通信情報保護學會論文誌
第 7 卷 第 2 號, 1997. 6

모듈라 멱승 연산의 빠른 수행을 위한 덧셈사슬 휴리스틱과 모듈라 곱셈 알고리즘들

홍 성 민*, 오 상 엽*, 윤 현 수*

An Addition-Chain Heuristics and Two Modular Multiplication Algorithms for Fast Modular Exponentiation

Seong-Min Hong, Sangyeop Oh, Hyunsoo Yoon

요 약

모듈라 멱승 연산($M^E \bmod N$)은 공개키 암호시스템에 있어서 가장 기본적이고 중요한 연산들 중 하나이다. 그런데 이는 512-비트 이상의 정수들과 같이 매우 큰 수들을 다루기때문에, 수행속도가 느려서 빠른 연산 알고리즘을 필요로 한다. 모듈라 멱승 연산은 모듈라 곱셈의 반복 수행으로 이루어져 있고, 이 때의 반복횟수는 지수(E)에 대한 덧셈사슬의 길이에 의해 결정된다. 따라서, 모듈라 멱승 연산을 빠르게 수행하기 위한 방법에는 두 가지가 있을 수 있다. 하나는 보다 짧은 덧셈사슬을 구함으로써 모듈라 곱셈의 반복횟수를 줄이는 것이고, 다른 하나는 각각의 모듈라 곱셈을 빠르게 수행하는 것이다. 본 논문에서는 하나의 덧셈사슬 휴리스틱과 두 개의 모듈라 곱셈 알고리즘들을 제안한다. 두 개의 모듈라 곱셈 알고리즘들 중 하나는 서로 다른 두 수들 간의 모듈라 곱셈을 빠르게 수행하기 위한 것이고, 다른 하나는 모듈라 제곱을 빠르게 수행하기 위한 것이다. 본 논문에서 제안하는 덧셈사슬 휴리스틱은 기존의 알고리즘들보다 짧은 덧셈사슬을 찾을 수 있다. 본 논문에서 제안하는 모듈라 곱셈 알고리즘들은 기존의 알고리즘들 보다 1/2 이하의 단정도 곱셈 만으로 모듈라 곱셈을 수행한다. 실제로 PC에서 구현하여 수행한 결과, 기존의 알고리즘들 중 가장 좋은 성능을 보이는 Montgomery 알고리즘에 비해 30~50%의 성능향상을 보인다.

Abstract

A modular exponentiation($M^E \bmod N$) is one of the most important operations in public-key cryptography. However, it takes much time because the modular exponentiation deals with very large operands as 512-bit integers. Modular exponentiation is composed of repetition of modular multiplications, and the number of repetition is the same as the length of the addition-chain of the

* 한국과학기술원 전산학과

exponent(E). Therefore, we can reduce the execution time of modular exponentiation by finding shorter addition-chain(i.e. reducing the number of repetitions) or by reducing the execution time of each modular multiplication. In this paper, we propose an addition-chain heuristics and two fast modular multiplication algorithms. Of two modular multiplication algorithms, one is for modular multiplication between different integers, and the other is for modular squaring. The proposed addition-chain heuristics finds the shortest addition-chain among exisiting algorithms. Two proposed modular multiplication algorithms require single-precision multiplications fewer than 1/2 times of those required for previous algorithms. Implementing on PC, proposed algorithms reduce execution times by 30-50% compared with the Montgomery algorithm, which is the best among previous algorithms.

1. 서 론

모듈라 멱승(modular exponentiation) 연산은 공개키 암호시스템에서 아주 기본적이고 자주 이용되는 연산들 중 하나이다.[1, 2] 그런데, 암호시스템에 이용되는 모듈라 멱승 연산은 그 수행속도가 매우 느리다. 왜냐하면, 실제로 사용될 수 있을만큼의 안전도를 얻기 위해서는 512비트 이상의 아주 큰 수들에 대한 연산이어야 하는데, 이러한 크기의 수는 현재의 컴퓨터 시스템으로는 쉽게 처리할 수 없기 때문이다. 따라서, 암호화와 복호화를 수행하는 데에 많은 시간이 소요되며, 이는 공개키 암호시스템이 좋은 특성을 많이 가지고 있음에도 불구하고 널리 사용되지 못하는 결정적인 이유이다. 따라서, 그동안 많은 학자들에 의해 모듈라 멱승 연산을 빠르게 수행하기 위한 연구들이 이루어져 왔다.[3, 4, 5, 6, 7, 8]

모듈라 멱승 연산은 모듈라 곱셈(modular multiplication)의 반복 수행으로 이루어진다. 따라서, 모듈라 멱승 연산을 빠르게 수행하기 위한 방법에는 두 가지가 있을 수 있다. 한 가지는 모듈라 곱셈의 반복횟수를 줄이는 것으로서, 덧셈사슬(addition-chain)을 이용하는 방법이다. 덧셈사슬이란 모듈라 멱승 연산을 수행하기 위해서 필요한 모듈라 곱셈들의 순서를 지수(exponent)들의 덧셈을 이용해서 나타낸 수열이다. 다른 한 가지는 각각의 모듈라 곱셈을 빠르게 수행하는 것이다.

모듈라 멱승 연산에 필요한 수행시간은 덧셈사슬의 길이에 비례하기 때문에 그동안 덧셈사슬에 대한 많은 연구들이 이루어져 왔다. 우선 가장 직관적인 이진방법(binary method)과 이를 개선한 방법들이[5], [9], [10]에서 제안되었다. 그리고, [5]에는 사전계산(pre-calculation)을 이용하는 작은윈도우(small-window) 기법이 제안되어 있다. Bos-Coster는 [3]에서 큰윈도우(large-window) 기법에 사용되는 휴리스틱(heuristic) 알고리즘들을 제안하였다. 또한, Yacobi는 [11]에서 큰 수들에 대한 빠른 연산과 데이터 압축 사이의 유사성을 이용하는 알고리즘을 제안하였다. 그동안 이러한 많은 알고리즘들이 제안되었으나, 기본적으로 최소 길이의 덧셈사슬을 구하는 문제가 NP-complete이므로 휴리스틱을 이용하는 접근방법이 필요하다.[1, 2]

모듈라 멱승 연산에 필요한 수행시간은 모듈라 곱셈 연산의 수행시간에 비례하기 때문에, 그동안 모듈라 곱셈을 빠르게 수행하기 위한 알고리즘들이 많이 제안되어 왔다. 모듈라 곱셈을 수행하는 기본적 방법(basic method)은 곱셈을 먼저 수행하고나서 그 결과에 모듈라 감소(modular reduction)를 수행하는 것이다. 이러한 기본적 방법에 의해서 모듈라 곱셈을 수행할 경우에 소요되는 연산시간은 모듈라 감소 알고리즘에 따라 결정된다. 따라서, 많은 모듈라 감소 알고리즘들이 제안되었는데, 그 중 대표적인 것으로는 고전적인 방법

(classical method),[5] Barrett 알고리즘,[13] 그리고 Montgomery 알고리즘[14] 등이 있다. 이들은 각각 다른 특성을 지니고 있으나, 일반적인 모듈라 멱승 연산에는 Montgomery 알고리즘이 가장 좋은 성능을 보인다.[6] 기존의 모듈라 곱셈 알고리즘들 중에는 기본적 방법 이외에도 다정도(multiple-precision) 곱셈과 모듈라 감소 연산을 하나의 연산으로 간주하는 알고리즘들과, 사전계산(pre-computation) 테이블을 이용해서 필요한 단정도(single-precision) 곱셈의 횟수를 줄이려는 방법들이 있다.[7, 15, 8, 10, 16] 그러나, 그러한 방법들은 필요한 단정도 곱셈 횟수의 측면에서 볼 때 기본적 방법과 큰 차이가 없다.

본 논문에서는 하나의 덧셈사슬 휴리스틱과 두 개의 알고리즘을 제안한다. 덧셈사슬 휴리스틱은 작은윈도우 방법으로 덧셈사슬을 구할 때에, 보다 짧은 덧셈사슬을 구할 수 있도록 하기 위한 것이다. 두 개의 알고리즘들은 모듈라 곱셈을 빠르게 수행하는 알고리즘들이다. 그들 중 하나는 서로 다른 두 수의 모듈라 곱셈을 빠르게 수행하는 알고리즘이다. 이는 [7]에서 Kawamura 등이 제안한 방법을 작은윈도우 기법에 적용할 수 있도록 확장한 것이다. 다른 하나는 모듈라 제곱(modular squaring)시에 모듈라 감소를 빠르게 수행할 수 있도록 모듈라 제곱의 연산순서(modular multiplication sequence)를 수정한 알고리즘이다. 첫 번째 알고리즘이 서로 다른 두 수의 모듈라 곱셈에 대해서는 매우 좋은 성능을 보이는 반면, 모듈라 제곱에는 사용할 수가 없다. 따라서, 모듈라 제곱을 빠르게 수행하는 두 번째 알고리즘과 결합하여 사용할 경우, 전체 모듈라 멱승 연산을 빠르게 수행할 수 있다.

본 논문에서는 제안된 휴리스틱의 성능을 기존의 덧셈사슬 알고리즘들의 성능과 비교한다. 그리고, 본 논문에서 제안하는 모듈라 곱셈 알고리즘들의 성능을 기존의 모듈라 곱셈 알고리즘들의 성능과 비교한다. 덧셈사슬 알고리즘들의 성능척도는 그 알고리즘을 사용했을 때 구해지는 덧셈사슬의 길이이다. 또한 모듈라 곱셈 알고리즘들의 성능척도는 각 알고리즘들이 수행되기 위해 필요한 단정도 곱셈의 횟수이다. 마지막으로 본 논문에서 제안하는 휴리스틱과 두 개의 알고리즘들을 실제로 구현하여, 그 연산시간을 측정한다. 또한 그 결과를 기존의 모듈라 멱승 연산 알고리즘의 수행시간과 비교한다.

본 논문의 구성은 다음과 같다. 2장에서는 작은윈도우 기법과 기본적 모듈라 곱셈 알고리즘을 사용해서 모듈라 멱승 연산을 수행하는 과정을 간략하게 기술한다. 3장에서는 본 논문에서 제안하는 덧셈사슬 휴리스틱을 설명하고, 4장에서는 새로운 모듈라 곱셈 알고리즘들을 제안한다. 5장에서는 본 논문에서 제안하는 알고리즘들의 성능을 분석하고 기존의 것들과 비교한다. 6장에서는 결론을 내리고 앞으로의 연구방향을 제시한다.

2. 모듈라 멱승 연산과정

RSA, ElGamal 등과 같은 암호시스템들에서 사용되는 모듈라 멱승 연산은 다음과 같이 정의된다.[1, 2]

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

M은 메세지를 의미하고, E는 키(key)를 의미한다. 따라서, M은 항상 다른 값을 갖지만 E는 같은 값으로 고정된 채로 사용되므로, E에 대해서는 빠르게 연산할 수 있는 순서를 구해놓고 이를 이용하는 방법이 효과적이다. 이 연산순서를 덧셈사슬이라고 하며, 다음과 같이 정의된다.

정의 2.1 임의의 양의 정수 E에 대한 덧셈사슬

(addition-chain)은 다음과 같은 성질을 지닌 일련의 수열 a_0, a_1, ... , a_l이다. l은 덧셈사슬의 길이를 의미한다.

1. $a_0 = E$, $a_l = E$.
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.

$$C_0 = M^{a_0} \bmod N = M,$$
$$C_1 = M^{a_1} \bmod N,$$
$$C_2 = M^{a_2} \bmod N,$$
$$C_{l-1} = M^{a_{l-1}} \bmod N,$$
$$C_l = M^{a_l} \bmod N = M^E \bmod N (= C). \quad (1)$$

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

$$C_i = (C_j \times C_k) \bmod N,$$
$$0 \le j \le k < i \le l. \quad (2)$$

위의 식들 (1)과 (2)에서 알 수 있듯이, 덧셈사슬의 길이가 짧을 수록 모듈라 멱승 연산을 수행하는 데에 필요한 모듈라 곱셈의 횟수가 줄어든다. 그러나, 가장 짧은 덧셈사슬을 구하는 문제가 NP-complete임이 증명되었다. 따라서, 보다 짧은 덧셈사슬을 구할 수 있는 알고리즘을 개발하기 위한 노력들이 있어왔다.[3, 5, 4, 11, 9, 10]

그러한 알고리즘들 중에서 작은윈도우 기법[5]이 분석가능한 가장 짧은 덧셈사슬을 구한다.[4, 3] 사실, 가장 짧은 덧셈사슬을 구하는 알고리즘은 [3]에서 Bos-Coster가 제안한 휴리스틱 알고리즘이다. 그러나, [3]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 [3]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 식(3)에서 w는 작은윈도우 기법에 사용되는 윈도우의 크기를 나타낸다.

$$C_i = C_{i-1} \times C_\alpha \bmod N, \quad 0 < i \le l,$$
$$0 \le \alpha < 2^{w-1}. \quad (3)$$
$$C_i = C_{i-1}^2 \bmod N, \quad 0 < i \le l. \quad (4)$$

3. 덧셈사슬 휴리스틱

본 절에서는 작은윈도우 기법을 사용해서 덧셈사슬을 구할 때, 보다 짧은 길이의 덧셈사슬을 구할 수 있는 휴리스틱을 제안한다. 작은윈도우 기법으로 구해지는 덧셈사슬의 길이는 입력값의 이진표현이 몇 개의 윈도우로 나누어지는가에 따라 결정된다. 본 논문에서 제안하는 휴리스틱은 윈도우를 적게 할당할 수 있도록 하기 위한 방법이다.

임의의 정수에 대한 덧셈사슬을 구하는 과정은 주어진 정수가 1이 될 때까지 쪼개 나가는 과정의 역으로 볼 수 있다. 여기서 쪼개 나간다는 것은 다음과 같이 정의 한다.

정의 3.1 정수 a를 쪼개 나간다는 것은 ops의 모든 원소들이 1이 될 때까지, 다음의 두 가지 연산을 반복하는 것을 말한다. 처음에 $ops = \{a\}$에서 시작한다.

- 1이 아닌 ops의 각 원소 a에 대해, $a \leftarrow a/2$를 수행한다.
- 1이 아닌 ops의 각 원소 a에 대해, $a \leftarrow a-x$를 수행하고, $ops \leftarrow ops \cup \{a, x\}$를 수행한다. 여기서 $x < a$이다.

작은윈도우 기법을 이러한 과정으로 생각하면, 3.1과 같이 정리될 수 있다.

알고리즘 3.1 Small_Window_Reverse(a, k)는 a와 k를 인수로 받아서, 크기가 k인 윈도우를 사용해서 작은윈도우 기법으로 a에 대한 덧셈사슬을 구해서 반환하는 함수이다. a는 $e_{n-1}e_{n-2}\cdots e_1e_0$, $e_i=0$ 또는 1, $0\le i<n$이라고 가정한다. α는 덧셈사슬을 집합의 형태로 표현한 것이며, w_a는 a의 최하위(least-significant) k비트에 해당하는 값을 의미한다.

```
1 Small_Window_Reverse(a, k)
2 {
3       α← {1, 2, 3, 5, 7, 9, ... , 2^k-1};
4       α← α∪{a};
5       do{
6         while{a mod 2 = 0){
7               a← a/2;
8               α← α∪{a};
9         }
10        a ← a-w_a;
11      } while{a≥2^(k-1)};
12      return(α);
13 }
```

알고리즘 3.1에서 10번째 줄을 수행하기 직전의 a값은 홀수이다. 왜냐하면 6-9번째 줄을 통해서 2를 포함하는 인수들은 모두 제거되었기 때문이다. 이 때, 10번째 줄을 수행하면, 뺄셈 한 번을 통해서 윈도우 하나를 제거하는 셈이다. 그러나, a가 3의 배수일 경우, a를 3으로 나누어 보면 a-$a/3$의 이진표현이 a-w의 이진표현과 같거나 더 적은 갯수의 윈도우를 포함할 수 있다. a-$a/3$의 이진표현이 a-w의 이진표현과 같은 갯수의 윈도우를 포함하고 있을 경우에, a에서 $a/3$을 빼는 것이 w를 빼는 것보다 빨리 1에 도달할 수 있다. 그러므로, a-$a/3$의 이진표현이 더 적은 갯수의 윈도우를 포함하고 있을 경우에는 당연히 $a/3$을 빼는 것이 더 낫다. 정리하면, 10번째 줄을 수행하기 전에, a-$a/3$의 이진표현이 포함하고 있는 윈도우 갯수와 a-w의 이진표현이 포함하고 있는 윈도우 갯수를 비교해 보아 a-$a/3$의 이진표현이 a-w의 이진표현과 같거나 더 적은 갯수의 윈도우로 분할 되면, a에서 w를 빼는 대신 $a/3$을 빼는 것이 덧셈사슬의 길이를 줄일 수 있다.

1467_{10}에 대한 덧셈사슬을 구하는 과정을 예로 들어 위에서 설명한 과정을 살펴본다. 1467_{10}의 이진표현은 10110111011_2이다. 이를 크기 4인 윈도우를 이용하는 알고리즘 3.1을 이용하면 다음과 같은 모양으로 윈도우가 할당된다.

$$\underline{101}\ \underline{1011}\ \underline{1011}$$

여기에서 알고리즘 3.1의 10번째 줄을 수행하면 그 결과는 다음과 같다.

$$\underline{101}\ \underline{1011}\ 0000 \qquad (5)$$

그러나, 1467_{10}에서 맨 오른쪽 윈도우 $\underline{1011}$에 해당하는 값 11_{10}을 빼는 대신, 1467_{10}을 3으로 나눈 489_{10}를 빼면 978_{10}이 된다. 이에 해당하는 이진표현을 크기 4인 윈도우로 분할하면 다음과 같은 모양새가 된다.

$$\underline{1111}\ 0\ \underline{1001}\ 0 \qquad (6)$$

식(5)와 식(6)을 비교해 본다. 두 식 모두 두 개의 윈도우를 포함하고 있다. 그러나, 덧셈사슬을 구하기 위해서 필요한 이동 - 알고리즘 3.1의 6-9번째 줄에 해당한다. - 의 횟수는 다르다. 식(5)는 총 8번의 이동을 필요로 하는 반면 식(6)은 총 6번의 이동만을 필요로 한다.

지금까지 3으로 나누어지는 수들에 대해 적용할 수 있는 휴리스틱을 설명하였다. 이를 확장하면 5, 9, 17, 33, ... 등으로 나누어지는 수들에도 적용할 수 있다. 즉, $2^\varepsilon+1$(ε은 작은 정

수)인 정수들로 나누어지는 수들의 경우에는 지금까지 설명한 휴리스틱을 적용할 수 있다. 그러나, 그 외의 수들로 나누어지는 수들에 대해서는 적용해도 별 효과가 없다. 왜냐하면, 2^e+1의 꼴이 아닌 정수들로 나누어지는 수는 그 수를 구하기 위해 덧셈이 더 필요하기 때문이다. 이러한 휴리스틱을 이용하면 알고리즘 3.1은 알고리즘 3.2로 전개될 수 있다.

알고리즘 3.2 Small_Window_Heuristics(a, k)는 a와 k를 인수로 받아서, 크기가 k인 윈도우를 사용해서 작은윈도우 기법으로 a에 대한 덧셈사슬을 구해서 반환하는 함수이다. a는 $e_{n-1}e_{n-2}\cdots e_1e_0$, $e_i=0$ 또는 1, $0\le i<n$이라고 가정한다. α는 덧셈사슬을 집합의 형태로 표현한 것이며, w_a는 a의 최하위 k비트에 해당하는 값을 의미한다.

```
1 Small_Window_Heuristics(a, k)
2 {
3       α ← {1, 2, 3, 5, 7, 9, ... ,2^k-1;
4       α ← αU{a};
5         do{
6               while(a mod 2=0=){
7                 a\a/2;
8                 α ← αU{a};
9           }
10          for{m∈{..., 65, 33, 17, 9, 5, 3}) {
11            if{(a mod m=0)
              and{(a/m의 윈도우 갯수
              a<a-w_a의 윈도우 갯수
12              a ← a-a/m;
13              continue;
14           }
15          } \\
16          a ← a-w_a;
17      }while(a ← 2^(k-1));
18 }
```

4. 모듈라 곱셈 알고리즘

본 절에서는 본 논문에서 제안하는 윈도우 모듈라 곱셈 알고리즘과 모듈라 제곱 알고리즘을 설명한다. 윈도우 모듈라 곱셈 알고리즘은 식(3)을 빠르게 수행할 수 있는 알고리즘이고, 모듈라 제곱 알고리즘은 식(4)를 빠르게 수행하는 알고리즘이다. 이 두 알고리즘을 이용하면 모듈라 멱승 연산에 사용되는 모듈라 곱셈들을 빠르게 수행할 수 있다.

본 절에서 사용되는 영문자들은 별도의 언급이 없으면 다음과 같은 의미를 지닌다. 대문자로 표시된 영문자는 다정도정수(multiple-precision integer)를 나타내고, 소문자로 표시된 영문자는 단정도 정수(single-precision integer)를 나타낼 때 사용된다. 예를 들면, 다정도 정수 C는 다음과 같이 표현된다.

$$C=\sum_{j=0}^{k} c_j\times b^k,\ \text{where}\ 0\le c_j<b \qquad (7)$$

현재 안전하다고 여겨지는 C의 크기는 512비트 또는 1024비트이다. 진수(radix)를 나타내는 b는 컴퓨터시스템에서 한 명령어로 수행할 수 있는 정수의 크기로 정해진다. 예를들어 32비트 컴퓨터에서는 2^{16}이 적합하다.

4.1 윈도우 모듈라 곱셈 알고리즘

본 절에서는 식(3)을 빠르게 수행할 수 있는 윈도우 모듈라 곱셈 알고리즘을 제안한다. 본 절에서 제안하는 알고리즘은 Kawamura 등이 [7]에서 제안한 방법을 작은 윈도우 기법에 적용할 수 있도록 확장한 것이다.

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. 다음 식에서 w는 윈도우 크기를 나타낸다.

$$
\begin{aligned}
C_i &= C_{i-1} \times M^{\alpha} \bmod N \\
&= \left(\sum_{j=0}^{k-1} c_{i-1,\ j} b^j\right) \times M^{\alpha} \bmod N \\
&= \left(\sum_{j=0}^{k-1} c_{i-1,\ j} \times (b^j \times s\ M^{\alpha} \bmod N)\right) \bmod N \\
&= \left(\sum_{j=0}^{k-1} c_{i-1,\ j} \times T[\alpha][j]\right) \bmod N,
\end{aligned}
$$

$$
T[\alpha][j] = b^j \times M^{\alpha} \bmod N,\ 0 \le \alpha < 2^{w-1}. \qquad (8)
$$

위의 식(8)에서 테이블 T는 다음 식을 이용해서 간단하게 구할 수 있다.

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

식(8)과 식(9)를 이용해서 윈도우 모듈라 곱셈을 수행하는 방법이 알고리즘 4.1과 알고리즘 4.2에 *C-like* 의사코드(pseudo-code)로 정리되어 있다.

알고리즘 4.1 Window_ mod ular_Multiplication(C, α, N)은 C, α, N을 인수로 받아서, $C \times M^{\alpha} \bmod N$을 반환한다. $T[\alpha][i] = M^{\alpha} \times b^i \bmod N$이고, 알고리즘 4.2에 의해 이미 계산되어 있다고 가정한다. $0 \le \alpha < 2^{w-1}$이고 $0 \le i < k$이다.

```
1 Window_ mod ular_Multiplication(C, α, N)
2 {
3       S ← 0;
4       i ← 0;
5       while(i<k){
6         S ← S+c_i×T[α][i];
7         i ← i+1;
8       }
9       return(S mod N);
10 }
```

알고리즘 4.2 Make_Table(M, N, w)를 인수로 받아서, $T[\alpha][i]$를 계산하여 저장한다. $0 \le \alpha < 2^{w-1}$이고, $0 \le i < k$이다.

```
1 Make_Table(M, N, w)
2 {
3       T[1][0] ← M;
4       T[2][0] ← M^2 mod N;
5       j ← 3;
6       while(j<w){
7         T[j][0] ← (T[j-2][0]×T[2][0]) mod N
8         i ← 1;
9         while(i<k){
10              T[j][i] ← (T[j][i-1]×b) mod
N;
11              i ← i+1;
12        }
13        j ← j+2;
14      }
15      i ← 1;
16      while(i<k){
17        T[0][i] ← (T[0][i-1]×b) mod N;
18        i ← i+1;
19      }
20 }
```

4.2 모듈라 제곱 알고리즘

모듈라 멱승 연산을 수행하기 위해서는 일반적으로 모듈라 제곱이 서로다른 두 수의 모듈라 곱셈보다 많이 필요하다. 게다가, 작은윈

도우 기법을 이용하게 되면 모듈라 제곱이 윈도우 모듈라 곱셈에 비해 월등히 많이 필요하다. 그런데, 4.1절에서 설명했던 윈도우 모듈라 곱셈 알고리즘이 매우 빠르기는 하지만 모듈라 제곱에는 사용될 수 없다. 따라서, 모듈라 멱승 연산을 빠르게 수행하기 위해서는 모듈라 제곱을 빠르게 수행할 수 있는 알고리즘이 별도로 필요하다. 본 절에서는 모듈라 제곱을 빠르게 수행할 수 있는 알고리즘을 제안한다.

본 절의 알고리즘은 비교적 작은 수에 대해서는 모듈라 감소가 보다 빠르게 수행될 수 있다는 점을 이용한다. 식(4)는 다음과 같이 전개될 수 있다.

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

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.

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

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

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

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.

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

식(11)과 식(12), 그리고 식(13)을 이용해서 모듈라 제곱을 수행하는 방법이 알고리즘 4.3과 알고리즘 4.4, 그리고 알고리즘 4.5에 *C-like* 의사코드(pseudo-code)로 정리되어 있다.

알고리즘 4.3 *mod ular_Squaring*(C, N)은 C와 N을 인수로 받아서, $C^2 \bmod N$을 반환한다. $T[i] = (M \times i)\} \bmod N$이고, 알고리즘 4.5에 의해 이미 계산되어 있다고 가정한다. 함수 *Small_Reduction*(S, N)은 알고리즘 4.4에 정리되어 있다. $0 < i < s$이고, $N = \sum_{j=0}^{l} n_j \times b^j$이다.

또한 $C = \sum_{j=0}^{k} c_j \times b^j$이며, $C = \sum_{j=1}^{l} c_j \times b^{j-1}$이다.

```
1 mod ular_Squaring(C, N)
2 {
3     if(k≤l/2)
4             return(C^2);
5     S ← mod ular_Squaring(C', N)
              ×b^2+C'×c_0×b+c_0^2;
6     if(S>b^(k+2))
7             S ← S-N×b^2;
8     return(Small\Reduction(S, N);
9 }
```

알고리즘 4.4 Small_Reduction(S, N)은 S와 N을 인수로 받아서 S mod N을 반환한다. β는 $T[\beta] \leq S < T[\beta+1]$를 만족하는 정수이다.

```
1 Small_Reduction(S, N)
2 {
3     i ← 0;
4     while(i<log s/log b){
5       S ← S-T[β];
6       i ← i+1;
7     }
8     return(S);
9 }
```

알고리즘 4.5 Make_Table(M, s)는 M과 s를 인수로 받아서, $T[i]$를 계산하여 저장한다. $0<i<s$.

```
1 Make_Table(M, s)
2 {
3     T[1] ← M;
4     i ← 2;
5     while(i<s){
6             T[i] ← T[i-1]+M;
7             i ← i+1;
8     }
9 }
```

5. 성능평가

본 장에서는 본 논문에서 제안한 덧셈사슬 휴리스틱과 모듈라 곱셈 알고리즘들의 성능을 평가하고, 기존의 알고리즘들과 비교한다. 덧셈사슬 휴리스틱의 성능은 그 알고리즘을 사용해서 구해지는 덧셈사슬의 길이로 나타낼 수 있다. 모듈라 곱셈 알고리즘의 성능척도로는 알고리즘을 수행하는 데에 필요한 단정도 곱셈의 횟수를 사용한다.

본 절에서는 본 논문에서 제안한 덧셈사슬 알고리즘의 성능을 평가한다. 본 논문에서 제안한 알고리즘은 휴리스틱 알고리즘이므로 체계적인 성능분석이 힘들다. 따라서, 본 논문에서는 알고리즘 3.2를 실제로 구현하여 임의의 정수들에 대해서 덧셈사슬을 구함으로써 성능을 측정한다. 본 절에서 사용할 a, n, k_w 등은 각각 다음과 같은 의미를 지닌다.

- a: 덧셈사슬을 구하고자 하는 임의의 정수
- n: a를 이진수로 표현하기 위해 필요한 비트수, 즉 $n=[\lg a]+1$이다.
- k_w: 윈도우의 크기

우선, 기존의 방법들을 사용해서 덧셈사슬을 구할 때의 성능을 살펴본다. 이진방법[5]으로 구해지는 덧셈사슬의 길이는 평균 $\frac{3}{2}n-1.5$이고, 최악의 경우에 $2n-2$이다. 작은윈도우 기법[4]을 사용하면, 평균길이 $2^{k_w-1}+n-k_w-0.5+\frac{n}{k_w+1}$의 덧셈사슬을 얻을 수 있고 최악의 경우

얻어지는 덧셈사슬의 길이는 $2^{k_w-1}+n-k_w+\frac{n}{k_w}$ 이다. [3]에서 Bos 등이 제안한 휴리스틱을 사용해서 큰윈도우 기법으로 덧셈사슬을 구하면, 512비트의 정수에 대해 평균길이 605의 덧셈사슬을 얻을 수 있다. 그러나, 이는 휴리스틱을 사용하므로 정확한 분석이 어렵다. Morain-Olivos[4,9]의 방법을 사용해서 구해지는 덧셈사슬(덧셈사슬의 정의에 의하면, Morain-Olivos의 방법으로 구해지는 일련의 수열은 덧셈사슬이 아니지만, 이를 이용하면 모듈라 멱승 연산을 수행할 수 있으므로, 덧셈사슬이라고 본다.)의 길이는 $\frac{5}{3}n$을 보장하고, 평균 $\frac{4}{3}n$이 된다. 그리고 [11]에서 Yacobi가 제안한 알고리즘을 사용하면, 평균 길이 $n-(\log n-\log\log n)+1.5(\frac{n}{\log n}+o(\frac{n}{\log n}))$의 덧셈사슬이 구해진다.

본 논문에서는 제안된 덧셈사슬 휴리스틱의 성능을 평가하기 위해 256비트, 512비트, 768비트, 1024비트의 길이를 가지는 임의의 정수 100개씩을 무작위로 생성하여, 각각에 대해 구한 덧셈사슬의 길이를 측정하였다. 임의의 정수를 생성하는 데에는 C library 함수인 "srand()"와 "rand()"를 사용하였다.

본 논문에서 제안한 덧셈사슬 휴리스틱을 사용하면, 길이가 512비트인 임의의 정수에 대해 평균길이 602인 덧셈사슬을 얻을 수 있다. 이는 기존의 어떠한 방법으로 얻을 수 있는 덧셈사슬의 길이보다 작은 길이이다. 표 1에 기존의 알고리즘들과 본 논문에서 제안한 휴리스틱을 사용해서 구할 수 있는 임의의 512비트 정수들에 대한 덧셈사슬의 길이들이 비교되어 있다. 표 \1에서, 오른쪽에 "*"가 붙어 있는 숫자들은 실제로 알고리즘을 구현하여 임의의 정수들에 대해 측정한 결과이고, 그렇지 않은 숫자들은 분석에 의한 결과이다.

그림 1에는 본 논문에서 제안한 휴리스틱으

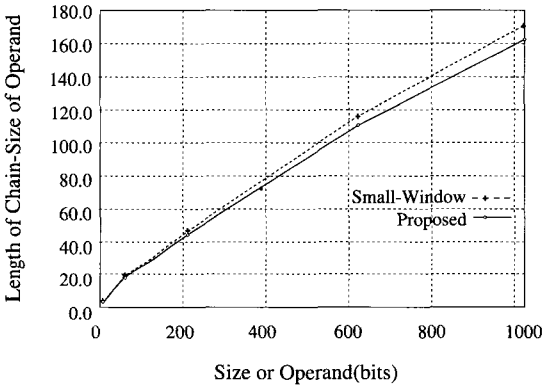


그림 1: 작은 윈도우 기법과 본 논문에서 제안한 알고리즘으로 구할 수 있는 덧셈사슬들의 길이 비교

표 1: 여러 알고리즘들에 의해서 구해지는 $[\lg a]=511$인 a에 대한 덧셈사슬들의 길이, 메모리 필요량의 단위는 피연산자의 갯수}

알고리즘	메모리 필요량	덧셈사슬의 길이	
		필요량	필요량
이진방법[5]	1	1024	768
Morian-Olvis[9]	2	769	683
Yacobi[11]($k_w=5$)	?	?	635
작은 윈도우[5]($k_w=5$)	16	626	609
Bos-Coster[3]	32	610*	605*
제안 알고리즘($k_w=5$)	17	606*	602*

로 구할 수 있는 덧셈사슬들의 평균길이가 입력값의 크기에 따라 나타나 있다. 또한 작은윈도우 기법으로 구해지는 덧셈사슬들의 평균길이도 함께 비교되어 있다. 가로축은 피연산자의 크기 n을 의미하고, 세로축은 덧셈사슬의 길이에서 n을 뺀 값을 나타낸다.

5.1.1 특성

표 1에서 보면, 본 논문에서 제안한 알고리즘이 가장 짧은 덧셈사슬을 구함을 알 수 있다. Bos-Coster의 알고리즘이 평균길이 605인 덧셈사슬을 구함으로써 본 논문의 휴리스틱 알고리즘과 비슷한 길이의 덧셈사슬을 구할 수 있다. 그러나, 이 Bos-Coster의 알고리즘은 사용가능 메모리를 약 두 배정도 더 필요로 한다. 또한 본 논문에서 제안한 휴리스틱 알고리즘은 작은윈도우 기법을 기반으로 하면서 윈도우의 갯수를 줄여나가는 방식을 취하기 때문에, 어떠한 정수에 대해서도 작은윈도우 기법으로 구할 수 있는 덧셈사슬보다 더 짧은 덧셈사슬을 구할 수 있다.

본 논문에서 제안한 휴리스틱을 이용하면, 입력으로 주어지는 정수가 많은 윈도우로 할당될수록 이득을 볼 확률이 높으므로 상당히 안정적인 길이의 덧셈사슬을 구할 수 있다. 이를 검증하기 위해, 난수발생시에 1과 0이 나타날 확률을 조절하여 실행시켜 본 결과가 그림 2에 나타나 있다. 가로축은 입력값의 이진표현에 나타나는 1의 갯수를 의미하며, 세로축은 해당 입력값에 대한 덧셈사슬의 길이를 나타낸다.

윈도우를 이용하는 알고리즘에 의해서 덧셈사슬을 구하면, 사용한 윈도우의 크기에 따라서 구해지는 덧셈사슬의 길이가 달라진다. 이는 본 논문에서 제안한 알고리즘의 경우도 마찬가지이다. 그런데, 윈도우의 크기가 크다는 것은 메모리가 많이 필요하다는 것이다. 즉,

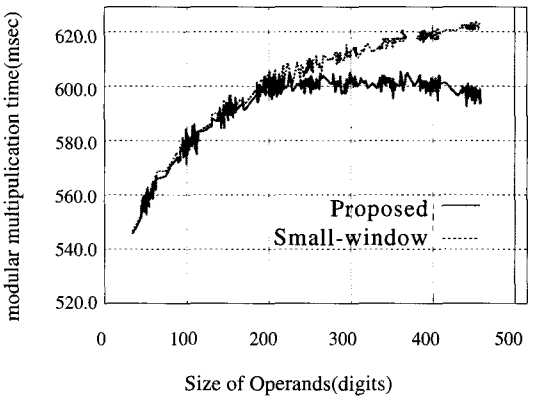


그림 2: 입력값의 이진표현에 들어있는 1의 갯수에 따른 덧셈사슬의 길이변화 비교

모듈라 멱승을 수행하는 데에 필요한 메모리의 양은 2^{k_w}에 비례해서 증가한다. 따라서, 윈도우 크기가 작을수록 메모리에 있어서는 이득을 보게 된다.

512비트 정수에 대해 본 논문에서 제안한 알고리즘이 구하는 덧셈사슬의 길이를 윈도우 크기에 따라 나타낸 그래프가 그림 3에 나타나 있다.

또한 그림 3에는 작은윈도우 기법을 사용해

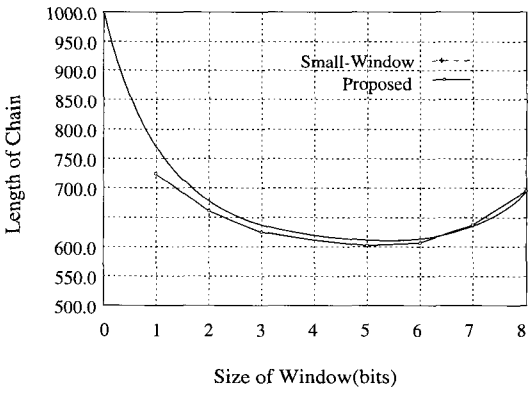


그림 3: 사용한 윈도우의 크기에 따른 덧셈사슬의 길이 변화

서 구할 때의 덧셈사슬의 길이 변화가 함께 나타나 있다. 그림 3에서 볼 수 있듯이 본 논문에서 제안한 알고리즘의 경우 곡선의 기울기가 완만하다. 즉, 사용가능한 메모리의 양이 적은 경우에도 좋은 성능을 보임을 의미한다.

5.2 모듈라 곱셈 알고리즘

본 절에서는 본 논문에서 제안한 모듈라 곱셈 알고리즘들의 성능을 분석하고, 기존의 알고리즘들과 비교한다. 우선 각 알고리즘들을 수행하는 데에 필요한 단정도 곱셈의 횟수를 계산하고, 다음으로 실제 수행시간을 측정한다.

5.2.1 단정도 곱셈의 횟수

기존의 모듈라 곱셈 알고리즘들과 본 논문에서 제안한 알고리즘들을 수행하는 데에 필요한 단정도 곱셈의 횟수를 계산하고, 이들을 비교한다.

기본적 방법 1절에서 언급했듯이, 기존의 모듈라 곱셈 알고리즘들의 종류는 다양하지만, 그들 모두는 필요한 단정도 곱셈의 횟수 측면에서 볼 때, 기본적 방법(basic method)과 별로 차이가 없다. 따라서, 본 절에서는 기본적 방법으로 모듈라 곱셈을 수행하는 데에 필요한 단정도 곱셈의 횟수를 계산한다.

식(7)과 같이 표현되는 큰 두 정수의 곱셈에 필요한 단정도 곱셈의 갯수는 k^2이다. 또한, 현재 알려진 모듈라 감소(mod ular reduction) 알고리즘들은 모두 $k(k+c)$번의 단정도 곱셈을 필요로 한다. 따라서, 모듈라 곱셈을 수행하는 데에 필요한 총 단정도 곱셈의 갯수는 다음과 같다.

$$2k^2+ck$$

여기서 c는 모듈라 감소 알고리즘에 따른 상수로서, 고전적(classical) 알고리즘[5]의 경우는 그 값이 '2' 이고, Barrett 알고리즘[13]의 경우는 그 값이 '4' 이며, Montgomery 알고리즘[14, 8]의 경우는 그 값이 '1' 이다.[6]

윈도우 모듈라 곱셈 알고리즘 본 절에서는 본 논문에서 제안한 윈도우 모듈라 곱셈 알고리즘의 성능을 살펴본다. 즉, 알고리즘 4.1과 알고리즘 4.2를 수행하는 데에 필요한 단정도 곱셈의 횟수를 계산한다.

식(8)을 살펴보면, 식(3)을 알고리즘 4.1을 이용해서 계산할 때 모듈라 감소 연산을 따로 수행할 필요가 없음을 알 수 있다. 단지 중간 단계의 모든 값을 더한 결과가 k자리를 넘어가게 되므로 이를 줄이기 위한 몇 번의 과외 연산만 추가로 필요할 뿐이다.

우선, 알고리즘 4.1에 사용되는 사전계산 테이블이 미리 계산되어 있다고 가정하고, 그 알고리즘을 수행하는 데에 필요한 단정도 곱셈의 갯수를 계산한다.

먼저, 각 자릿수(알고리즘 4.1의 6번째 줄의 c_i)와 해당 자릿수 단위의 모듈라 감소 결과(같은 줄의 $T[\alpha][i]$)를 곱하는 데에 k번의 단정도 곱셈이 필요하다. 그런데, 알고리즘 4.1에서 C의 자릿수가 k이므로 모두 $k\times k$번의 단정도 곱셈이 필요하다. 그런데, 각 중간결과들을 더한 값이 N보다 약간 커지게 된다. 그러나, 그 커지는 비율이 아주 작은 크기이므로 4.2절에서 설명한 모듈라 제곱 알고리즘에 사용되는 사전계산 테이블을 이용하면 단정도 곱셈을 사용하지 않고 최종 나머지 값(residue)을 구할 수 있다. 따라서, 식(8)을 계산하는 데에는 모두

$$k^2 \qquad (14)$$

번의 단정도 곱셈이 필요하다.

다음으로 알고리즘 4.1에 사용되는 사전계산 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 필요 없다. 그 다음의 연산들은 모두 한 번의 왼쪽으로의 이동(left-shift)과 모듈라 감소를 필요로 하게 된다. 그런데, 이 때의 모듈라 감소는 모두 N보다 한 자릿수가 큰 수에 대한 모듈라 감소이므로 k번의 단정도 곱셈으로 계산된다. 따라서, 테이블의 모든 값을 계산하는 데에 필요한 단정도 곱셈의 갯수는 다음과 같다. w는 작은윈도우 기법에서의 윈도우의 크기이다.

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

그러나, 이 때에 필요한 단정도 곱셈은 모두 N보다 한 자릿수가 큰 수에 대한 모듈라 감소에 필요한 것이므로, 4.2절에서 설명한 모듈라 제곱에 사용되는 사전계산 테이블을 이용하면 단정도 곱셈이 필요없게 된다.

모듈라 제곱 알고리즘 본 절에서는 본 논문에서 제안한 모듈라 제곱 알고리즘의 성능을 살펴본다. 즉, 알고리즘 4.3, 알고리즘 4.4, 그리고 알고리즘 4.5를 수행하는 데에 필요한 단정도 곱셈의 횟수를 계산한다.

식(12)를 살펴보면 본 논문에서 제안한 모듈라 제곱 알고리즘을 수행하는 데에 필요한 단정도 곱셈의 갯수를 계산할 수 있다. 우선, 알고리즘 4.4에 사용되는 사전계산 테이블이 이미 계산되어 있다고 가정하고, 알고리즘 4.3과 알고리즘 4.4를 계산하는 데에 필요한 단정도 곱셈의 횟수를 계산한다.

먼저 재귀함수호출(recursive function call)의 경계조건(boundary condition)에 도달 한 후에, 식(11)을 계산하는 데에 $\frac{k^2+2k}{8}$번의 단정도 곱셈이 필요하다. 그리고,
식(12)의 우변 중에서, 첫째 항은 식(13)에서 알 수 있듯이 곱셈을 필요로 하지 않는다. 그리고, 둘째 항은 알고리즘 4.3의 5번째 줄에 있는 $c_0 \times C' \times b$에 의해 계산된다. 그런데, C'은 한 번씩 재귀호출(recursive call)을 수행할 때마다 한 자릿수씩 감소되므로 j번 재귀호출 했을 때의 C'은 k-j-1 자릿수의 정수가 된다. 그리고, b를 곱하는 연산은 단순한 왼쪽으로의 이동(left-shift)이고 c_0는 단정도 정수이므로, j번째 단계에서 $c_0 \times C' \times b$를 계산하는 데에 필요한 단정도 곱셈의 횟수는 k-j-1 이다. 그런데, 재귀호출을 $k/2$번 반복하면 모듈라 제곱의 수행이 완료된다. 따라서, 모든 재귀호출(recursive call) 동안에 식(12)의 우변 중 둘째 항을 계산하는 데에 필요한 단정도 곱셈의 총 횟수는
$\frac{3k^2+2k}{8}\left(=\sum_{j=-\frac{k}{2}}^{k-1} j\right)$이다.

마지막으로 세째 항은 각 단계마다 한 번씩의 단정도 곱셈을 수행 하므로 총 $k/2$번의 단정도 곱셈을 필요로 하게 된다. 따라서, 식(4)를 본 논문에서 제안한 모듈라 제곱 알고리즘을 이용해서 계산하는 데에 필요한 단정도 곱셈의 총 갯수는 다음과 같다.

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

다음으로 알고리즘 4.4에서 이용되는 사전계산 테이블을 만드는 데에 필요한 단정도 곱셈의 횟수를 계산한다. 식(13)의 사전계산 테이블 T는 알고리즘 4.5를 수행 함으로써 만들어지는데, 이는 덧셈들 만으로 이루어져 있으므로 사전계산 테이블을 만드는 데에는 단정도 곱셈이 필요하지 않다.

비교평가 기존의 모듈라 곱셈 알고리즘들과 본 논문에서 제안한 모듈라 곱셈 방법을 수행하는 데에 필요한 단정도 곱셈의 횟수들을 비교한 결과가 표 2와 그림 4에 나타나 있다. 본 논문에서 제안한 알고리즘들에서 테이블을 만드는 데에 걸리는 시간과 Montgomery 알고리즘에서의 사전계산(pre-calculation)과 사후계산(post-calculation)에 걸리는 시간은 고려하지 않았다. 왜냐하면, 한 번의 모듈라 멱승 연산을 위해서는 많은 모듈라 곱셈이 필요하므로, 과

외연산들이 차지하는 비중은 극히 미미하기 때문이다. 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. 표 2의 s는 식(13)에 나타난 s이다.

표 2: 각 알고리즘들의 성능과 메모리 필요량 : 성능척도는 필요로 하는 단정도 곱셈의 횟수, 메모리 필요량의 단위는 피연산자의 갯수. k는 피연산자의 자릿수.

알 고 리 즘		단정도 곱셈	메모리
윈도우 모듈라 곱셈	Bar.[13]	$2k^2+4k$	2^{w-1}
	SOR.[15]	$2k^2+2k$	$2^{w-1}+k$
	Mont.[14]	$2k^2+k$	2^{w-1}
	제안	k^2	$k\times 2^{w-1}$
일반 제곱		k^2	2^{w-1}
윈도우 모듈라 곱셈	Bar.[13]	$\frac{3}{2}(k^2)+\frac{9}{2}k$	0
	SOR.[15]	$\frac{3}{2}(k^2)+\frac{5}{2}k$	k
	Mont.[14]	$\frac{3}{2}(k^2+k)$	0
	제안	$\frac{1}{2}(k^2+k)$	$k\times s$
일반 제곱		$\frac{1}{2}(k^2+k)$	0

표 2와 그림 4에서 알 수 있듯이 본 논문에서 제안한 모듈라 곱셈 알고리즘들에서 필요한 단정도 곱셈의 갯수는 일반 다정도 곱셈과 일반 다정도 제곱에 필요한 그것과 동일하다. 즉, 모듈라 감소 연산을 위한 별도의 단정도 곱셈을 필요로 하지 않는다. 윈도우 모듈라 곱셈 알고리즘은 사전계산을 통해서, 그리고 모듈라 제곱 알고리즘은 사전계산과 뺄셈을 통해서 모듈라 감소 연산에 필요한 단정도 곱셈을 없앴다.

5.2.2 구현 및 논의

본 논문에서 제안하는 모듈라 곱셈 알고리즘들을 사용하여 모듈라 곱셈 연산을 수행하는 프로그램을 실제로 구현하여, 그 연산시간을 측정한 결과가 표 3에 나타나 있다. 비교대상은 기존의 알고리즘들 중 가장 좋은 성능을 보이는 Montgomery 알고리즘이다. 구현에 사용된 시스템은 Pentium-90 마이크로프로세서를 사용하는 PC이며, C 언어로 구현해서 Watcom C(version 10.0) 컴파일러를 이용해서 실행화일을 얻어 수행한 결과이다. 16-bit를 한 자릿수로 삼았다($b=2^{16}$).

표 2에 의하면, 본 논문에서 제안된 알고리즘들은 기존의 Montgomery 알고리즘에 비해 각각 약 2배와 3배 빠르다. 표 3과 그림 5(a)

에 나타나 있듯이 윈도우 모듈라 곱셈 알고리즘의 경우에는 실제 연산시간이 표 2와 그림 4(a)에 일치한다. 그러나, 모듈라 제곱 알고리즘의 경우에'는 그렇지 못하다. 그 이유는 표 2가 모듈라 곱셈에 필요한 단정도 곱셈의 횟수만을 계산한 것이기 때문이다.

일반 범용 프로세서에서 곱셈이 덧셈에 비해 많은 시간이 걸리는 것은 사실이지만, 실제 시스템에서의 연산시간을 예측하기 위해서는 덧셈횟수도 고려해야 한다. 알고리즘을 수행하는 데에 필요한 단정도 덧셈의 횟수를 단정도 곱셈의 횟수로 환산해서, 각 알고리즘의 성능을 나타낸 결과가 표 4에 나타나 있다. 표 4에서 r은 단정도 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)

그림 6은 표 4에 대한 그래프로서, r에 따른 각 알고리즘들의 Montgomery 알고리즘에 대한 성능비를 나타낸 것이다. 성능비는 다음과 같이 정의 된다.

정의 5.2 알고리즘 (가)의 알고리즘 (나)에 대한 성능비=(알고리즘 (나)를 수행하는 데에 필요한 단정도 곱셈의 횟수)/(알고리즘 (가)를 수행하는 데에 필요한 단정도 곱셈의 횟수)

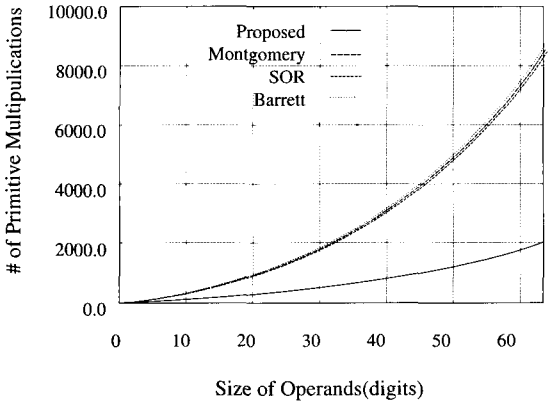
그림 6(a)에 나타난 바에 의하면, 윈도우 모듈라 곱셈 알고리즘의 경우에는 r이 달라지더라도 성능비는 거의 변하지 않는다. 이는 어떤 시스템에서도 본 논문에서 제안한 윈도우 모듈라 곱셈 알고리즘이 기존의 Montgomery 알고리즘에 비해 2배 정도 빠름을 의미한다. 반면

표3: 각 알고리즘들의 실제 수행시간. WMM(Window mod ular Multiplication)은 윈도우 모듈라 곱셈을 의미하고, MS(modular Squaring)은 모듈라 제곱을 의미한다. 수행시간 열(column)의 subcolumn들은 피연산자의 크기를 의미하며, 단위는 비트(bit)수이다.

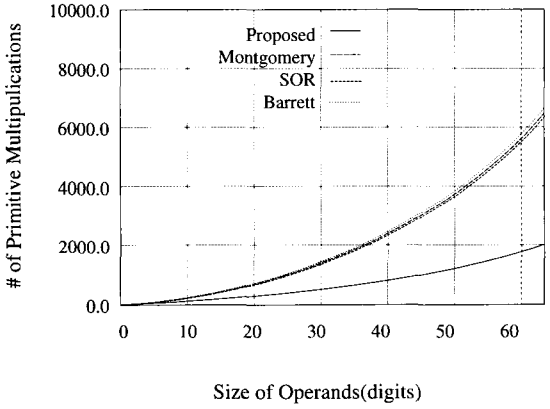
알 고 리 즘		수행시간(msec.)			
		256	512	768	1024
WMM	Mont.[14]	0.14	0.54	1.2	2.1
	제안	0.060	0.22	0.49	0.87
MS	Mont.[14]	0.12	0.45	0.92	1.7
	제안	0.085	0.30	0.70	1.2

표4: 단정도 덧셈 횟수까지 포함해서 계산된 성능 비교 :
성능척도는 필요로 하는 단정도 곱셈의 횟수, k는 피연산자의 자릿수.

알 고 리 즘		단정도 곱셈 횟수
윈도우 모듈라 곱셈	Mont.[14]	$2k^2+k+r(2k^2)$
	제안	$k^2+r\times(k^2+3(k+1))$
모듈라 제곱	Mont.[14]	$\frac{3}{2}(k^2+k)+r\times(\frac{3}{2}k^2+k)$
	제안	$\frac{1}{2}(k^2+k)+r\times\frac{5}{2}(k^2+k)$

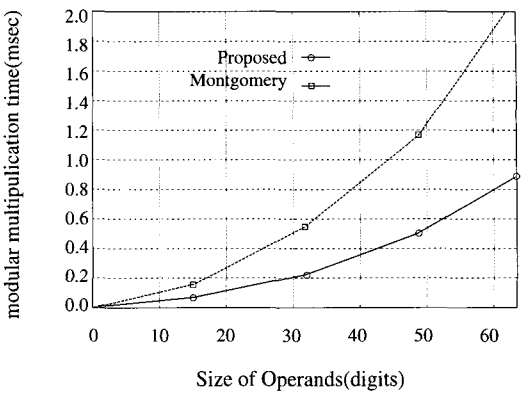


(a) 윈도우 모듈라 곱셈

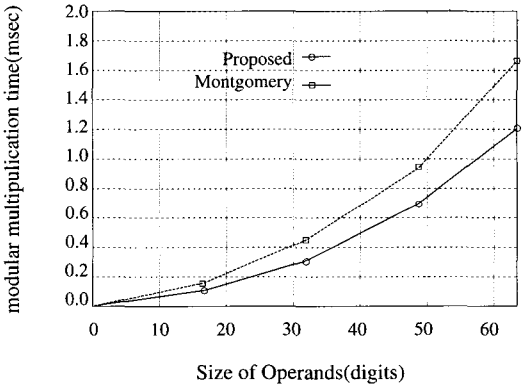


(b) 모듈라 제곱

그림 4: 각 알고리즘들을 수행하는 데에 필요한 단정도 곱셈의 횟수



(a) 윈도우 모듈라 곱셈



(b) 모듈라 제곱

그림 5: 각 알고리즘들의 실제 수행시간

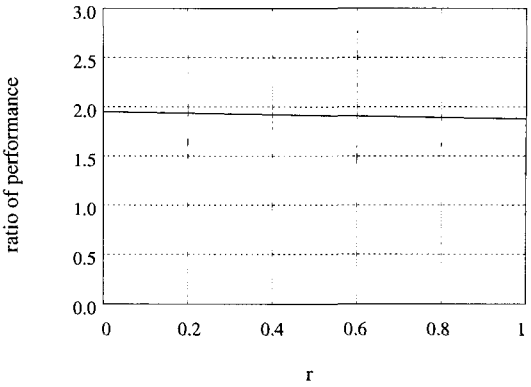
에 그림 6(b)를 보게 되면, r에 따라서 성능비가 많이 달라진다. 즉, 알고리즘이 구현되는 시스템에 따라서, 모듈라 제곱 알고리즘의 성능이 달라지게 된다.

컴퓨터 시스템 내에서 한 명령어를 수행할 때에, 기억장소의 양과 파이프라인(pipeline)여부 등 여러 가지 변수에 의해서 그 수행시간이 결정되므로, 정확한 r값을 알기는 어렵다. 하지만 각 명령어를 수행하기 위한 클럭(clock)수를 알 수 있으므로 대략적인 r값을 알 수 있다. 따라서, 본 논문에서 제안한 알고리즘을 이용할 때 해당 시스템에서 얻을 수 있는 성능향상을 예측할 수 있다. Pentium PC의 경우, r이 약 0.3~0.4로서 성능비는 약 1.5가량 된다. 이는 본 논문에서 제안한 모듈라 제곱 알고리즘을 사용하면, 기존의 Montgomery 알고리즘을 사용할 경우에 비해 수행시간이 약 30%정도 줄어드는 것을 의미한다. 이는 표 3에 나타나 있는 측정결과와 일치한다.

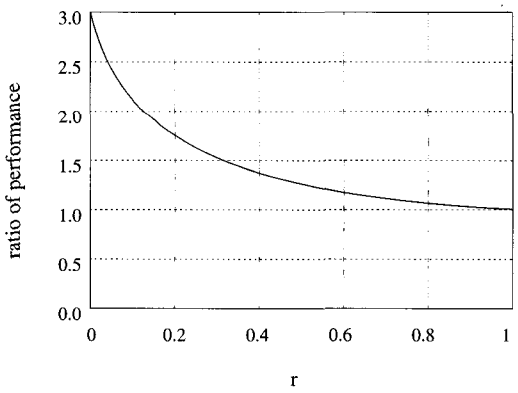
6. 결론

본 논문에서는 모듈라 멱승 연산을 빠르게 수행할 수 있도록 하기 위한 몇가지 알고리즘을 제안하였다. 한 가지는 모듈라 곱셈의 반복 횟수를 줄이기 위한 덧셈사슬 휴리스틱이고, 다른 두 가지는 모듈라 곱셈을 빠르게 수행할 수 있도록 하기 위한 모듈라 곱셈 알고리즘들이다. 두 가지 모듈라 곱셈 알고리즘들 중 한 가지는 서로 다른 두 수의 모듈라 곱셈을 빠르게 수행하는 알고리즘이고, 다른 한 가지는 모듈라 제곱을 빠르게 수행할 수 있는 알고리즘이다.

본 논문에서는 본 논문에서 제안한 알고리즘들의 성능을 분석하였다. 덧셈사슬 휴리스틱의 성능척도는 그 알고리즘을 사용해서 구해지는 덧셈사슬의 길이로서, 본 논문에서 제안한 휴리스틱을 작은윈도우 기법에 적용하면 임의의 512비트 정수에 대해 평균길이 602의 덧셈사슬을 구할 수 있다. 이는 기존의 다른 어떤 알고리즘으로 구할 수 있는 덧셈사슬의 길이보다 짧은 것이다. 모듈라 곱셈 알고리즘의 성능척도는 그 알고리즘을 수행할 때 필요한 단정도 곱셈의 횟수로서, 본 논문에서 제안한 윈도우 모듈라 곱셈 알고리즘과 모듈라 제곱 알고리즘은 기존의 알고리즘들 중 가장 좋은 성능을 보이는 Montgomery 알고리즘보다 각각 1/2과 1/3만큼의 단정도 곱셈만을 필요로 한다. 실제로 PC에서 구현한 결과, 본 논문에서 제안한 알고리즘들이 약 30-50%의 성능 향상을 보였다.



(a) 윈도우 모듈라 곱셈



(b) 모듈라 제곱

그림 6.: r에 따른 성능비의 변화

참 고 문 헌

[1] R.L.Rivest, A.Shamir, and L.Adleman, "A method for obtaining digital signature and public key crytosystems," CACM, vol. 21, pp. 120~126, 1978.

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

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

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

[5] D.E.Knuth, The art of computer programming. Addition-Wesley, Inc., 1981.

[6] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three mod ular redution funtions," in Crypto'93, 1994.

[7] S. Kawamura, K. Takabayashi, and A. Shimbo, "A fast mod ular exponentiation algorithm," IE-ICE Transactions., vol. E-74, pp. 2136~2142, August 1991.

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

[9] J. Jedwab and C. J. Mitchell, "Minimum weight mod ified signed-digit representations and fast exponentiation," Electronics Letters, vol. 25, pp. 1171~1172, 1989.

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

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

[12] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," SIAM J. Comput., vol. 10, pp.

638~646, August 1981.

[13] 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.

[14] P. L. Montgomery, "modular multiplication without trial division," Mathematics of Computation, vol. 44, 1985.

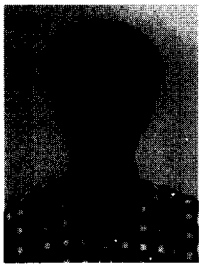
[15] P. Findlay and B. Johnson, " mod ular exponentiation using recursive sums of residues," in Crypto'89, 1990.

[16] H. Morita and C. Yang, "A mod ular multiplication," IEICE Trans. Fundamentals, vol. E76-A, pp. 70~77, Januur 1993.

□ 著者紹介

홍 성 민

1994년 2월 한국과학기술원 학사과정 졸업
1996년 2월 한국과학기술원 석사과정 졸업
1996년 3월 ~ 현재 한국과학기술원 박사과정 재학중



오 상 엽

1992년 2월 한국과학기술원 학사과정 졸업
1994년 2월 한국과학기술원 석사과정 졸업
1994년 3월 ~ 현재 현재 한국과학기술원 박사과정 재학중

윤 현 수

1979년 서울대학교 전자공학과 졸업
1981년 한국과학기술원 전산학과 석사학위 취득
1981년 ~ 1984년 삼성전자 연구원
1988년 오하이오 주립대학 전산학 박사학위 취득
1988년 ~ 1989년 AT&T Bell Labs. 연구원
1989년 ~ 현재 한국과학기술원 전산학과 부교수