The Journal of The Institute of Internet, Broadcasting and Communication (IIBC)
Vol. 23, No. 5, pp.15-20, Oct. 31, 2023. pISSN 2289-0238, eISSN 2289-0246

https://doi.org/10.7236/JIIBC.2023.23.5.15
JIIBC 2023-5-3

배열을 이용한 이산대수의 사이클 검출

Cycle Detection of Discrete Logarithm using an Array

이상운*

Sang-Un Lee*

요 약 지금까지는 Pollard의 Rho 알고리즘이 대칭키의 암호를 해독하는 이산대수 문제에 대해 가장 효율적인 방법으로 알려져 있다. 그러나 이 알고리즘은 거인걸음 보폭 $m=\lceil \sqrt{p} \rceil$ 개의 데이터를 저장해야 하는 단점과 더불어 $O(\sqrt{p})$ 수행 복잡도를 보다 감소시킬 수 있는 방법에 대한 연구가 진행되고 있다. 본 논문은 이산대수의 사이클 검출을 위한 Nivasch의 스택 법의 데이터 갱신 횟수를 73% 이상 감소시키는 배열법을 제안하였다. 제안된 방법은 배열을 적용하였으며, $(x_i < 0.5x_{i-1}) \cap (x_i < 0.5(p-1))$인 경우에 한해 배열 값을 갱신하는 방법을 적용하였다. 제안된 방법은 스택법과 동일한 모듈러 연산횟수를 보였지만 스택 법에 비해 이진탐색 법을 적용하여 배열 갱신 횟수와 탐색 시간을 획기적으로 감소시켰다.

Abstract Until now, Pollard's Rho algorithm has been known as the most efficient way for discrete algebraic problems to decrypt symmetric keys. However, the algorithm is being studied on how to further reduce the complexity of $O(\sqrt{p})$ performance, along with the disadvantage of having to store the giant stride $m=\lceil \sqrt{p} \rceil$ data. This paper proposes an array method for cycle detection in discrete logarithms. The proposed method reduces the number of updates of stack memory by at least 73%. This is done by only updating the array when $(x_i < 0.5x_{i-1}) \cap (x_i < 0.5(p-1))$. The proposed array method undergoes the same number of modular calculation as stack method, but significantly reduces the number of updates and the execution time for array through the use of a binary search method.

Key Words : Discrete logarithm, Cycle detection, Pointer method, Stack method, Array method

Ⅰ. 서 론

대칭암호 비밀 키는 소수(prime number, p)를 사용하며, $\alpha^{\gamma} \equiv \beta \pmod{p}$에서 α, β, p가 주어졌을 때 비밀 키 γ을 구하는 이산대수(discrete logarithm)의 수학적 난제에 기반하고 있다.[1,2]

이산대수 알고리즘에는 아기걸음-거인걸음(Baby-step Giant-step), Pollard의 캥거루와 Rho (ρ), Pohlig-Hellman, Index calcurus, Number Field Sieves, Function Field Sieve 등 다양하게 존재하고 있다.[2] 지금까지는 Pollard의 Rho 알고리즘[3]이 대칭키의 암호를 해독하는 이산대수 문제에 대해 가장

*정회원, 강릉원주대학교 과학기술대학 멀티미디어공학과
접수일자 2023년 2월 1일, 수정완료 2023년 9월 3일
게재확정일자 2023년 10월 6일

Received: 1 February, 2023 / Revised: 3 September, 2023 / Accepted: 6 October, 2023
*Corresponding Author: sulee@gwnu.ac.kr
Dept. of Multimedia Eng., Gangneung-Wonju National University, Korea

효율적인 방법으로 알려져 있어 이 방법에 대한 후속 연구가 진행되고 있다.[4-8]

Shank의 아기걸음-거인걸음 알고리즘[9]은 거인걸음 보폭 $m = \lceil \sqrt{p} \rceil$ 개의 데이터를 저장해야 하는 단점을 갖고 있으며, 수행 복잡도는 $O(\sqrt{p})$이다.[10] 반면에, Pollard의 Rho 알고리즘[3]은 단지 2개의 포인터만 사용하여 충돌을 검출하는 방법으로 메모리 문제를 해결하였으나 수행 복잡도는 $O(\sqrt{p})$이다. Pollard의 Rho(ρ) 알고리즘에 대한 후속 연구가 Brent[4,5], Teske[6], Nivasch[7]와 Cheon et al.[8]에 의해 수행되었다. Teske[6]는 일반화된 Rho 알고리즘을 제안하였으며, Brent[4,5]와 Nivasch[7]는 Pollard의 Rho 알고리즘의 효율적 수행 방법을 제안하였다. 또한 Lee[11]은 다중 ρ 알고리즘을 제안하기도 하였다. 사이클 검출에 있어 Lee[12]는 큐(queue)를 이용하여 Pollard ρ 알고리즘의 수행횟수를 65.02%, Brent 알고리즘의 수행횟수를 47.80% 감소시키는 결과를 얻었다.

본 논문은 Lee[12]의 큐 이용 방법이나 Nivasch의 스택 이용 알고리즘[7]에 비해 보다 알고리즘 수행횟수를 감소시켜 사이클을 검출할 수 있는 알고리즘을 제안한다. 2장에서는 사이클 검출 방법을 고찰한다. 3장에서는 배열을 이용한 사이클 검출 방법을 제안한다. 4장에서는 실험을 통해 제안된 알고리즘의 사이클 검출 성능을 검증하여 본다.

II. 이산대수의 사이클 검출

이산대수 문제를 풀기 위해 사이클 검출 기법을 적용한 방법에는 Pollard의 ρ 포인터 이용법[3], Brent의 포인터 이용법[4,5], Nivasch의 스택 이용법[7] 등이 있다.

Pollard의 ρ 방법은 사이클 검출에 있어서 가장 효율적인 방법으로 알려진 Floyd 알고리즘에 기반하고 있다.[13,14] 또한, 사이클을 검출하는 방법이 그리스 문자 Rho(ρ)와 유사하여 Rho 알고리즘이라 명명하였다. ρ 알고리즘[4]은 $\alpha^{\gamma} \equiv \beta \pmod p$에서 γ을 구하기 위해 식 (1)을 찾으며, $\alpha^a \beta^b \equiv \alpha^A \beta^B$의 충돌을 찾기 위해 $x_0 = y_0 = 1$, 즉 $a = b = A = B = 0$에서 거북이와 토끼가 동시에 출발하여 거북이는 정상 속도인 $x_i \leftarrow f(x_{i-1})$로, 토끼는 거북이의 2배 속도인 $y_i \leftarrow f(f(y_{i-1})) = x_{2i}$로 달려가면서 $x_i = x_{2i}$이면 충돌이 발생하여 종료하는 방식으로 수행된다. 여기서 x_i와 y_i는 식 (2)로 계산되며, $y_i = x_{2i}$ 값이다.

$$\alpha^a \beta^b \equiv \alpha^A \beta^B, \ (B-b)\gamma = (a-A) \tag{1}$$

$$\alpha^a (\alpha^{\gamma})^b \equiv \alpha^A (\alpha^{\gamma}) \beta^B, \alpha^{a+\gamma b} \equiv \alpha^{A+\gamma B},$$
$$(a+\gamma b) = (A+\gamma B), \gamma(B-b) = (a-A)$$

$$f(x) = \begin{cases} x_{i-1}^2 \pmod p, & a_i = 2a_{i-1} \pmod G, b_i = 2b_{i-1} \pmod G \\ \alpha x_{i-1} \pmod p, & a_i = a_{i-1} + 1 \pmod G, b_i = b_{i-1} \\ \beta x_{i-1} \pmod p, & a_i = a_{i-1}, b_i = b_{i-1} + 1 \pmod G \end{cases}$$

$$f(y) = \begin{cases} y_{i-1}^2 \pmod p, & A_i = 2A_{i-1} \pmod G, B_i = 2B_{i-1} \pmod G \\ \alpha y_{i-1} \pmod p, & A_i = A_{i-1} + 1 \pmod G, B_i = B_{i-1} \\ \beta y_{i-1} \pmod p, & A_i = A_{i-1}, B_i = B_{i-1} + 1 \pmod G \end{cases} \tag{2}$$

ρ 알고리즘의 일반형은 식 (3)과 같다.[6] 여기서 $M = \alpha^m, N = \beta^n$으로 m, n을 임의로 설정한다.

$$f(x) = \begin{cases} x_{i-1}^2 \pmod p, & a_i = 2a_{i-1} \pmod G, b_i = 2b_{i-1} \pmod G \\ Mx_{i-1} \pmod p, & a_i = a_{i-1} + m \pmod G, b_i = b_{i-1} \\ Nx_{i-1} \pmod p, & a_i = a_{i-1}, b_i = b_{i-1} + n \pmod G \end{cases} \tag{3}$$

Brent 알고리즘[4,5]은 ρ 알고리즘에서 $y \leftarrow f(f(y))$를 적용하지 않고, 단지 $x \leftarrow f(x)$만 활용한다. 이 방법은 $y \leftarrow x_i, (i = 2^k)$로 수행횟수 i가 2의 배수일 때 x_i의 값을 y에 저장하여 비교하는 방식이다. 이로 인해 ρ 알고리즘은 1회 수행에 $x \leftarrow f(x)$와 $y \leftarrow f(f(y))$의 3회 모듈러 연산을 수행하는 것을 1회로 감소시켜 Floyd의 사이클 검출 알고리즘에 비해 약 36% 빠르며, ρ 알고리즘의 수행시간을 24% 향상시켰다.[14]

Pollard와 Brent는 사이클 검출을 위해 2개 포인터(pointer)만을 활용하였다. Brent 알고리즘[4,5]의 경우 y를 1개 포인터만 적용함에 따라 이전 y값과 동일하여 수행횟수를 줄일 수 있음에도 불구하고 수행횟수가 많아지는 경우가 발생할 수도 있다.

Brent 알고리즘[4,5]의 문제점을 개선한 방법이 Nivasch[7]의 스택(stack)을 활용하는 방법이다. 이 방법은 Brent 알고리즘과 같이 $x \leftarrow f(x)$만을 적용하며, i번째 수행횟수에서의 x_i 값이 스택의 최대치로 Top에 위치하도록 한다. 즉, 스택에는 오름차순으로 값이 저장된다. 이 방법은 Brent 알고리즘에 비해 20% 빠른 것으로 증명되었다.[7] 그러나 스택의 크기 뿐 아니라 값을 갱신하는데 많은 시간이 소요될 수 있다. Nivasch[7]는 $p = 5 \times 10^8$의 경우 스택 크기가 [44,53]임을 제시하였다.

크기 k, Top(T)와 Bottom(B)를 갖고 있는 스택을 가정하여 보자. 만약, $x_i > T$이면 포인터를 $k \leftarrow k+1$로 증가시켜 x_i 값을 push하면 되지만 $x_i < T$이면 역 탐색(backward search)을 수행하여 $x_i < s_j$인 j 위치까지 pop을 수행하여 j 위치에 x_i 값을 push해야 한다. 만약, 최악의 경우인 $x_i < B$라면 $O(k)$의 수행시간이 필요하다. 이 방법은 비록 데이터가 정렬되어 있음에도 불구

하고 반드시 역 탐색 방법을 수행하야 한다. 스택의 역탐색법은 선형 탐색(linear search) 또는 Brute-force 탐색 법만을 적용할 수 있다.

Ⅲ. 배열 이용 알고리즘

본 장에서 제안하는 방법은 배열(array)을 이용해 다음과 같이 수행한다.

Step 1. 식 (4)의 $x \leftarrow f(x)$를 계산한다.

$$\alpha^{\gamma} \equiv \beta (\mathrm{mod}\, p),\ x_0 = \alpha\beta\ \ (a=b=1).$$

$$x_{i-1} (\mathrm{mod}\, 3) = \begin{cases} 0,\ x_i = x_{i-1}^2 (\mathrm{mod}\, p) \\ 1,\ x_i = \alpha x_{i-1} (\mathrm{mod}\, p) \\ 2,\ x_i = \beta x_{i-1} (\mathrm{mod}\, p) \end{cases} \quad (4)$$

$f(x)$는 식 (4)의 random walks 형태인 Pollard의 기본형 ρ 함수 이외에 식 (3)의 일반형 ρ 함수를 적용할 수도 있으나 여기서는 식 (4)의 기본형 ρ 함수를 적용한다.

Step 2. y는 배열 A를 적용한다.

배열 A의 $A(1)$은 최소치, k번째 $A(k)$는 최대치가 저장되며, 포인터는 k를 가리킨다. 초기치 $x_0 = \alpha\beta (\mathrm{mod}\, p)$가 $A(1)$에 저장된다.

Step 3. $(x_i < 0.5x_{i-1}) \cap (x_i < 0.5(p-1))$이 한정하여 배열 A값을 갱신한다.

```
if   x_i < 0.5x_{i-1} and x_i < 0.5(p-1) then
     /* 배열 A 갱신
     if        x_i > A(k) then
               k←k+1, A(k)←x_i, a_max←x_i
     else if   x_i < A(1) then
               초기화, k←1, A(1)←x_i, a_min←x_i
     else if   x_i = A(1) or x_i = A(k) then
               알고리즘 종료
     else if   A(1) < x_i < A(k) then /* 이진탐색
               if       x_i = A(j) then 알고리즘 종료
               else if  A(j-1) < x_i < A(j) then
                        A(j)←x_i, a_max←x_i, k←j.
else skip.
```

스택 방법은 모든 x_i에 대해 스택 값을 갱신하는데 반해 배열 방법은 $x_i < 0.5x_{i-1}$와 $x_i < 0.5(p-1)$ 조건을 만족하는 경우에 한정시키는 차이점이 있다. 또한, 배열 갱신 방법에 있어서는 $x_i > A(k)$이면 스택과 동일한 방법으로 $k \leftarrow k+1$에 단순히 추가된다. 반면에 $x_i < A(1)$이면 배열을 초기화시키고 $A(1) \leftarrow x_i$로 추가시키는 단순한 과정을 수행한다. 마지막으로 $A(1) < x_i < A(k)$이면 이진탐색으로 x_i와 동일한 값이 존재하는지 검증하며, 만약 동일한 값이 없으면 x_i가 배열의 최대치가 되도록 k의 위치를 감소시킨다.

Step 3을 수행함으로서 배열 갱신 횟수를 감소시킬 뿐 아니라 이진탐색을 적용하여 수행시간 복잡도 $O(\log k)$로 배열 탐색 시간도 획기적으로 줄일 수 있다.

만약, x_i가 그림 1과 같다고 가정하여 보자. 본 데이터는 Shamir[13]에서 인용되었다. 여기서 최대치는 9로 10을 법으로 하는 경우라고 가정하면 $x_i < 0.5(p-1) = 4$로 배열의 갱신 최대치는 4이다. 이 데이터는 사이클 시작점을 μ, 사이클 길이를 λ라 할 때 $\mu = 2, \lambda = 8$이며, 사이클에서의 최소치는 1이다. 따라서 스택을 이용하면 사이클 최소치 $x_5 = 1 = x_{13}$을 찾는다.

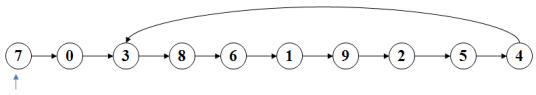

그림 1. $x_i = f(x)$
Fig. 1. $x_i = f(x)$

표 1. 스택과 배열 갱신횟수
Table 1. The number of updates for Stack and Array

수행 횟수	x_i	스택			배열		
		변경전	변경후	수행방법	변경 전	변경후	수행 방법
초기치	7		7	-		7	-
1	0	7	0	탐색, 갱신	7	0	초기화, 추가
2	**3**	0	0,3	추가			-
3	8	0,3	0,3,8	추가			-
4	6	0,3,8	0,3,6	탐색, 갱신			-
5	1	0,3,6	0,1	탐색, 갱신	0,3	0,1	탐색, 갱신
6	9	0,1	0,1,9	추가			-
7	2	0,1,9	0,1,2	탐색, 갱신	0,1	0,1,2	추가
8	5	0,1,2	0,1,2,5	추가			-
9	4	0,1,2,5	0,1,2,4	탐색, 갱신			-
10	**3**	0,1,2,4	0,1,2,3	탐색, 갱신			-
11	8	0,1,2,3	0,1,2,3,8	추가			-
12	6	0,1,2,3,8	0,1,2,3,6	탐색, 갱신			-
13	**1**	0,1,2,3,6	0,**1**	탐색	0,**1**,2	0,**1**	탐색

그림 1의 데이터에 대해 스택을 적용하는 방법과 배열을 적용하는 방법을 수행하는 과정은 표 1과 같다. 스택을 이용하는 방법은 탐색 및 갱신(search & update) 8회와 추가(add) 5회의 13회를 수행한다. 반면에 배열을 이용하는 방법은 $x_i < 0.5x_{i-1}$와 $x_i < 0.5(p-1)$을 적용하면 단순 추가 2회, 탐색과 갱신 2회의 4회로 감소시킨다.

Ⅳ. 실험 및 결과 분석

$n=1019$에 대해 $\alpha^\gamma \equiv \beta(\mathrm{mod}\, n)$에서 $\alpha=2,5$, $\gamma=10, 100, 200, \cdots, 1000$에 대해 Pollard의 Rho, Brent, Nivasch의 스택 이용법과 제안된 배열 이용법에 대한 모듈러 연산 횟수와 메모리 충돌 검증 횟수 (갱신 횟수 포함)를 비교하였다.

표 2. $2^{100} \equiv 548(\mathrm{mod}\, 1019)$**의 사이클 검출**

Table 2. Cycle Detection of $2^{100} \equiv 548(\mathrm{mod}\, 1019)$

$2^{100} \equiv 548(\mathrm{mod}\, 1019)$							
수행횟수 (i)	x_i	사이클 검출					
		Pollard의 Pointer 방법 $y=x_{2i}$	Brent의 Pointer 방법 $y=x_j\ (i=2^k)$	Nivasch의 스택 방법 $y=Stack$		제안된 배열 방법 $y=Array$	
초기치	77	77	77	77		77	
1	**417**	659	77	77,417	추가		
2	659	812	77 → 659	77,417,659	추가		
3	406	148	**659**	77,**406**	탐색, 갱신		
4	812	187	659 → 812	77,406,812	추가		
5	682	133	**812**	77,406,**682**	탐색, 갱신		
6	148	51	812	77,**148**	탐색, 갱신	77,148	추가
7	296	786	812	77,148,296	추가		
8	187	42	812 → **187**	77,148,**187**	탐색, 갱신		
9	374	471	**187**	77,148,187,374	추가		
10	133	417	187	77,**133**	탐색, 갱신	77,**133**	탐색, 갱신
11	266	406	187	77,133,266	추가		
12	51	682	187	**51**	탐색, 갱신	51	초기화, 추가
13	563	296	187	51,563	추가		
14	786	374	187	51,563,786	추가		
15	282	266	187	51,**282**	탐색, 갱신	51,282	추가
16	42	563	187 → 42	**42**	탐색, 갱신	42	초기화, 추가
17	745	282	**42**	42,745	추가		
18	**471**	745	42	42,**471**	탐색, 갱신		
19	**718**	**718=718**	42	42,471,718	추가		
20	417		42	42,**417**	탐색, 갱신		
21	659		42	42,417,659	추가		
22	406		42	42,**406**	탐색, 갱신		
23	812		42	42,406,812	추가		
24	682		42	42,406,**682**	탐색, 갱신		
25	148		42	42,**148**	탐색, 갱신	42,148	추가
26	296		42	42,148,296	추가		
27	187		42	42,148,**187**	탐색, 갱신		
28	374		42	42,148,187,374	추가		
29	133		42	42,**133**	탐색, 갱신	42,**133**	탐색, 갱신
30	266		42	42,133,266	추가		
31	51		42	42,**51**	탐색, 갱신	42,**51**	탐색, 갱신
32	563		42 → 563	42,51,563	추가		
33	786		**563**	42,51,563,786	추가		
34	282		563	42,51,**282**	탐색, 갱신	42,51,282	추가
35	**42**		563	42-**42**	탐색	42-**42**	탐색
36	745		563				
37	471		563				
38	**718**		563				
39	417		563				
40	659		563				
41	406		563				
42	812		563				
43	682		563				
44	148		563				
45	296		563				
46	187		563				
47	374		563				
48	133		563				
49	266		563				
50	51		563				
51	**563**		**563**				
메모리 최대 크기				4		3	
메모리 충돌 검증횟수				35		10	

$2^{100} \equiv 548(\mathrm{mod}\, 1019)$에 대해 각 알고리즘별 수행방법을 비교한 결과는 표 2에 제시되어 있으며, $\alpha=2,5$, $\gamma=10, 100, 200, \cdots, 1000$에 대해 Pollard의 Rho, Brent, Nivasch의 스택 이용법과 제안된 배열 이용법의 경우는 표 3에 제시되어 있다.

표 3. $\alpha^\gamma \equiv \beta(\mathrm{mod}\, 1019)$**의 모듈러 연산과 메모리 충돌 검증 횟수**

Table 3. The number of Modular computation and collision verification for $\alpha^\gamma \equiv \beta(\mathrm{mod}\, 1019)$

α	모듈러 연산 횟수 (메모리 탐색 횟수)								
	$\alpha^\gamma \equiv \beta_\gamma$	μ	λ	사이클 최소값	Pollard의 2 Pointer 법 $x_0=y_0=\alpha\beta$ $y=x_{2i}$	Brent의 2 Pointer 법 $x_0=\alpha\beta$ $y=x_j\ (i=2^k)$	Nivasch의 스택 이용법 $x_0=\alpha\beta$ $y=Stack$	제안된 배열 이용법 $x_0=\alpha\beta$ $y=Array$	스택 이용법 대비 충돌 검증 비율
2	$2^{10} \equiv 5$	33	17	$x_{43}=86=x_{60}$	102 (34)	81 (81)	60 (60)	60 (15)	25.00 %
	$2^{100} \equiv 548$	1	19	$x_{16}=42=x_{35}$	57 (19)	51 (51)	35 (35)	35 (10)	28.57 %
	$2^{200} \equiv 718$	7	60	$x_{13}=37=x_{73}$	180 (60)	124 (124)	73 (73)	73 (15)	20.55 %
	$2^{300} \equiv 130$	7	20	$x_{15}=9=x_{35}$	60 (20)	52 (52)	35 (35)	35 (10)	28.57 %
	$2^{400} \equiv 929$	11	79	$x_{37}=3=x_{166}$	237 (79)	186 (186)	166 (166)	166 (43)	25.90 %
	$2^{500} \equiv 611$	34	12	$x_{40}=313=x_{57}$	108 (36)	76 (76)	57 (57)	57 (12)	21.05 %
	$2^{600} \equiv 596$	2	33	$x_8=44=x_{42}$	90 (33)	97 (97)	42 (42)	42 (10)	23.81 %
	$2^{700} \equiv 528$	28	46	$x_{46}=20=x_{82}$	138 (46)	110 (110)	92 (92)	92 (21)	22.83 %
	$2^{800} \equiv 967$	65	54	$x_{161}=4=x_{161}$	324 (108)	307 (307)	161 (161)	161 (39)	24.22 %
	$2^{900} \equiv 36$	12	10	$x_{10}=297=x_{35}$	60 (20)	26 (26)	**28** (28)	**25** (4) $x_{15}=333=x_{35}$	14.29 %
	$2^{1000} \equiv 367$	41	35	$x_{68}=35=x_{104}$	210 (70)	99 (99)	104 (104)	104 (23)	22.12 %
5	$5^{10} \equiv 548$	23	60	$x_{33}=4=x_{33}$	180 (60)	124 (124)	88 (88)	88 (20)	22.73 %
	$5^{100} \equiv 367$	15	15	$x_{31}=42=x_{31}$	45 (15)	31 (31)	41 (41)	41 (16)	39.02 %
	$5^{200} \equiv 181$	22	7	$x_{28}=23=x_{33}$	84 (28)	39 (39)	33 (33)	33 (12)	36.36 %
	$5^{300} \equiv 192$	10	19	$x_{17}=33=x_{36}$	57 (19)	51 (51)	36 (36)	36 (12)	33.33 %
	$5^{400} \equiv 153$	15	2	$x_{15}=204=x_{17}$	48 (16)	18 (18)	**17** (17)	**19** (5) $x_{17}=204=x_{34}$	29.41 %
	$5^{500} \equiv 106$	1	20	$x_{18}=5=x_{38}$	60 (20)	52 (52)	39 (39)	39 (12)	30.77 %
	$5^{600} \equiv 1017$	66	13	$x_{22}=48=x_{83}$	234 (78)	266 (266)	95 (95)	95 (21)	22.11 %
	$5^{700} \equiv 844$	4	27	$x_{16}=19=x_{43}$	81 (27)	59 (59)	43 (43)	43 (12)	27.91 %
	$5^{800} \equiv 991$	5	38	$x_{21}=9=x_{59}$	114 (38)	102 (102)	59 (59)	59 (13)	22.03 %
	$5^{900} \equiv 933$	7	25	$x_7=80=x_{32}$	75 (25)	57 (57)	32 (32)	32 (8)	25.00 %
	$5^{1000} \equiv 27$	12	14	$x_{17}=150=x_{31}$	42 (14)	30 (30)	31 (31)	31 (11)	35.48 %
	평균								**26.41 %**

$2^{100} \equiv 548(\mathrm{mod}\, 1019)$의 경우 $\mu=1, (x_1=417), \lambda=19$이다. 따라서 $x_1=417=x_{20}$가 사이클을 형성하고 있다. Nivasch의 스택 이용법이나 제안된 배열 이용법 모두 대부분은 사이클에서의 최소치 $x_1 \le x_{\min} \le x_{19}$을 $\mu+\lambda < x_{\min} \le \mu+2\lambda$에서 찾는 방법이다. 즉, $x_{\min}(x_{16}=42=x_{35})$로 $x \leftarrow f(x)$를 35회 수행한다. Nivasch의 스택 이용법은 35회 모두에 대해 단순 추가나 역 탐색 갱신을 수행하는데 반해 제안된 배열 이용법은 $(x_i < 0.5x_{i-1}) \cap (x_i < 0.5(p-1))$의 10회만을 수행하며, 메모리 크기 측면에서도 스택의 4에 비해 배열은 3으로 감소된 효과를 얻는다.

또한, 스택에 단순 추가시키는 경우를 제외한 역추적과 갱신에는 18회를 수행하는데 반해, 배열의 이진 탐색과 갱신에는 4회만을 수행하였다. 결론적으로, Brent의 포인터 방법은 모듈러 연산횟수는 Nivasch의 스택 법에 비해 많이 수행되나 포인터 값 갱신횟수는 매우 적은 장점이 있다. 따라서 2가지 방법 중에서 어떤 것이 보다 좋은지에 대해서는 판단할 수가 없다. 제안된 배열법은 Nivasch의 스택 법과 거의 동일한 모듈러 연산횟수를 수행하지만 배열 값 갱신 횟수는 73% 이상으로 크게 감소시키는 효과를 얻었다. 결론적으로 따라서 단순 추가를 제외한 순수한 배열 값 탐색과 갱신 횟수만을 고려해 볼 때, Brent의 포인터 법보다는 많이 수행되지만 모듈러 연산횟수가 작아 Brent의 포인터 법 보다 좋은 알고리즘이라 할 수 있다.

표 4. $\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}} (\mathrm{mod}\, 1019)$의 모듈러 연산과 메모리 충돌 검증 횟수

Table 4. The number of Modular computation and Collision verification for $\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}} (\mathrm{mod}\, 1019)$

α	모듈러 연산 횟수 (메모리 탐색 횟수)								
	$\alpha^{\gamma^{-1}} = \beta_{\gamma^{-1}}$	μ	λ	사이클 최소값	Pollard의 2 Pointer 법 $x_0 = y_0 = \alpha\beta$ $y = x_{2i}$	Brent의 2 Pointer 법 $x_0 = \alpha\beta$ $y = x_{2^i}\ (i = 2^k)$	Nivasch의 스택 이용법 $x_0 = \alpha\beta$ $y = Stack$	제안된 배열 이용법 $x_0 = \alpha\beta$ $y = Array$	스택 이용법 대비 중복 검증 비율
2	$2^{1003} \equiv 204$	54	27	$x_{39} = 34 = x_{66}$	162 (54)	91 (91)	86 (86)	86 (21)	24.42 %
	$2^{910} \equiv 225$	12	40	$x_{16} = 21 = x_{56}$	120 (40)	104 (104)	56 (56)	56 (14)	25.00 %
	$2^{1111} \equiv 694$	12	37	$x_{25} = 5 = x_{62}$	111 (37)	101 (101)	62 (62)	62 (15)	24.19 %
	$2^{710} \equiv 243$	44	41	$x_{33} = 51 = x_{94}$	186 (82)	105 (105)	94 (94)	94 (21)	22.34 %
	$2^{613} \equiv 668$	32	10	$x_{32} = 9 = x_{42}$	120 (40)	42 (42)	48 (48)	48 (12)	25.00 %
	$2^{510} \equiv 507$	57	37	$x_{55} = 17 = x_{128}$	222 (74)	101 (101)	125 (125)	125 (34)	27.20 %
	$2^{410} \equiv 966$	37	11	$x_{39} = 104 = x_{43}$	135 (45)	73 (73)	48 (48)	48 (11)	22.92 %
	$2^{310} \equiv 303$	38	23	$x_{53} = 16 = x_{31}$	138 (46)	87 (87)	81 (81)	81 (21)	25.93 %
	$2^{210} \equiv 921$	66	21	$x_{79} = 3 = x_{100}$	252 (84)	274 (274)	100 (100)	100 (18)	18.00 %
	$2^{110} \equiv 368$	50	15	$x_{33} = 60 = x_{63}$	180 (60)	76 (76)	65 (65)	65 (17)	26.15 %
	$2^{10} \equiv 261$	18	28	$x_{76} = 37 = x_{84}$	84 (28)	60 (60)	54 (54)	54 (13)	24.07 %
5	$5^{1003} \equiv 225$	10	25	$x_{11} = 144 = x_{36}$	75 (25)	57 (57)	36 (36)	36 (6)	16.67 %
	$5^{910} \equiv 261$	25	22	$x_{40} = 48 = x_{67}$	132 (44)	54 (54)	62 (62)	62 (15)	24.19 %
	$5^{810} \equiv 867$	50	11	$x_{35} = 101 = x_{69}$	165 (55)	75 (75)	69 (69)	**68** (17) $x_{97} = 132 = x_{162}$	24.64 %
	$5^{710} \equiv 69$	29	6	$x_{34} = 14 = x_{92}$	108 (36)	38 (38)	40 (40)	40 (11)	27.50 %
	$5^{610} \equiv 686$	2	9	$x_8 = 224 = x_{12}$	27 (9)	25 (25)	12 (12)	**17** (4) $x_8 = 259 = x_{17}$	33.33 %
	$5^{510} \equiv 721$	18	40	$x_{37} = 19 = x_{77}$	120 (40)	104 (104)	77 (77)	77 (24)	31.17 %
	$5^{410} \equiv 685$	2	35	$x_{34} = 31 = x_{69}$	105 (35)	99 (99)	69 (69)	69 (16)	23.19 %
	$5^{310} \equiv 460$	1	11	$x_3 = 104 = x_{13}$	33 (11)	27 (27)	13 (13)	13 (5)	38.46 %
	$5^{210} \equiv 837$	35	20	$x_{44} = 4 = x_{64}$	120 (40)	84 (84)	64 (64)	64 (20)	31.25 %
	$5^{110} \equiv 391$	9	15	$x_{17} = 29 = x_{37}$	45 (15)	31 (31)	32 (32)	32 (6)	18.75 %
	$5^{10} \equiv 151$	19	48	$x_{49} = 69 = x_{97}$	144 (48)	112 (112)	97 (97)	97 (24)	24.74 %
	평균								**25.41 %**

표 5. 스택과 이중-배열의 모듈러 연산과 메모리 충돌 검증 횟수

Table 5. The number of modular computation and collision verification for Stack and dual-Array method

α	모듈러 연산횟수 (메모리 탐색 횟수)								
	$\alpha^{\gamma} \equiv \beta_{\gamma}$	μ	λ	Nivasch의 스택 이용법	제안된 배열 이용법			스택 이용법 대비 비율	
					$\alpha^{\gamma} \equiv \beta_{\gamma}$	$\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}}$	최소값	모듈러 연산횟수	메모리 탐색횟수
2	$2^{10} \equiv 5$	33	17	60 (60)	**60 (15)**	86 (21)	60 (15)	100.00 %	25.00 %
	$2^{100} \equiv 548$	1	19	35 (35)	**35 (10)**	56 (14)	35 (10)	100.00 %	28.57 %
	$2^{200} \equiv 718$	7	60	73 (73)	73 (15)	**62 (15)**	**62** (15)	84.93 %	20.55 %
	$2^{300} \equiv 130$	7	20	35 (35)	**35 (10)**	94 (21)	35 (10)	100.00 %	28.57 %
	$2^{400} \equiv 929$	11	79	166 (166)	166 (43)	**48 (12)**	**48** (12)	28.92 %	7.23 %
	$2^{500} \equiv 611$	34	12	57 (57)	**57 (12)**	125 (34)	57 (12)	100.00 %	21.05 %
	$2^{600} \equiv 596$	2	33	42 (42)	**42 (10)**	48 (11)	42 (10)	100.00 %	23.81 %
	$2^{700} \equiv 528$	28	46	92 (92)	92 (21)	**81 (21)**	**81** (21)	88.04 %	22.83 %
	$2^{800} \equiv 967$	65	54	161 (161)	161 (39)	**100 (18)**	**100** (18)	62.11 %	11.18 %
	$2^{900} \equiv 36$	12	10	38 (38)	**26 (4)**	65 (17)	**26** (4)	89.29 %	14.29 %
	$2^{1000} \equiv 367$	41	35	104 (104)	104 (23)	**54 (13)**	**54** (13)	51.92 %	12.50 %
5	$5^{10} \equiv 548$	23	60	88 (88)	88 (20)	**36 (6)**	**36** (6)	40.91 %	6.82 %
	$5^{100} \equiv 367$	15	15	41 (41)	**41 (16)**	62 (15)	41 (16)	100.00 %	39.02 %
	$5^{200} \equiv 151$	22	7	33 (33)	**33 (12)**	68 (17)	33 (12)	100.00 %	36.36 %
	$5^{300} \equiv 192$	10	19	36 (36)	**36 (12)**	40 (11)	36 (12)	100.00 %	33.33 %
	$5^{400} \equiv 153$	15	2	17 (17)	19 (5)	**17 (4)**	17 (4)	100.00 %	23.53 %
	$5^{500} \equiv 106$	1	20	39 (39)	**39 (12)**	77 (24)	39 (12)	100.00 %	30.77 %
	$5^{600} \equiv 1017$	66	13	95 (95)	95 (21)	**69 (16)**	**69** (16)	72.63 %	16.84 %
	$5^{700} \equiv 844$	4	27	43 (43)	43 (12)	**13 (5)**	**13** (5)	30.23 %	11.63 %
	$5^{800} \equiv 991$	5	38	59 (59)	**59 (13)**	64 (20)	59 (13)	100.00 %	22.03 %
	$5^{900} \equiv 933$	7	25	32 (32)	32 (8)	**32 (6)**	32 (6)	100.00 %	18.75 %
	$5^{1000} \equiv 27$	12	14	31 (31)	**31 (11)**	97 (24)	31 (11)	100.00 %	35.48 %
	평균							**84.04 %**	**22.28 %**

실제 적용 시에는 $\alpha^{\gamma} \equiv \beta_{\gamma} (\mathrm{mod}\, p)$의 역함수 $\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}}$ $\alpha^{\gamma^{-1}} \equiv \beta_{\gamma^{-1}} (\mathrm{mod}\, p)$을 구하여 이중으로 처리하는 방식을 취할 수 있다. $\gamma + \gamma^{-1} = (p-1)$로 $\gamma = (p-1) - \gamma^{-1}$로 구할 수 있다. 역함수에 대한 실험 결과는 표 4에 제시되어 있으며, Nivasch의 스택 이용법과 이중-배열 방법의 모듈러 연산횟수와 메모리 탐색 횟수 비교는 표 5에 제시되어 있다. 표에서 역함수와 이중으로 배열 처리하면 스택 이용법에 비해 모듈러 연산횟수를 15.96% 감소시킬 수 있으며, 메모리 탐색 횟수를 77.72% 감소시킬 수 있다.

V. 결 론

본 논문은 이산대수 문제에 있어서 사이클을 검출하는 Nivasch의 스택 법의 문제점을 개선하기 위해 배열을 적용하는 방법을 제안하였다. 제안된 방법은 $(x_i < 0.5 x_{i-1}) \cap (x_i < 0.5(p-1))$ 조건을 만족하는 경우에 한해 배열의 데이터에 대해 추가 또는 갱신하는 방법을 적용하였다. 이러한 방법을 적용함으로 인해 배열의 최대 크기도 스택에 비해 감소시켰다. 또한, $x_i < A(1)$인 경우 스택에 비해 단순 추가가 용이하였으며, $A(1) < x_i < A(k)$의 경우 이진탐색 법을 적용할 수 있어 x_i를 배열의 최대치로 하는 배열 데이터 갱신도 빠르게 수행할 수 있었다. 제안된 알고리즘을 적용한 결과 Nivasch의 스택 법에 비해 배열 추가와 탐색 수행횟수를 73% 이상 줄일 수 있음을 보였다.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, Section 31.7 The RSA Public-key Cryptosystem", 2nd Ed., MIT Press and McGraw-Hill. pp. 881-887, 2001.

[2] D. R. Stinson, "Cryptography: Theory and Practice," 3rd ed., London, CRC Press, 2006.

[3] J. M. Pollard, "Monte Carlo Methods for Index Computation(mod p)", Mathematics of Computation, Vol. 32, No. 143, pp. 918-924, Jul. 1978, https://doi.org/10.1090/S0025-5718-1978-0491431-9

[4] R. P. Brent, "An Improved Monte Carlo Factorization Algorithm", Bit Numerical Mathematics, Vol. 20, No. 2, pp. 176-184, Jun. 1980, https://doi.org/ 10.1007/BF01933190

[5] S. Bai and R. P. Brent, "On the Efficiency of Pollard's RhoMethod for Discrete Logarithms", Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory Symposium, Vol. 77, pp. 125-131, Jan. 2008.

[6] E. Teske, "Speeding Up Pollard's Rho Method for Computing Discrete Logarithms", Lecture Notes in Computer Science, Vol. 1423, pp. 541-554, Jun. 1998, https://doi.org/10.1007/BFb0054891

[7] G. Nivasch, "Cycle Detection Using a Stack", Information Processing Letters, Vol. 90, No. 3, pp. 135-140, May 2004, https://doi.org/10.1016/j.ipl.2004.01.016

[8] J. H. Cheon, J. Hong, and M. K. Kim, "Speeding Up the Pollard Rho Method on Finite Fields", 14th International Conference on the Theory and

Application of Cryptology and Information Security, pp. 471-488, Dec. 2008, https://doi.org/10.1007/978-3-540-89255-7_29

[9] D. Shanks, "The Infrastructure of a Real Quadratic Field and its Applications", Proceedings of the 1972 Number Theory Conference, University of Colorado, Boulder, 1972.

[10] S. U. Lee, "Baby-Step Adult-Step Algorithm for Discrete Logarithm," Journal of KIIT, Vol. 11, No. 10, pp. 121-128, Oct. 2013.

[11] S. U. Lee, "Multiple Parallel-Pollard's Rho Discrete Logarithm Algorithm," Journal of The Korea Society of Computer and Information, Vol. 20 No. 8, pp. 29-33, Aug. 2015, https://doi.org/ 10.9708/jksci.2015.20.8.029

[12] S. U. Lee, "Cycle Detection in Discrete Logarithm Using a Queue," The Journal of The Institute of Internet, Broadcasting and Communication, Vol. 17, No. 3, pp. 1-7, Jun. 2017, https://doi.org/10.7236/JIIBC.2017.17.3.1

[13] A. Shamir, "Random Graphs in Cryptography", 7th Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms, 2007.

[14] Wikipedia, "Cycle Detection", http://en.wikipedia.org/wiki/Cycle_detection, Wikimedia Foundation, Inc, Retrieved Jan. 2023.

저 자 소 개

이 상 운(정회원)

- 1987년 : 한국항공대학교 항공전자공학과 (학사)
- 1997년 : 경상대학교 컴퓨터과학과 (석사)
- 2001년 : 경상대학교 컴퓨터과학과 (박사)
- 2003년 : 강원도립대학 컴퓨터응용과 전임강사
- 2004년 ~ 2007.2 : 국립 원주대학 여성교양과 조교수
- 2007.3 ~ 2015.3 : 강릉원주대학교 멀티미디어공학과 부교수
- 2015.4 ~ 현재 : 강릉원주대학교 멀티미디어공학과 정교수
- 관심분야 : 소프트웨어 프로젝트 관리, 개발 방법론, 분석과 설계 방법론, 시험 및 품질보증, 소프트웨어 신뢰성, 인공지능과 빅데이터분석, 최적화 알고리즘
- e-mail : sulee@gwnu.ac.kr