

Chapter 2.

DH 키 교환 알고리즘 구현

2023.09.13.

서민혜

mhseo@duksung.ac.kr

Contents

❖ 수학적 배경

❖ GNU Multiprecision Library

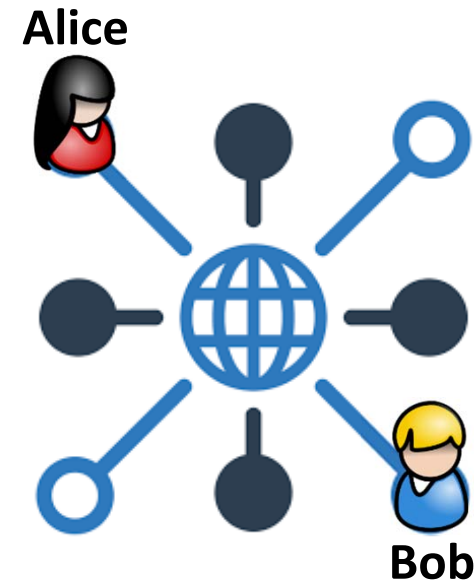
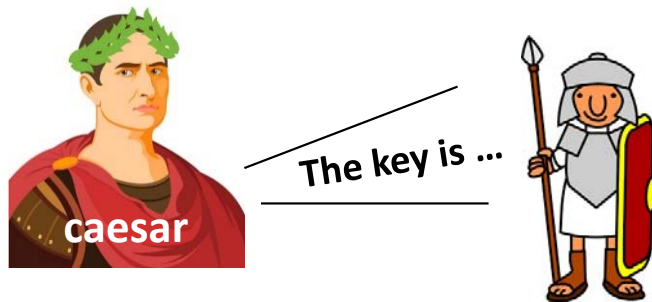
❖ 클래스 구현

- 테스트

수학적 배경

❖ 키교환 (Key exchange)

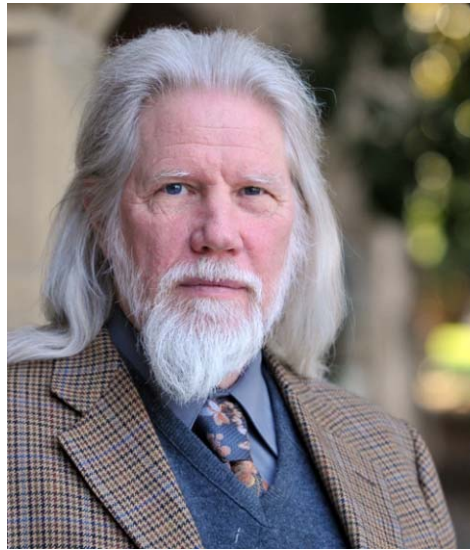
- 사용자들 간에 “안전하게 비밀키를 공유”하는 방법
 - 암호시스템(특히 대칭키 암호)을 사용하기 위한 필수 과정
 - 대칭키 암호의 근본적인 문제인 **키 공유 문제**를 해결



수학적 배경

❖ 디피-헬먼 키교환

- “New Directions in Cryptography” (1976)
 - Whitfield **Diffie** and Martin **Hellman**



(Turing award, 2015)

- 공개키 암호와 전자서명에 대한 개념을 제시 (groundbreaking paper)
- 현재 인터넷에서 사용되는 모든 보안 프로토콜의 기반이 되는 내용

수학적 배경

❖ 디피-헬먼 키교환

- 타인에게 알려져도 상관없는 정보를 두 사람이 교환하는 것만으로 공통의 비밀 정보(비밀키)를 만들어내는 방법
 - 암호 혹은 서명 알고리즘은 아님
 - 공개된 정보를 사용하여 비밀 정보를 생성한다는 의미에서 공개키 암호의 일종으로 볼 수 있음
- 이산 대수 문제(Discrete logarithm problem)의 어려움에 기반하여 설계
 - $y = g^x \bmod p$
 - g, p : 널리 알려진 값 (system parameters)



수학적 배경

❖ 이산 대수 문제 (Discrete logarithm problem)

- 모드 상이 아니면 제곱근을 구하는 것은 어렵지 않음
 - $7^x = 49 \rightarrow x = 2$
 - 하지만 모드 상에서는 쉽지 않은 문제!



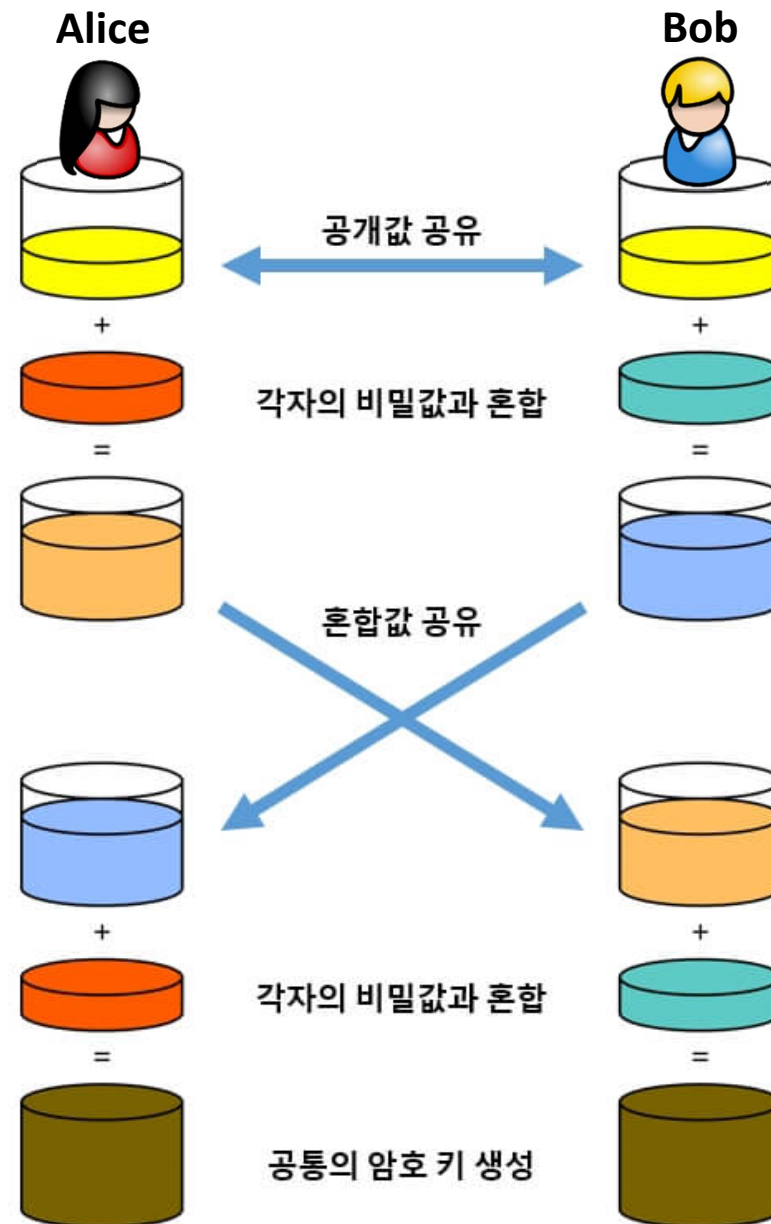
- 암호에서는 모듈러스(modulus) p 를 굉장히 큰 소수(prime)로 설정 (2,048-bit)

961303453135835045741915812806154279093098455949962158225831508796
479404550564706384912571601803475031209866660649242019180878066742
1096063354219926661209

(ex. 512-bit prime)

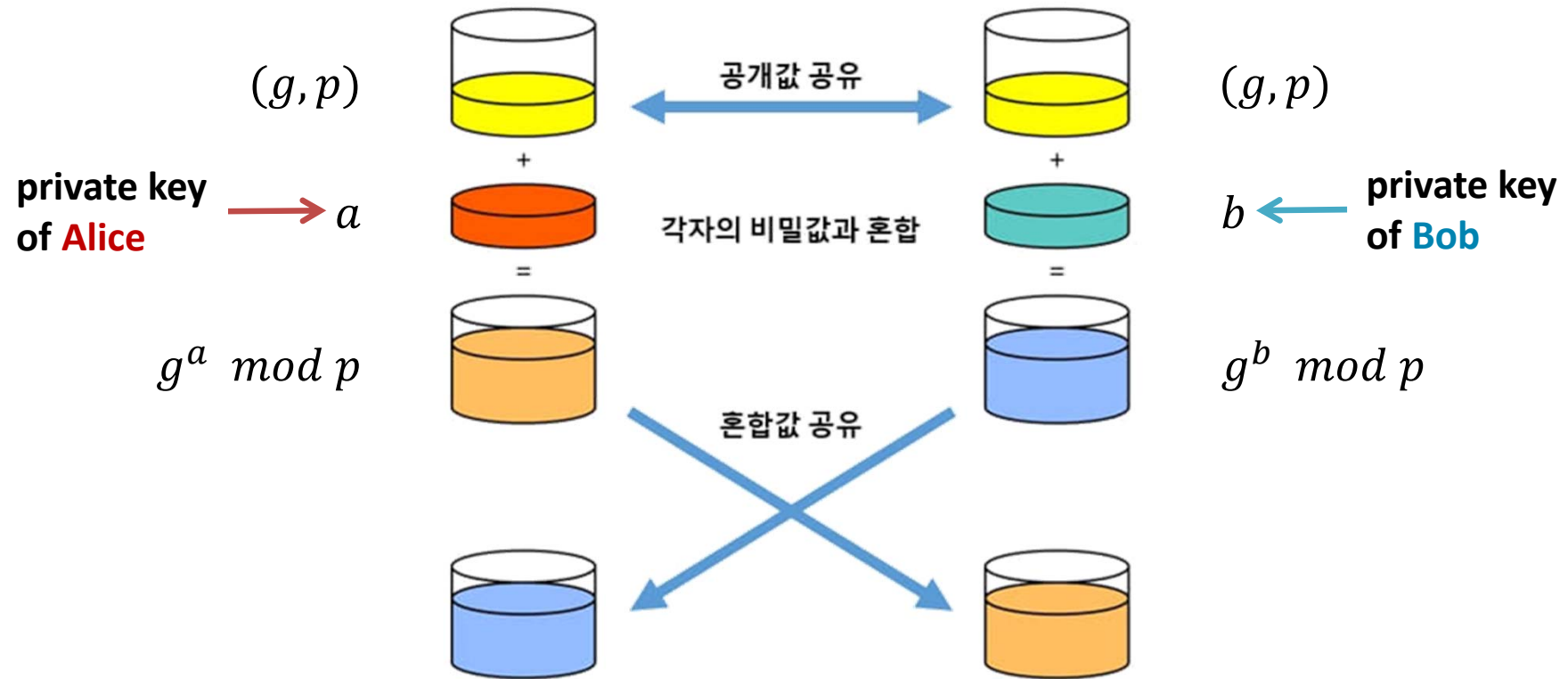
수학적 배경

❖ 디피-헬먼 키교환



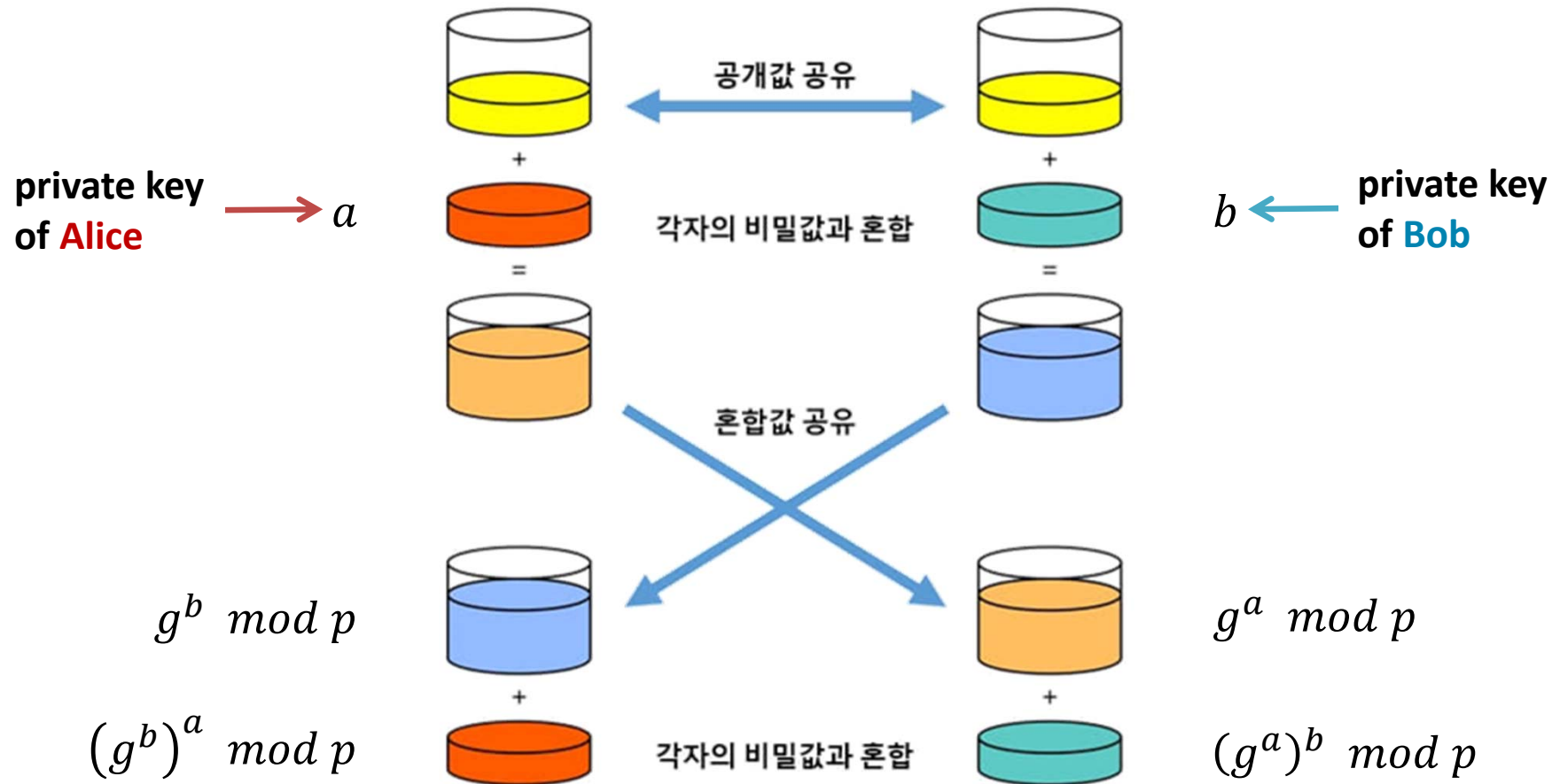
수학적 배경

❖ 디피-헬먼 키교환



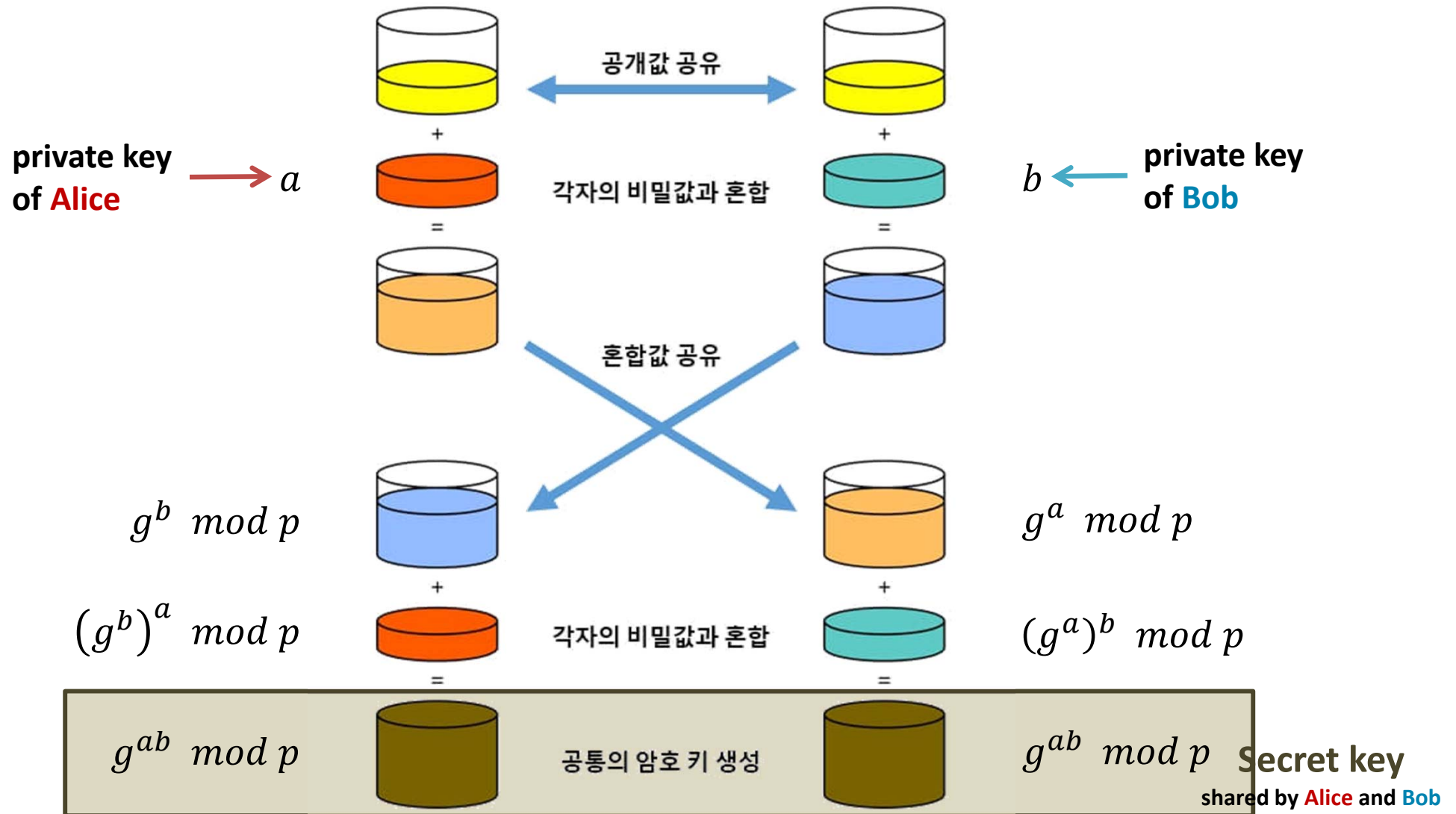
수학적 배경

❖ 디피-헬먼 키교환



수학적 배경

❖ 디피-헬먼 키교환



수학적 배경

❖ 디피-헬먼 키교환의 안전성

- 인터넷 상에서 공개되는 정보
 - (g, p) : 시스템 파라미터
 - $g^a \bmod p$: Alice to Bob
 - $g^b \bmod p$: Bob to Alice

 위의 정보들로부터 공격자는 **Alice와 Bob의 개인키**(a 와 b)를 알아낼 수 있을까?

 위의 정보들을 이용해서 공격자는 **Alice와 Bob이 공유한 비밀키**(g^{ab})를 생성할 수 있을까?

수학적 배경

❖ 디피-헬먼 키교환의 안전성

- 인터넷 상에서 공개되는 정보
 - (g, p) : 시스템 파라미터
 - $g^a \bmod p$: Alice to Bob
 - $g^b \bmod p$: Bob to Alice

 위의 정보들로부터 공격자는 **Alice와 Bob의 개인키**(a 와 b)를 알아낼 수 있을까? **No!**

- 이산 대수 문제의 어려움!

 위의 정보들을 이용해서 공격자는 **Alice와 Bob이 공유한 비밀키**(g^{ab})를 생성할 수 있을까? **No!**

- $g^a \cdot g^b \bmod p = g^{a+b} \bmod p \neq g^{ab} \bmod p$

수학적 배경

❖ 구현 시 고려사항 (1)

• 모드 연산의 간소화

- 모드 연산은 **계산의 도중에 mod**를 취해도 같은 결과가 나옴
 - $7^4 \bmod 12 = (7 \times 7 \bmod 12) \times (7 \times 7 \bmod 12)$
 $= (49 \bmod 12) \times (49 \bmod 12) = 1 \times 1 \bmod 12 = 1$
 - 매우 큰 수의 지수승을 할 때 나머지를 이용해서 계산하면 훨씬 간단
 - $3^{100} \bmod 23 = ?$
 - » **Case 1.**
 $= (3^4)^{25} \bmod 23 = 81^{25} \bmod 23 = (81 \bmod 23)^{25} = 12^{25} \bmod 23$
 - » **Case 2.**
 $= 3^3 \cdot 3^{97} \bmod 23 = (27 \bmod 23) \cdot (3^{25} \bmod 23) = 4 \cdot (3^{25} \bmod 23)$
- ✓ `int powm(int base, int exp, int mod)`

수학적 배경

❖ 구현 시 고려사항 (1)

- 모드 연산의 간소화
 - `int powm(int base, int exp, int mod)`

```
int powm(int base, int exp, int mod)
{
```

```
mhseo@DESKTOP-1065DRA: ~/projects
```

```
mhseo@DESKTOP-1065DRA:~$ ls
projects
mhseo@DESKTOP-1065DRA:~$ cd projects
mhseo@DESKTOP-1065DRA:~/projects$ ls
caesar  caesar.cpp  helloworld  helloworld.cpp  powm  powm.cpp
mhseo@DESKTOP-1065DRA:~/projects$ ./powm
base: 3
exp: 100
mod: 23
result: 3
mhseo@DESKTOP-1065DRA:~/projects$
```

```
}
```

수학적 배경

❖ DH 키교환의 예

$$p = 29 \quad (g, p)$$

$$g = 19$$

$$a = 11 \quad a$$

($\{2, \dots, p-2\}$ 중에 랜덤하게 선택)

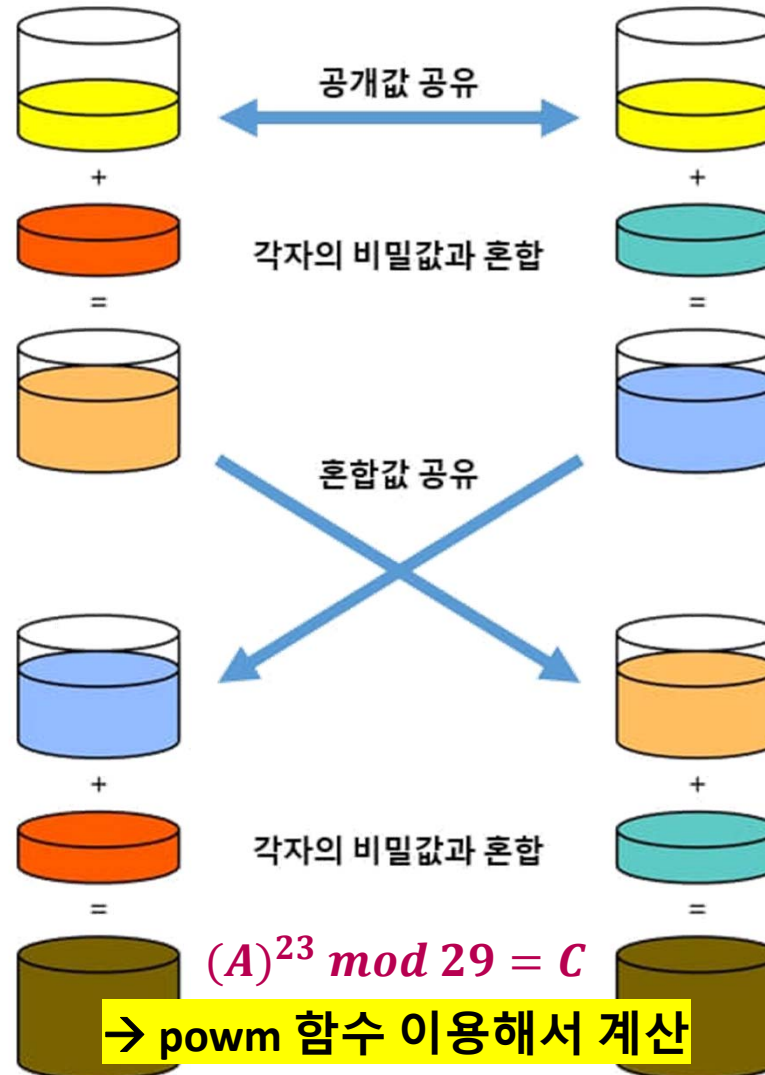
$$g^a \mod p$$

$$19^{11} \mod 29 = A$$

→ powm 함수 이용해서 계산

$$(g^b)^a \mod p$$

$$g^{ab} \mod p$$



$$(g, p) \quad p = 29$$

$$g = 19$$

$$b = 23 \quad b$$

($\{2, \dots, p-2\}$ 중에 랜덤하게 선택)

$$g^b \mod p$$

$$19^{23} \mod 29 = B$$

→ powm 함수 이용해서 계산

$$(g^a)^b \mod p$$

$$g^{ab} \mod p$$

$$(A)^{23} \mod 29 = C$$

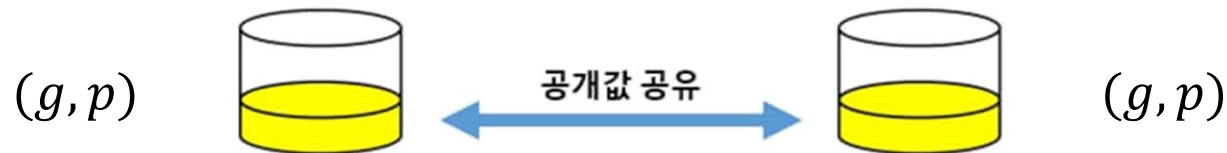
→ powm 함수 이용해서 계산

→ 동일한 키 생성되는지 확인

수학적 배경

❖ 구현 시 고려사항 (2)

- 원시근 (primitive roots)



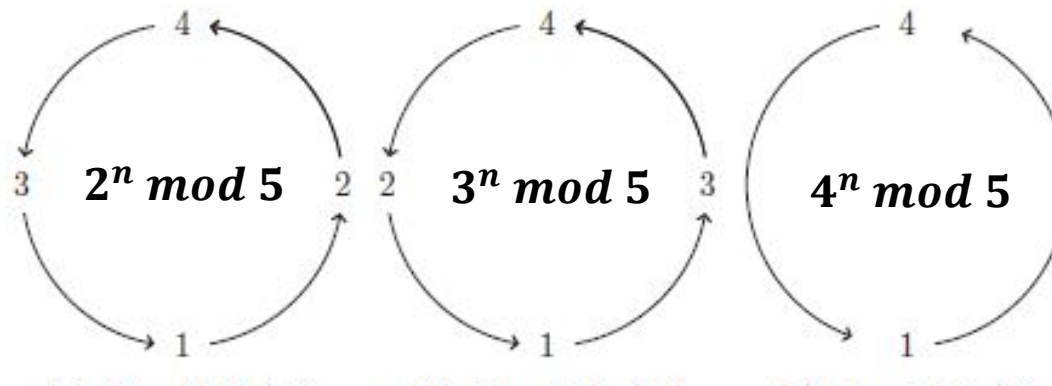
- 디피-헬만 키교환에서 p 가 g 로 나누어 떨어지는 수면 안 됨
 - » $g^a \bmod p, g^b \bmod p$ 연산의 결과값이 모두 0이 됨
- 따라서 일반적으로 모듈러스(modulus) p 를 소수(prime number)로 설정함
 - » 모드 p 상에서 p 로 나누어 떨어지는 수는 없음
- 수학적으로 더 정확하게는 g 는 p 의 **원시근**이어야 함
 - » 소수 p 에 대하여 $g^{p-1} \equiv 1 \pmod{p}$ 를 만족하는 모든 정수 g 를 모듈러스 p 에 대한 원시근이라고 함

수학적 배경

❖ 구현 시 고려사항 (2)

- 원시근 (primitive roots)

- 소수 p 에 대하여 $g^{p-1} \equiv 1 \pmod{p}$ 를 만족하는 모든 정수 g
- $p = 5$ 일 때 원시근은 2와 3 (4는 원시근이 되지 못함)
 - » 2의 경우: $2, 4, 8 \rightarrow 3, 6 \rightarrow 1, 2, \dots$ (5 이하의 모든 수를 거침)
 - » 3의 경우: $3, 9 \rightarrow 4, 12 \rightarrow 2, 6 \rightarrow 1, 3, \dots$ (5 이하의 모든 수를 거침)
 - » 4의 경우: $4, 16 \rightarrow 1, 4, \dots$

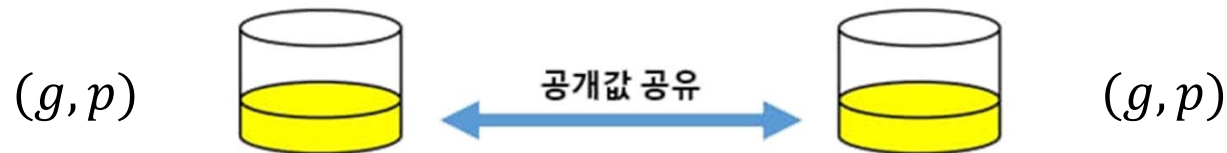


- p 의 원시근으로 g 를 정하면, g 의 임의의 지수승을 p 로 나눈 나머지는 p 이하의 자연수로 고르게 나타남

수학적 배경

❖ 구현 시 고려사항 (2)

- 원시근 (primitive roots)



- 디피-헬만 키교환 구현 시 아래 사항을 꼭 확인하는 과정이 필요
 - » p 가 원하는 크기의 소수가 맞는지
 - » $(p$ 가 주어졌을 때) g 가 p 의 원시근이 맞는지
 - ✓ `bool is_primitive(int base, int mod)`
 - ✓ `vector<int> primitive_root(int mod)`

수학적 배경

❖ 구현 시 고려사항 (2)

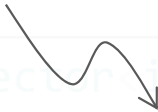
- 원시근 (primitive root)
 - `bool is_primitive(int base, int mod)`

```
bool is_primitive(int base, int mod)
{
    //mod should be prime number
    int exp = 2;
    int r = base * base;
    for(; r != base; exp++)
    {
        r *= base;
        r %= mod;
    }
    return exp == mod;
}
```

수학적 배경

❖ 구현 시 고려사항 (2)

- 원시근 (primitive root)
 - `vector<int> primitive_root(int mod)`



```
vector<int> primitive_root(int mod)
{
    #include <vector>
    vector<int> v;
    for(int base=2; base<mod; base++)
    {
        if(is_primitive(base, mod))
            v.push_back(base);
    }
    return v;
}
```

수학적 배경

❖ 구현 시 고려사항 (2)

- 원시근 (primitive root)

- TEST: 소수 29의 모든 원시근 출력

```
mhseo@DESKTOP-1065DRA:~/projects$ ./primitive
2,3,8,10,11,14,15,18,19,21,26,27,
```

```
#include<vector>
#include<iostream>

using namespace std;

bool is_primitive(int base, int mod);
vector<int> primitive_root(int mod);

int main()
{
    for(int i : primitive_root(29))
        cout << i << ', ';
    cout << endl;
}
```

Contents

❖ 수학적 배경

❖ GNU Multiprecision Library

❖ 클래스 구현

- 테스트

❖ GNU Multi Precision(GMP) Library

- C/C++에서는 암호학에서 사용하는 **매우 큰 수**를 다루기 어려움

```
961303453135835045741915812806154279093098455949962158225831508796
479404550564706384912571601803475031209866660649242019180878066742
1096063354219926661209
```

(ex. 512-bit prime)

- GMP 라이브러리에서는 매우 큰 정수를 다루는 **mpz_class** 제공
 - mpz_class는 일반 정수형처럼 사칙연산이 가능
 - 스트링으로 매우 큰 수를 생성할 수도 있음
 - » Ex) `mpz_class z{"0x3124fd3231feac3122231116764674"}`

GMP function categories

There are several categories of functions in GMP:

1. High-level signed integer arithmetic functions (`mpz`). There are about 150 arithmetic and logic functions in this category.
2. High-level rational arithmetic functions (`mpq`). This category consists of about 35 functions, but all `mpz` functions can be used too, by applying them to the numerator and denominator separately.
3. High-level floating-point arithmetic functions (`mpf`). This is the GMP function category to use if the C type 'double' doesn't give enough precision for an application. There are about 70 functions in this category. New projects should strongly consider using the much more complete GMP extension library **mpfr** instead of `mpf`.

GNU Multiprecision Library

❖ GNU M

- <http://www.gnu.org/software/gmp/>

GNU MP 6.2.1

Next: [Copying](#), Prev: [Index](#)

5.1 Initialization Functions

The functions for integer arithmetic assume that all integer objects are initialized. You do that by calling the function `mpz_init`. For example,

```
{
  mpz_t integ;
  mpz_init (integ);
  ...
  mpz_add (integ, ...);
  ...
  mpz_sub (integ, ...);

  /* Unless the program is about to exit, do ... */
  mpz_clear (integ);
}
```

As you can see, you can store new values any number of times, once an object is initialized.

Function: `void mpz_init (mpz_t x)`

Initialize `x`, and set its value to 0.

Function: `void mpz_inits (mpz_t x, ...)`

Initialize a NULL-terminated list of `mpz_t` variables, and set their values to 0.

Function: `void mpz_init2 (mpz_t x, mp_bitcnt_t n)`

Initialize `x`, with space for `n`-bit numbers, and set its value to 0. Calling this function instead of `mpz_init` or `mpz_inits` is never necessary; reallocation is handled automatically by GMP when needed.

While `n` defines the initial space, `x` will grow automatically in the normal way, if necessary, for subsequent values stored. `mpz_init2` makes it possible to avoid such reallocations if a maximum size is known in advance.

In preparation for an operation, GMP often allocates one limb more than ultimately needed. To make sure GMP will not perform reallocation for `x`, you need to add the number of bits in `mp_limb_t` to `n`.

Function: `void mpz_clear (mpz_t x)`

Free the space occupied by `x`. Call this function for all `mpz_t` variables when you are done with them.

This manual describes...

Copyright ...

Permission ...

License, Ver ...

Cover Text ...

Manual, lik ...

- [Initial](#)
- [Assign](#)
- [Simul](#)
- [Conve](#)
- [Integ](#)
- [Integ](#)
- [Integ](#)
- [Integ](#)
- [Integ](#)
- [Numb](#)
- [Integ](#)
- [Integ](#)
- [I/O of](#)
- [Integ](#)
- [Integ](#)
- [Misce](#)
- [Integ](#)
- [Algorithms](#)
- [Internals](#)

GNU Multiprecision Library

❖ GMP 설치

- 기본 인터페이스는 C언어로 제공
 - 다른 언어에 대해서는 래퍼(wrapper)를 통한 인터페이스를 제공
 - C++, Ocaml, Perl, Python
- 먼저 GMP 라이브러리의 C++ 버전인 gmpxx 설치
 - Ubuntu 실행 후 아래 명령어 입력
 - » **sudo apt update && sudo apt upgrade**
 - » **sudo apt-get install libgmp-dev**
 - » **sudo apt install libgmpxx4ldbl** (숫자 1 아니고 소문자 l 입니다..)
 - » [sudo] password for cyber: **cyber**
 - 설치가 완료되면 visual studio code 실행
 - » **cd projects**
 - » **code .**

GNU Multiprecision Library

❖ GMP 사용

- GMP 라이브러리의 주요 함수

함수 원형	함수 기능
<code>mpz_nextprime(r, z)</code>	z 이후에 오는 첫 소수를 r 에 저장
<code>mpz_powm(r, m, e, K)</code>	$m^e \bmod K$ 를 구해서 r 에 저장
<code>gcd(mpz_class a, mpz_class b)</code>	a 와 b 의 최대공약수를 구함
<code>lcm(mpz_class a, mpz_class b)</code>	a 와 b 의 최소공배수를 구함
<code>mpz_invert(d, e, ϕ)</code>	나머지 역원(modular inverse)를 구함 : $e \cdot \blacksquare \bmod \phi \equiv 1$ 이 되는 값(\blacksquare)을 구해서 d 에 저장
<code>get_ui()</code>	mpz_class를 unsigned int 형으로 변환
<code>mpz_sizeinbase(z, int base)</code>	z 를 base진수로 나타냈을 때의 자릿수 리턴 : <code>mpz_sizeinbase(z, 2)</code> 는 z 를 2진수로 나타냈을 때 자릿수

- **Note.** 앞에 mpz가 붙은 함수는 C 버전밖에 없으므로 C++의 mpz_class에서 C의 mpz 구조체로 바꿔주는 함수 사용 → **get_mpz_t()**

GNU Multiprecision Library

❖ GMP 사용

- C++에서 편하게 사용하기 위해 래퍼(wrapper) 함수 만들기
 - **nextprime** 함수: `mpz_nextprime(r, z)` 이용 $\rightarrow n$ 보다 큰 최초의 소수를 return

```
1
2  #include <gmpxx.h>
3
4  mpz_class nextprime(mpz_class n)
5  {
6      mpz_class r;
7      mpz_nextprime(r.get_mpz_t(), n.get_mpz_t());
8      return r;
9  }
```

- **powm** 함수: `mpz_powm(r, m, e, K)` 이용 $\rightarrow m^e \bmod K$ 값을 return
- **invert** 함수: mp
- **sizeinbase** 함수

```
mpz_class powm(mpz_class base, mpz_class exp, mpz_class mod)
{
    mpz_class r;
    mpz_powm(r.get_mpz_t(), base.get_mpz_t(), exp.get_mpz_t(), mod.get_mpz_t());
    return r;
}
```

GNU Multiprecision Library

❖ GMP 사용

- 모드 p 상에서 랜덤한 값 선택 (개인키로 사용)

5.13 Random Number Functions

The random number functions of GMP come in two groups; older function that rely on a global state, and newer functions that accept a state parameter that is read and modified. Please see the [Random Number Functions](#) for more information on how to use and not to use random number functions.

Function: `void mpz_urandomm (mpz_t rop, gmp_randstate_t state, const mpz_t n)`

Generate a uniform random integer in the range 0 to $n-1$, inclusive.

The variable *state* must be initialized by calling one of the `gmp_randinit` functions ([Random State Initialization](#)) before invoking this function.

Function: `void gmp_randinit_mt (gmp_randstate_t state)`

Initialize *state* for a Mersenne Twister algorithm. This algorithm is fast and has good randomness properties.

– How?

- 1) `gmp_randstate_t` 라는 자료형을 가지는 변수 선언
- 2) `gmp_randinit_mt` 함수를 이용해서 해당 변수를 초기화
- 3) `mpz_urandomm` 함수를 이용해서 p 보다 작은 랜덤값 생성

Contents

❖ 수학적 배경

❖ GNU Multiprecision Library

❖ 클래스 구현

- 테스트

클래스 구현

❖ 공개 파라미터 설정

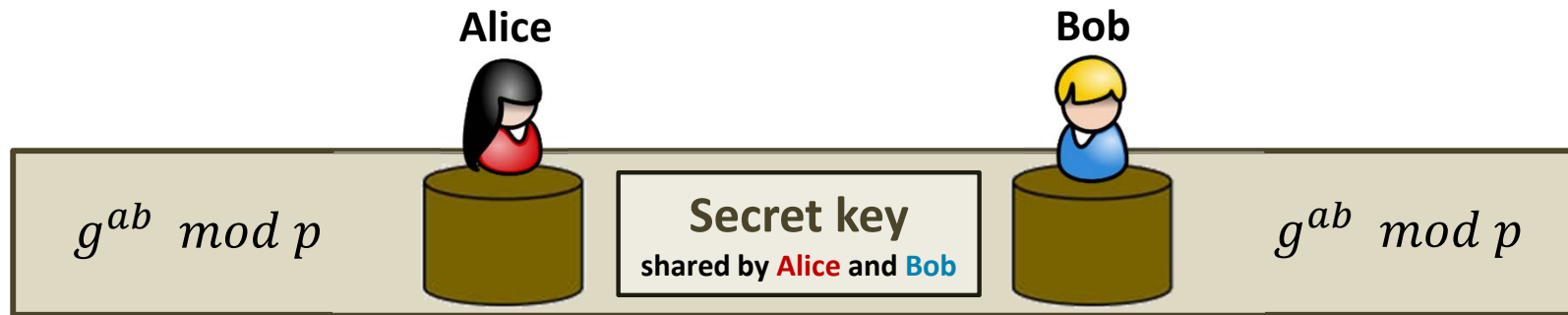
- RFC7919*에서 제공하는 소수 p 와 생성원(generator) g 값을 사용
 - <https://datatracker.ietf.org/doc/rfc7919/>
- 2,048-bit group
 - $p = 2^{2048} - 2^{1984} + \{[2^{1918} * e] + 560316\} * 2^{64} - 1$
 - The hexadecimal representation of p is:
FFFFFFFF FFFFFFFF ADF85458 A2BB4A9A AFDC5620 273D3CF1
D8B9C583 CE2D3695 A9E13641 146433FB CC939DCE 249B3EF9
7D2FE363 630C75D8 F681B202 AEC4617A D3DF1ED5 D5FD6561
2433F51F 5F066ED0 85636555 3DED1AF3 B557135E 7F57C935
984F0C70 E0E68B77 E2A689DA F3EFE872 1DF158A1 36ADE735
30ACCA4F 483A797A BC0AB182 B324FB61 D108A94B B2C8E3FB
B96ADAB7 60D7F468 1D4F42A3 DE394DF4 AE56EDE7 6372BB19
0B07A7C8 EE0A6D70 9E02FCE1 CDF7E2EC C03404CD 28342F61
9172FE9C E98583FF 8E4F1232 EEF28183 C3FE3B1B 4C6FAD73
3BB5FCBC 2EC22005 C58EF183 7D1683B2 C6F34A26 C1B2E9FA
886B4238 61285C97 FFFFFFFF FFFFFFFF
 - The generator is: $g = 2$

* "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS", 2016

클래스 구현

❖ 비교 연산

- 최종적으로 Alice와 Bob이 동일한 키를 공유했는지 확인하려면?



- 비교 연산을 통한 검증

5.10 Comparison Functions

Function: `int mpz_cmp (const mpz_t op1, const mpz_t op2)`

Function: `int mpz_cmp_d (const mpz_t op1, double op2)`

Macro: `int mpz_cmp_si (const mpz_t op1, signed long int op2)`

Macro: `int mpz_cmp_ui (const mpz_t op1, unsigned long int op2)`

Compare *op1* and *op2*. Return a positive value if *op1* > *op2*, zero if *op1* = *op2*, or a negative value if *op1* < *op2*.

» **mpz_cmp** 함수 사용 → “두 값이 같으면 0을 출력”한다는 사실을 이용

클래스 구현

❖ Overview

- `#include <gmpxx.h>`
- 래퍼(wrapper) 함수 정의

```
mpz_class powm(mpz_class base, mpz_class exp, mpz_class mod);  
mpz_class urandomm(gmp_randstate_t state, const mpz_class n);  
int cmp(mpz_class A, mpz_class B);
```

- `main()` 함수
 - 구현에 필요한 모든 변수 선언 (`mpz_class`, `gmp_randstate_t`)
 - 공개 파라미터 (p, g) 설정

```
/* DH parameters for TLS (2,048-bit group) -- RFC7919 */  
/////////////////////////////////////  
mpz_class p{"0xFFFFFFFFFFFFFFFFFFFFFFFADF85458A2BB4A9AAFD5620273D3CF1D8B9C583CE2D3695A9E13641  
mpz_class g = 2;  
//////////////////////////////////////
```

- Alice & Bob: 개인키 생성 – 공개키 계산 – 공유하는 비밀키 계산
- 비교 연산을 통한 검증

테스트

❖ 컴파일 옵션

- gcc를 이용하여 컴파일하는 경우 별도의 플래그가 필요
- 소스 파일(.cpp) 작성이 완료되면
 - Ubuntu 터미널에서 projects 디렉토리로 이동한 후
 - 아래 커맨드 실행 (숫자 1 아니고 소문자 l...)

gcc (파일명).cpp -lstdc++ -lgmpxx -lgmp -o (파일명)

클래스 구현

❖ Overview

- `#include <gmpxx.h>`
- 래퍼(wrapper) 함수 정의

```
mpz_class powm(mpz_class base, mpz_class exp, mpz_class mod);  
mpz_class urandomm(gmp_randstate_t state, const mpz_class n);  
int cmp(mpz_class A, mpz_class B);
```

- `main()` 함수
 - 구현에 필요한 모든 변수 선언 (`mpz_class`, `gmp_randstate_t`)
 - 공개 파라미터 (p, g) 설정
 - Alice & Bob:
 - » 개인키 생성
 - » 공개키 계산
 - » 공유하는 비밀키 계산
 - 비교 연산을 통한 검증

테스트

❖ 컴파일 옵션

- gcc를 이용하여 컴파일하는 경우 별도의 플래그가 필요
- 소스 파일(.cpp) 작성이 완료되면
 - Ubuntu 터미널에서 projects 디렉토리로 이동한 후
 - 아래 커맨드 실행 (숫자 1 아니고 소문자 l...)

gcc (파일명).cpp -lstdc++ -lgmpxx -lgmp -o (파일명)

- 실행 파일이 생성되면 실행시켜서 결과 확인

```
mhseo@DESKTOP-1065DRA:~/projects$ gcc dh.cpp -lstdc++ -lgmpxx -lgmp -o dh
mhseo@DESKTOP-1065DRA:~/projects$ ./dh
Success!
```

테스트

✓ 래퍼(wrapper) 함수 정의

```
mpz_class powm(mpz_class base, mpz_class exp, mpz_class mod);  
mpz_class urandomm(gmp_randstate_t state, const mpz_class n);  
int cmp(mpz_class A, mpz_class B);
```

```
mpz_class powm(mpz_class base, mpz_class exp, mpz_class mod)  
{  
    mpz_class r;  
    mpz_powm(r.get_mpz_t(), base.get_mpz_t(), exp.get_mpz_t(), mod.get_mpz_t());  
    return r;  
}  
  
mpz_class urandomm(gmp_randstate_t state, const mpz_class n)  
{  
    mpz_class r;  
    mpz_urandomm(r.get_mpz_t(), state, n.get_mpz_t());  
    return r;  
}  
  
int cmp(mpz_class A, mpz_class B)  
{  
    int r;  
    r = mpz_cmp(A.get_mpz_t(), B.get_mpz_t());  
    return r;  
}
```

테스트

✓ 메인(main) 함수 정의

```
int main()
{
    mpz_class xA;
    mpz_class xB;
    mpz_class pubA;
    mpz_class pubB;
    mpz_class KA;
    mpz_class KB;

    /* DH parameters for TLS (2,048-bit group) --
    //////////////////////////////////////
    mpz_class
    p{"0xFFFFFFFFFFFFFFFFFFFFFADDF85458A2BB4A9AAFD5C620273D3
    3CE2D3695A9E13641146433FBCC939DCE249B3EF97D2FE3636
    1B202AEC4617AD3DF1ED5D5FD65612433F51F5F066ED085636
    3B557135E7F57C935984F0C70E0E68B77E2A689DAF3EFE8721
    DE73530ACCA4F483A797ABC0AB182B324FB61D108A94BB2C8E
    760D7F4681D4F42A3DE394DF4AE56EDE76372BB190B07A7C8E
    2FCE1CDF7E2ECC03404CD28342F619172FE9CE98583FF8E4F1
    3C3FE3B1B4C6FAD733BB5FCBC2EC22005C58EF1837D1683B2C
    2EFAA886B423861285C97FFFFFFFFFFFFFFFFFFFFF"};
    mpz_class g = 2;
    //////////////////////////////////////
```

```
    gmp_randstate_t state;
    gmp_randinit_mt(state);

    //////////////////////////////////////
    /* Generating public keys */
    //////////////////////////////////////
    /* Alice */
    xA = urandomm(state, p);
    pubA = powm(g, xA, p);
    /* Bob */
    xB = urandomm(state, p);
    pubB = powm(g, xB, p);

    //////////////////////////////////////
    /* Generating K */
    //////////////////////////////////////
    KA = powm(pubB, xA, p);
    KB = powm(pubA, xB, p);

    if(cmp(KA, KB) == 0)
        cout << "Success!" << endl;
    else
        cout << "Failed!" << endl;

    return 0;
}
```

Summary

❖ 수학적 배경

- Diffie-Hellman 키교환 알고리즘
- 이산 대수 문제
- 구현 시 고려사항

❖ GNU Multiprecision Library

- GMP 라이브러리 설치
- GMP 라이브러리 사용

❖ 클래스 구현

- Diffie-Hellman 키교환 알고리즘 구현
- 테스트 (cmp)

Q&A

Thank you ☺