양자 몽고메리 곱셈

장경배

https://youtu.be/CdVSEshB960

HANSUNG UNIVERSITY 한성대학교

CryptoCraft LAB
https://crypto.modoo.at

몽고메리 곱셈

$\boxed{\mathrm{x} \cdot y \bmod N}$

- 우선 곱셈 대상을 몽고메리 도메인에 맞게끔 변환

$$x' \triangleq xR \bmod N.$$

- 몽고메리 덧셈

$$(x \pm y)' \equiv x' \pm y' \pmod N$$

몽고메리 곱셈

$\boxed{x \cdot y \bmod N}$

- 우선 곱셈 대상을 몽고메리 도메인에 맞게끔 변환

$$x' \triangleq xR \bmod N.$$

- **몽고메리 곱셈**

$$(xy)' \equiv (xR)(yR)R^{-1} \equiv x'y'R^{-1} \pmod{N}.$$

$\boxed{xyR \bmod N\ ??}$

$$\textsc{MonPro}(x', y' \mid N, R) \triangleq x'y'R^{-1} \bmod N.$$

몽고메리 곱셈

- **사전연산이 필요**

- R은 단순 2의 지수 승이기 때문에 모듈러, 나눗셈 연산이 간단

ALGORITHM 1: Montgomery Reduction

Require: An m-bit modulus M, Montgomery radix $R = 2^m$, an operand T ,where $T = A \cdot B$ or $T = A \cdot A$ in the range $[0, 2^{2m})$, and $\boxed{\text{pre-computed constant } M' = -M^{-1} \bmod R}$

Ensure: Montgomery product ($Z = \text{MonRed}(T, R, M) = T \cdot R^{-1} \bmod M$)

1: $Q \leftarrow T \cdot M' \bmod \boxed{R}$

2: $Z \leftarrow (T + Q \cdot M)\boxed{/R}$

3: **if** $Z \geq M$ **then** $Z \leftarrow Z - M$ **end if**

4: **return** Z

```
function MontMulGF2k(a, b, r, n, n1)
t:=a*b;
u:=t*n1 mod r;
c:=(t+u*n)/r;
return (Fp!c);
end function;
```

몽고메리 곱셈

```
function BitMontMulGF2k(a,b,n)
ac:=Coefficients(a);
k0:=#ac;
k:=Degree(n);
for i:= (k0+1) to k do
ac[i]:=0;
end for;

c:=Fp!0;
for i:=1 to k do
c:=Fp!(c+ac[i]*b);
c0:=Coefficient(c,0);
c:=Fp!(c+c0*n);
c:=c/x;
end for;
return (Fp!c);
end function;

c1:=BitMontMulGF2k(a,b,n);
c-c1;
```

→ a의 Coefficient 확인

→ $\textsc{MonPro}(x', y' \mid N, R) \triangleq x'y'R^{-1} \bmod N.$

양자 몽고메리 구현

```
function BitMontMulGF2k(a,b,n)
ac:=Coefficients(a);
k0:=#ac;
k:=Degree(n);
for i:= (k0+1) to k do
ac[i]:=0;
end for;

c:=Fp!0;
for i:=1 to k do
c:=Fp!(c+ac[i]*b);
c0:=Coefficient(c,0);
c:=Fp!(c+c0*n);
c:=c/x;
end for;
return (Fp!c);
end function;

c1:=BitMontMulGF2k(a,b,n);
c-c1;
```

```python
def Montgomery(eng):
    a = eng.allocate_qureg(12)
    b = eng.allocate_qureg(12)
    S = eng.allocate_qureg(12)

    for i in range(12):
        finite_mul_add(eng, a[i], b, S)
        CNOT | (S[0], S[3])  # 0(0), #0(3)+0(0)
        #Rotate_right(eng, S)

    All(Measure) | S
    for i in range(12):
        print(int(S[11-i]), end=' ')
    print('\n')

def finite_mul_add(eng, a, b, S):
    for i in range(12):
        Toffoli | (a, b[i], S[i])

def Rotate_right(eng, S):
    #right shift 1_bit (same with (32-c-bit) left)
    for i in range(11):
        Swap | (S[i], S[i + 1])
```

양자 몽고메리 구현

```
function BitMontMulGF2k(a,b,n)
ac:=Coefficients(a);
k0:=#ac;
k:=Degree(n);
for i:= (k0+1) to k do
ac[i]:=0;
end for;

c:=Fp!0;
for i:=1 to k do
c:=Fp!(c+ac[i]*b);
c0:=Coefficient(c,0);
c:=Fp!(c+c0*n);
c:=c/x;
end for;
return (Fp!c);
end function;

c1:=BitMontMulGF2k(a,b,n);
c-c1;
```

→ 일반 Toffoli 사용한 a*b 곱셈

```python
def Montgomery(eng):
    a = eng.allocate_qureg(12)
    b = eng.allocate_qureg(12)
    S = eng.allocate_qureg(12)

    for i in range(12):
        finite_mul_add(eng, a[i], b, S)
        CNOT | (S[0], S[3])  # 0(0), #0(3)+0(0)
        #Rotate_right(eng, S)

    All(Measure) | S
    for i in range(12):
        print(int(S[11-i]), end=' ')
    print('\n')

def finite_mul_add(eng, a, b, S):
    for i in range(12):
        Toffoli | (a, b[i], S[i])

def Rotate_right(eng, S):

    for i in range(11):
        Swap | (S[i], S[i + 1])
```

양자 몽고메리 구현

```
function BitMontMulGF2k(a,b,n)
ac:=Coefficients(a);
k0:=#ac;
k:=Degree(n);
for i:= (k0+1) to k do
ac[i]:=0;
end for;

c:=Fp!0;
for i:=1 to k do
c:=Fp!(c+ac[i]*b);
c0:=Coefficient(c,0);
c:=Fp!(c+c0*n);
c:=c/x;
end for;
return (Fp!c);
end function;

c1:=BitMontMulGF2k(a,b,n);
c-c1;
```

→ CNOT + Right Rotation으로 해결 가능

```python
def Montgomery(eng):
    a = eng.allocate_qureg(12)
    b = eng.allocate_qureg(12)
    S = eng.allocate_qureg(12)

    for i in range(12):
        finite_mul_add(eng, a[i], b, S)
        CNOT | (S[0], S[3])  # 0(0), #0(3)+0(0)
        #Rotate_right(eng, S)

    All(Measure) | S
    for i in range(12):
        print(int(S[11-i]), end=' ')
    print('\n')

def finite_mul_add(eng, a, b, S):
    for i in range(12):
        Toffoli | (a, b[i], S[i])

def Rotate_right(eng, S):

    for i in range(11):
        Swap | (S[i], S[i + 1])
```

양자 몽고메리 구현

$M = x^{12} + x^{3} + 1$

Input : x11, x10, x9 , … x3, x2, x1, $\boxed{\text{x0}}$

x11, x10, x9 , … x3, x2, x1, $\boxed{\text{x0}}$ → `c:=Fp!(c+c0*n);`

Output : x0, x11, x9, …, (x3+x0), x2, x'1 → `c:=c/x;`

1. 모듈러 다항식의 최고 차항 제외하고 CNOT 게이트

2. Rotation right → x0를 그대로 최고 차항으로 써도 됨

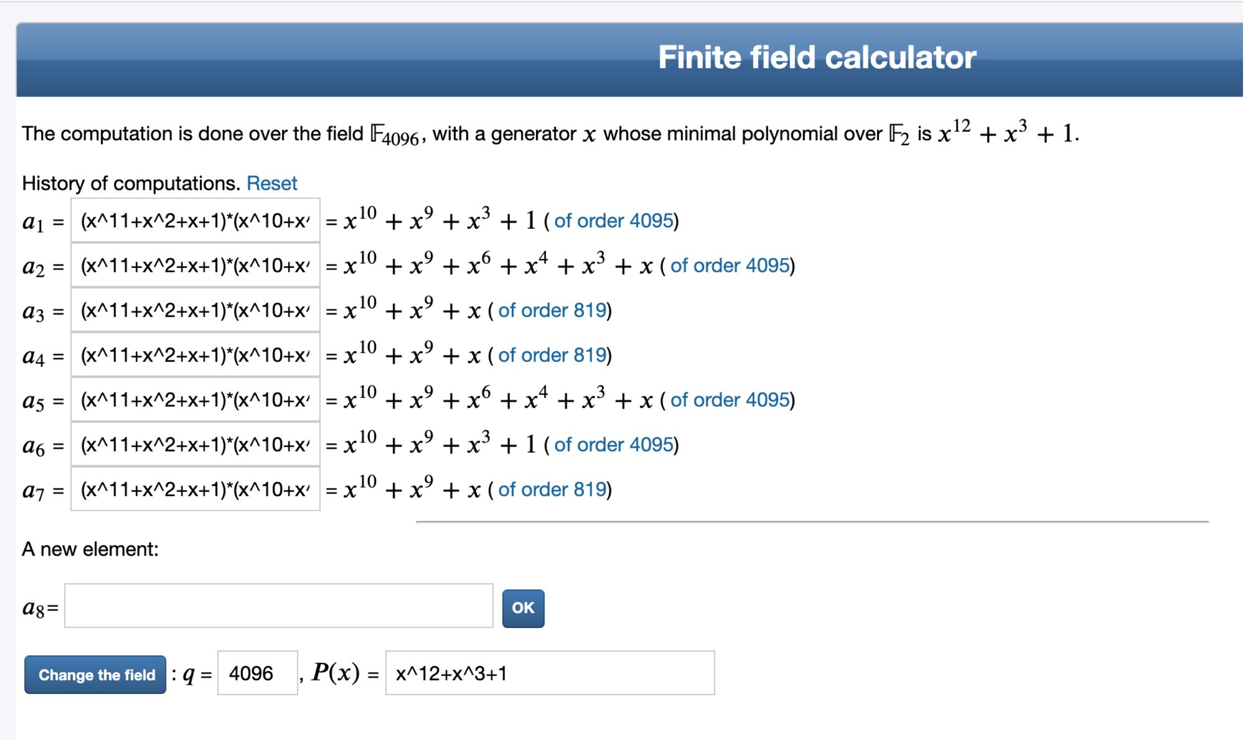
큐빗 없이 CNOT으로만 구현 가능

양자 몽고메리 구현

```python
def Montgomery(eng):
    a = eng.allocate_qureg(12)
    b = eng.allocate_qureg(12)
    S = eng.allocate_qureg(12)

    X | a[0]
    X | a[1]
    X | a[2]
    X | a[11]

    X | b[0]
    X | b[2]
    X | b[10]

    for i in range(12):
        finite_mul_add(eng, a[i], b, S)
        CNOT | (S[0], S[3])  # 0(0), #0(3)+0(0)
        #Rotate_right(eng, S)

    All(Measure) | S
    for i in range(12):
        print(int(S[11-i]), end=' ')
    print('\n')

def finite_mul_add(eng, a, b, S):
    for i in range(12):
        Toffoli | (a, b[i], S[i])

def Rotate_right(eng, S):
    #right shift 1 bit (same with (32-c-bit) left)
    for i in range(11):
        Swap | (S[i], S[i + 1])
```

Resource

```
Gate counts:
    Allocate : 36
    CCX : 144
    CX : 12
    Deallocate : 36
    Measure : 12
    X : 7

Depth : 69.
```



a b r^-1

(x^11+x^2+x+1)*(x^10+x^2+1)*(x^9 + x^6 + x^3)

```
Montgo_Mul
/Users/kb/PycharmProjects/pr
0 1 1 0 0 0 0 0 0 0 1 0
```

Future work

TABLE 1. Comparison of quantum modular multipliers.

Method	Resources	Cuccaro (mod 2^n)	Draper (mod 2^n)	Pham-Svore (mod 2^n)	[13] (mod 2^n)	proposed (mod 2^n)	proposed (mod $2^n - 1$)
quantum	Qubits	$3n$	$5n$	$16n^2$	$5n$	$5n$	$5n$
-	Gates	$12n^2$	$60n^2$	$384n^2 \log_2 n$	$20n^2$	$10n^2$	$10n^2$
classical	Depth	$12n^2$	$24n \log_2 n$	$56 \log_2 n$	$8n \log_2 n$	$4n \log_2 n$	$4n \log_2 n$
quantum	Qubits	X	X	X	X	$6n$	$6n$
-	gates	X	X	X	X	$11n^2$	$11n^2$
quantum	Depth	X	X	X	X	$\frac{1}{2}n^2$	n^2

[13]

Table 1: Resource comparison of in-place quantum modular multipliers. Only the leading order term is shown for each count.

Binary Arithmetic:

Proposal	Architecture	Qubits	Gates†	Depth†
*Cuccaro et. al. [12]	Modular Addition (Ripple-Carry)	$3n$	$12n^2$	$12n^2$
*Draper et. al. [13]	Modular Addition (Prefix)	$5n$	$60n^2$	$24n \log_2 n$
Zalka [4]	Schönhage-Strassen (FFT)	$24 \ldots 96n$	$2^{16}n$	$2^{16}n^{0.2}$
Pham-Svore [18]	Carry-Save (nearest-neighbor)	$16n^2$	$384n^2 \log_2 n$	$56 \log_2 n$
This work	**Exact Division (Prefix)**	$\mathbf{5n}$	$\mathbf{20n^2}$	$\mathbf{8n \log_2 n}$
	Montgomery Reduction (Prefix)	$\mathbf{5n}$	$\mathbf{20n^2}$	$\mathbf{8n \log_2 n}$
	Barrett Reduction (Prefix)	$\mathbf{5n}$	$\mathbf{20n^2}$	$\mathbf{8n \log_2 n}$
	Exact Division (Ripple)	$\mathbf{3n}$	$\mathbf{4n^2}$	$\mathbf{4n^2}$
	Montgomery Reduction (Ripple)	$\mathbf{3n}$	$\mathbf{4n^2}$	$\mathbf{4n^2}$
	Barrett Reduction (Ripple)	$\mathbf{3n}$	$\mathbf{4n^2}$	$\mathbf{4n^2}$

*Reference proposes an adder only. We assume 3 adders per modular add, $2n$ modular adds per multiply.
†Total gate counts and depths provided in TOFFOLI gates.

Future work

ALGORITHM 3: Hybrid Montgomery Reduction

Require: An m-bit modulus M, Montgomery radix $R = 2^m$, and its half radix $R_h = 2^{\frac{m}{2}}$, an operand T, where$T = A \cdot B$ or $T = A \cdot A$ in the range $[0, 2^{2m})$

Ensure: Montgomery product $Z = \text{MonRed}(T, R, M) = T \cdot R^{-1} \bmod M$

$\{CARRY_1, Z_L\},\ Q_L \leftarrow \text{SubMonRed}(T[0, 2^m), M[0, 2^{\frac{m}{2}}), R_h)$

$K_L \leftarrow Q_L \times M[2^{\frac{m}{2}}, 2^m)$

$\{CARRY_2, K_L\} \leftarrow K_L + Z_L + T[2^m, 2^{m+\frac{m}{2}}) \cdot 2^{\frac{m}{2}}$

$\{CARRY_3, Z_H\},\ Q_H \leftarrow \text{SubMonRed}(K_L, M[0, 2^{\frac{m}{2}}), R_h)$

$CARRY_3 \leftarrow CARRY_3 + CARRY_2$

$K_H \leftarrow Q_H \times M[2^{\frac{m}{2}}, 2^m)$

$\{CARRY_4, K_H\} \leftarrow K_H + Z_H + CARRY_1 + T[2^{m+\frac{m}{2}}, 2^{2m}) \cdot 2^{\frac{m}{2}}$

$\{CARRY_4, K_H\} \leftarrow CARRY_3 \cdot 2^{\frac{m}{2}} + \{CARRY_4, K_H\}$

if $\{CARRY_4, K_H\} \geq M$ **then** $K_H \leftarrow \{CARRY_4, K_H\} - M$ **end if**

return K_H