end{aligned} \tag{12}$$

위의 식(12)를 계산할 때에, 모듈라 감소가 필요한 부분은 $((C_{i-1}^{(j)})^2 \bmod N)b^2$ 뿐이다. 이는 이미 구해져 있는 $(C_{i-1}^{(j)})^2 \bmod N$을 두 자릿수만큼 왼쪽으로 이동 시키고, 그에 대한 모듈라 감소를 수행함으로써 이루어진다. 두 자릿수만큼 왼쪽으로 이동한 결과에 모듈라 감소를 수행하는 것은 테이블을 이용한 뺄셈만으로 이루어 질 수 있다. 다음 식(13)에 그 과정이 나타나 있다.

$$\begin{aligned} &(((C_{i-1}^{(j)})^2 \bmod N)b^2) \bmod N \\ &= (((((C_{i-1}^{(j)})^2 \bmod N)\log s - T[m_0])\log s - T[m_1])\cdots)\log s - T[m_{t-1}], \\ &\quad T[m_x] = m_x \times N, 0 \le m_x < s, 0 \le x < t, t = \tfrac{\log b}{\log s}. \end{aligned} \tag{13}$$

# 4 시간 복잡도

기존의 모듈라 곱셈 알고리즘들과 본 논문에서 제안한 알고리즘들의 시간복잡도를 계산하고, 이들을 비교한다. 시간복잡도의 척도가 되는 것은 각 알고리즘들이 수행되는 데에 필요한 단정도 곱셈의 횟수이다.

## 4.1 기본적인 방법

식(5)와 같이 표현되는 큰 두 정수의 곱셈에 필요한 단정도 곱셈의 갯수는 $k^2$이다. 또한, 현재 알려진 모듈라 감소 알고리즘들은 모두 $k(k+c)$($c$는 알고리즘에 따른 상수로서, 고전적(classical) 알고리즘의 경우는 '2', Barrett의 경우는 '4', 그리고 Montgomery의 경우는 '1'이다[14].)번의 단정도 곱셈을 필요로 한다. 따라서, 모듈라 곱셈을 수행하는 데에 필요한 총 단정도 곱셈의 갯수는 다음과 같다.

$$2k^2 + ck$$

## 4.2 알고리즘 1

식(8)에 의하면, 식(3)을 계산할 때에 모듈라 감소 연산을 따로 수행할 필요가 없다. 단지 중간단계의 모든 값을 더한 결과가 $k$자리를 넘어가게 되므로 이를 줄이기 위한 몇 번의 과외연산만 추가로 필요할 뿐이다.

먼저, 각 자릿수와 해당 자릿수 단위의 모듈라 감소 결과를 곱하는 데에 $k$번의 단정도 곱셈이 필요하고, 자릿수가 $k$이므로 모두 $k^2$번이 필요하다. 그런데, 각 중간결과들을 더한

값이 $N$보다 약간 커지게 된다. 그러나, 그 커지는 비율이 아주 작은 크기이므로 다음절의 알고리즘 2에 사용될 테이블을 이용하면 단정도 곱셈을 사용하지 않고 최종 나머지 값을 구할 수 있다. 따라서, 식(8)을 계산하는 데에 모두

$$k^2 \tag{14}$$

번의 단정도 곱셈이 필요하다.

다음으로 테이블을 계산하는 데에 필요한 단정도 곱셈의 갯수를 생각해 본다. 식(9)를 살펴보면, $T[\alpha][0]$는 이전 단계의 모듈라 곱셈에 의해서 구해진 값이므로 아무런 부가의 연산이 필요없다. 그 다음의 연산들은 모두 한 번의 왼쪽으로의 이동(left-shift)과 모듈라 감소를 필요로 하게 된다. 그런데, 이 때의 모듈라 감소는 모두 $N$보다 한 자릿수가 큰 수에 대한 모듈라 감소이므로 $k$번의 단정도 곱셈으로 계산된다. 따라서, 테이블의 모든 값을 계산하는 데에 필요한 단정도 곱셈의 갯수는 다음과 같다. $w$는 작은윈도우 기법에서의 윈도우의 크기이다.

$$2^{w-1} \times k(k-1) \tag{15}$$

### 4.3 알고리즘 2

식(12)를 살펴보면 알고리즘 2를 이용해서 모듈라 제곱을 수행하는 데에 필요한 단정도 곱셈의 갯수를 계산할 수 있다. 먼저 재귀함수호출의 끝냄조건(terminate condition)에 도달한 후에, 식(11)을 계산하는 데에 $\frac{k^2+2k}{8}$번의 단정도 곱셈이 필요하다. 그리고, 식(12)의 우변 중에서, 첫째항은 식(13)에서 알 수 있듯이 곱셈을 필요로 하지 않는다. 그리고, 둘째항은 모든 recursion 동안에 총 $\frac{3k^2-2k}{8}(=\sum_{j=\frac{k}{2}}^{k-1} j)$ 번의 단정도 곱셈을 필요로 한다. 마지막으로 세째항은 각 단계마다 한 번씩의 곱셈을 하므로 총 $k/2$번의 단정도 곱셈을 필요로 하게 된다. 따라서, 식(4)를 알고리즘 2를 이용해서 계산하는 데에 필요한 단정도 곱셈의 횟수는 다음과 같다.

$$\frac{k^2+k}{2} \tag{16}$$

식(13)의 테이블 $T$는 덧셈을 통해서 만들어 질 수 있으므로, 테이블 만드는 데에는 단정도 곱셈이 필요하지 않다.

### 4.4 비교평가

지금까지 기존의 기본적인 모듈라 곱셈 알고리즘(Multiply & Reduction)과 본 논문에서 제안한 알고리즘들의 시간복잡도를 계산하였다. 그러나, 기존의 모듈라 곱셈 알고리즘들 중에는 3.1절에서 설명한 기본적인 방법 이외에도 다정도 곱셈과 모듈라 감소 연산을 하나의 연산으로 간주하는 알고리즘들과, 사전계산(pre-computation) 테이블을 이용해서 필요한 단정도 곱셈의 횟수를 줄이려는 방법들이 있다[15, 13, 18, 11, 16]. 이들을 간략하게 살펴본다.

[11]과 [15]에서는 이진 방법(binary method)[6]으로 모듈라 멱승 연산을 수행할 때에 효율적으로 모듈라 곱셈을 수행할 수 있는 알고리즘이 제안되었다. 그러나, 이진 방법은 작은 윈도우 기법에 비해 그 성능이 현저히 떨어진다. 따라서, 전체 모듈라 멱승 연산을 수행하는 데에는 큰 도움이 되지 못한다. [11]과 [13]에서는 나머지들의 합(sums of residue)들을 재귀적으로 더하면서 부분적인 모듈라 감소 연산을 수행하는 모듈라 곱셈 알고리즘이 제안되었다. 그러나, 모듈라 감소를 수행하기 위해서 $k^2$번의 단정도 곱셈이 필요하고, 완전한 모듈라 감소 연산이 아니기때문에, 최종결과값을 얻기위한 과외연산(overhead calculation)이 필요하여 기본적인 방법과 비슷한 횟수의 모듈라 곱셈을 필요로 한다. [16]에서 제안된 방법은 연산에 필요한 메모리의 양을 줄이는 방법들이다. 계산량은 기본적인 방법과 거의 동

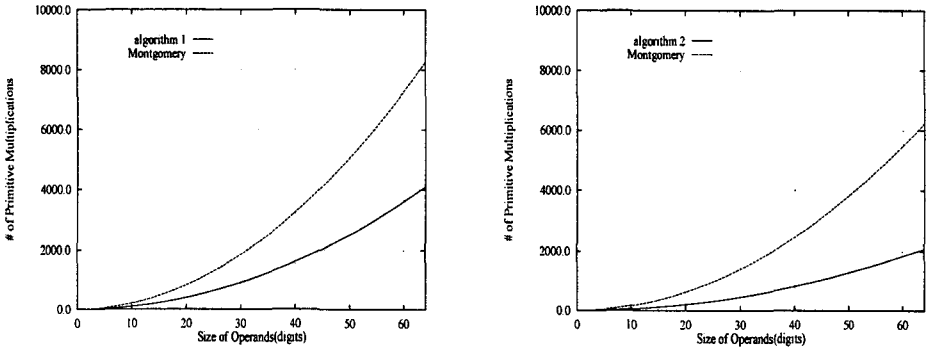
일하다.

지금까지 살펴본 바와 같이, 기존의 사전계산 테이블 이용방법이나 곱셈과 모듈라 감소 연산을 하나의 연산으로 간주하는 방법들은, 3.1절의 기본적인 방법에 비해, 필요한 단정도 곱셈의 횟수에서 볼 때 큰 차이가 없다. 그리고, 기존의 모듈라 감소 알고리즘들 중에서 가장 좋은 것은 Montgomery 알고리즘이다[14]. 따라서, 본 논문에서는 알고리즘 1과 2를 Montogomery 알고리즘을 모듈라 감소에 사용하는 기본적인 방법과 비교한다.

앞에서 분석한 시간복잡도를 비교한 결과가 표1과 그림1에 나타나 있다. 알고리즘 1과 2에서 테이블을 만드는 데에 걸리는 시간과 Montgomery 알고리즘에서의 사전계산과 사후계산(post-calculation)에 걸리는 시간은 고려하지 않았다. 왜냐하면, 한 번의 모듈라 멱승연산을 위해서는 많은 모듈라 곱셈이 필요하기 때문에, 과외연산들이 차지하는 비중은 극히 미미하기때문이다. 참고로 하기위해 모듈라 감소 연산을 필요로하지 않는 일반 다정도 곱셈(ordinary multiple-precision multiplication)과 일반 다정도 제곱(ordinary multiple-precision squaring)에 필요한 연산횟수도 함께 표시하였다. 표1의 $s$는 식(13)에 나타난 $s$이다.

| 알고리즘 | | 단정도 곱셈 횟수 | 메모리 필요량 |
|---|---|---|---|
| 윈도우 모듈라 곱셈 | Montgomery | $2k^2+k$ | $2^{w-1}$ |
| | 알고리즘 1 | $k^2$ | $k \times 2^{w-1}$ |
| | 일반 곱셈 | $k^2$ | $2^{w-1}$ |
| 모듈라 제곱 | Montgomery | $\frac{3}{2}(k^2+k)$ | 0 |
| | 알고리즘 2 | $\frac{1}{2}(k^2+k)$ | $k \times s$ |
| | 일반 제곱 | $\frac{1}{2}(k^2+k)$ | 0 |

표 1: 각 알고리즘들의 시간복잡도와 메모리 필요량, 메모리 필요량의 단위는 피연산자의 갯수




(a)윈도우 모듈라 곱셈 (b)모듈라 제곱

그림 1: 각 알고리즘들의 시간복잡도

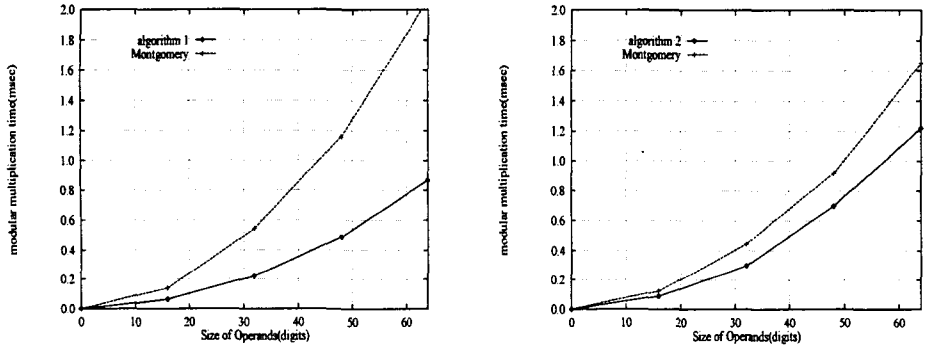
표1과 그림1에서 알 수 있듯이 알고리즘 1과 2에서 필요한 단정도 곱셈의 갯수는 일반 다정도 곱셈과 일반 다정도 제곱에 필요한 그것과 동일하다. 즉, 모듈라 감소 연산을 위한 별도의 단정도 곱셈을 필요로 하지 않는다. 알고리즘 1은 사전계산을 통해서, 그리고 알고리즘 2는 산전계산과 뺄셈을 통해서 모듈라 감소 연산에 필요한 단정도 곱셈을 없앴다.

## 5 구현 및 논의

본 논문에서 제안하는 알고리즘 1과 2를 사용하여 모듈라 곱셈 연산을 수행하는 프로그램을 실제로 구현하여, 그 연산시간을 측정한 결과가 표2에 나타나 있다. 비교대상은 [18]에 기술된 개선된 Montgomery 알고리즘이다. 사용된 시스템은 Pentium-90 마이크로프로세서를 사용하는 PC이며, C 언어로 구현해서 Watcom C(version 10.0) 컴파일러로 컴파일하여 수행한 결과이다. 16-bit를 한 자릿수로 삼았다($b = 2^{16}$).

| 알고리즘 | | 수행시간(msec.) | | | |
|---|---|---|---|---|---|
| | | 256-bit | 512-bit | 768-bit | 1024-bit |
| 윈도우 모듈라 곱셈 | Montgomery | 0.137 | 0.544 | 1.16 | 2.07 |
| | 알고리즘 1 | 0.0604 | 0.220 | 0.489 | 0.868 |
| 모듈라 제곱 | Montgomery | 0.121 | 0.445 | 0.917 | 1.65 |
| | 알고리즘 2 | 0.0851 | 0.297 | 0.698 | 1.22 |

표 2: 각 알고리즘들의 실제 수행시간




(a)윈도우 모듈라 곱셈 (b)모듈라 제곱

그림 2: 각 알고리즘들의 실제 수행시간

표1에 의하면, 알고리즘 1과 2는 기존의 Montgomery 알고리즘에 비해 각각 약 2배와 3배 빠르다. 표2과 그림2(a)에 나타나 있듯이 알고리즘 1의 경우에는 실제 연산시간이 표1과 그림1(a)에 일치한다. 그러나, 알고리즘 2의 경우에는 그렇지 못하다. 그 이유는 표1이 모듈라 곱셈에 필요한 단정도 곱셈의 횟수만을 계산한 것이기때문이다. 일반 범용 프로세서에서 곱셈이 덧셈에 비해 많은 시간이 걸리는 것은 사실이지만, 실제 시스템에서의 연산시간을 예측하기 위해서는 덧셈횟수도 고려해야 한다. 덧셈횟수까지 포함해서 각 알고리즘에 대한 시간복잡도를 계산한 결과가 표3에 나타나 있다. 표3에서 $r$은 덧셈에 필요한 시간을 단정도 곱셈의 횟수로 나타내기 위한 상수로서 다음과 같이 정의 되며, 알고리즘이 구현되는 시스템에 따라 결정된다.

**정의 5.1** $r$ = (단정도 덧셈에 필요한 시간)/( 단정도 곱셈에 필요한 시간)

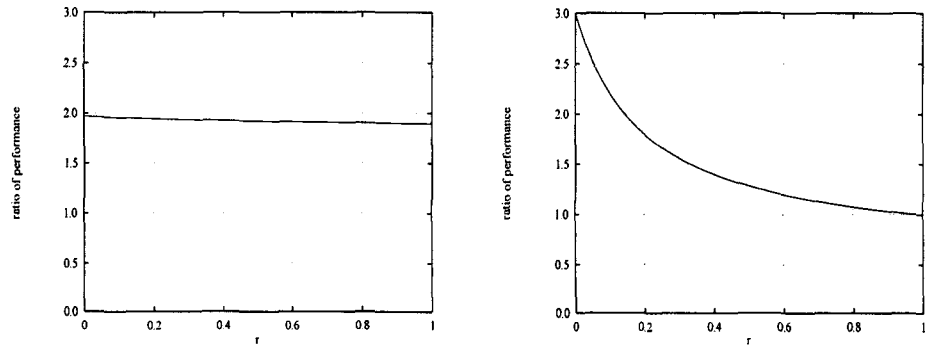
그림3은 표3에 대한 그래프로서, $r$에 따른 각 알고리즘들의 Montgomery 알고리즘에 대한 성능비를 나타낸 것이다. 성능비는 다음과 같이 정의 된다.

| 알고리즘 | | 단정도 곱셈 횟수 |
|---|---|---|
| 윈도우 모듈라 곱셈 | Montgomery | $2k^2+k+r\times(2k^2)$ |
| | 알고리즘 1 | $k^2+r\times(k^2+3(k+1))$ |
| 모듈라 제곱 | Montgomery | $\frac{3}{2}(k^2+k)+r\times(\frac{3}{2}k^2+k)$ |
| | 알고리즘 2 | $\frac{1}{2}(k^2+k)+r\times\frac{5}{2}(k^2+k)$ |

표 3: 덧셈까지 포함한 시간복잡도

**정의 5.2** 알고리즘 (가)의 알고리즘 (나)에 대한 성능비

$$= \frac{\text{알고리즘 (나)의 시간복잡도}}{\text{알고리즘 (가)의 시간복잡도}}$$




(a) 윈도우 모듈라 곱셈 (b) 모듈라 제곱

그림 3: $r$에 따른 성능비의 변화

그림3(a)에 나타난 바에 의하면, 알고리즘1의 경우에는 $r$이 달라지더라도 성능비는 거의 변하지 않는다. 이는 어떤 시스템에서도 알고리즘 1이 기존의 Montgomery 알고리즘에 비해 2배가 빠름을 의미한다. 반면에 그림3(b)를 보게되면, $r$에 따라서 성능비가 많이 달라진다. 즉, 알고리즘이 구현되는 시스템에 따라서, 알고리즘 2의 성능이 달라지게 된다. Pentium PC의 경우, $r$이 약 0.37로서 성능비는 1.43이 된다. 이는 알고리즘 2가 기존의 Montgomery 알고리즘에 비해 약 30%정도 속도가 빠름을 의미한다.

## 6 결론

본 논문에서는 소프트웨어적으로 모듈라 곱셈을 빠르게 수행할 수 있는 알고리즘 두 개를 제안하였다. 하나는 작은윈도우 기법을 사용해서 구해지는 덧셈사슬의 특성을 이용하는 알고리즘이고, 다른하나는 다정도 제곱 연산의 순서를 조절하여 모듈라 감소 연산을 용이하게 함으로써 수행속도를 빠르게 하는 알고리즘이다. 이 두 알고리즘은 모듈라 감소를 위한 별도의 단정도 곱셈 없이 모듈라 곱셈 연산을 수행한다. 또한 본 논문에서는, Pentium PC 환경에서 알고리즘 1과 2가 기존의 Montgomery 알고리즘에 비해 각각 약 100%와 30%의 성능향상을 얻을 수 있음을 보였다.

## 참고 문헌


[1] W.Diffie and M.E.Hellman, "New directions in cryptography," *IEEE Trans. Computers*, vol. IT-22, pp. 644–654, June 1976.

[2] W. Diffie, "The first ten years of public-key cryptography," in *Proceeding of the IEEE*, vol. 76,NO.5, pp. 560–576, May 1988.

[3] R.L.Rivest, A.Shamir, and L.Adleman, "A method for obtaining digital signatures and public key cryptosystems," *CACM*, vol. 21, pp. 120–126, 1978.

[4] T.ElGmal, "A public-key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, 1985.

[5] J. Bos and M. Coster, "Addition chain heuristics," in *Crypto'89*, pp. 400–407, 1989.

[6] D.E.Knuth, *The art of computer programming Vol.2.* Addition-Wesley,Inc., 1981.

[7] M.J.Coster, *Some algorithms on addition chains and their complexity.* CWI Report CS-R9024, 1990.

[8] Y.Yacobi, "Exponentiating faster with addition chains," in *Eurocrypt'90*, 1991.

[9] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM J. Comput.*, vol. 10, pp. 638–646, August 1981.

[10] J.Jedwab and C.J.Mitchell, "Minimum weight modified signed-digit representations and fast exponentiation," *Electronics Letters*, vol. 25, pp. 1171–1172, 1989.

[11] A.Selby and C.Mitcheil, "Algorithms for software implementations of RSA," *IEE Proceedings(E)*, vol. 136, pp. 166–170, MAY 1989.

[12] P. L.Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, 1985.

[13] P.Findlay and B.Johnson, "Modular exponentiation using recursive sums of residues," in *Crypto'89*, 1990.

[14] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Crypto'93*, 1994.

[15] S. Kawamura, K. Takabayashi, and A. Shimbo, "A fast modular exponentiation algorithm," *IEICE Transactions.*, vol. E-74, pp. 2136–2142, August 1991.

[16] H. Morita and C. Yang, "A modular-multiplication algorithm using lookahead determination," *IEICE Trans. Fundamentals*, vol. E76-A, pp. 70–77, January 1993.

[17] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Crypto'86*, pp. 311–323, 1987.

[18] S.R.Dusse and B.S.Kaliski, "A cryptographic library for the motorola DSP56000," in *Eurocrypt'90*, pp. 230–244, 1991.