

AIRPLANE  
300KM  
평가표

2023년 12월

위승주, 이재석

Department of Information Security, Cryptology, and Mathematics,  
Kookmin University

## 큰정수 연산 라이브러리 평가표

완성도 (20/20점)	(1-1) 오류나 경고없이 컴파일이 되는가?	2/2
	(1-2) 오류 없이 실행이 되는가?	2/2
	(1-3) 메모리 누수는 없는가?	2/2
	(1-4) 코드 가독성이 좋은가?	2/2
	(1-5) 컴파일러 종속이 없는가?	2/2
	(1-6) 코드 사용이 용이한가? - 테스트 코드를 충분히 제공하는가?	2/2
	(1-7) 8/32/64 비트 단위 연산처리를 모두 지원하는가?	2/2
	(1-8) 다음 연산을 지원하는가? - 1 덧셈, 1 뺄셈, 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승	6/6
구현정확성 (14/14점)	(2-1) 구현 정확성 검증을 위해 충분히 많은 테스트를 했는가?	2/2
	(2-2) 다음 연산이 정확한가? - 8비트: 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승 - 32비트: 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승 - 64비트: 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승	12/12
성능/우수성 (14/16점)	(3-1) Karatsuba 곱셈을 지원하는가?	2/2
	(3-2) 워드 단위 처리 긴나눗셈을 지원하는가?	0/2
	(3-3) Barrett reduction을 지원하는가?	2/2
	(3-4) 라이브러리 성능을 보여주는 예시를 제공하는가?	10/10
기타 (0/0점)	(4-1) 평가 점수에 대한 근거를 충분히 제시하는가?	0
총점		48
평가 의견		
프로젝트 구성원 위승주, 이재석		

## 1. 완성도(20/20점)

### 1-1 오류나 경고없이 컴파일이 되는가?(2/2)

컴파일 진행 시, 오류, 경고는 나타나지 않습니다. 다만, 'realloc'에서 NULL 포인터를 참조할 수 있다는 realloc 함수의 고질적인 경고 처리가 코드 안에서 나타납니다. 이 부분은 수업 시간에 언급된 내용과 같게 불가피한 부분이라고 생각되어 해당 점수는 2점을 주었습니다.

### 1-2 오류 없이 실행이 되는가?(2/2)

리눅스 환경의 gcc, g++에서 오류 없이 실행됨을 확인하였습니다.  
window 환경의 msvc에서 오류가 없음을 확인하였습니다.  
따라서 2점을 주었습니다.

### 1-3 메모리 누수는 없는가?(2/2)

메모리 누수는 없습니다. Valgrind를 통해 확인하였으며 메모리 누수는 발견되지 않았습니다.

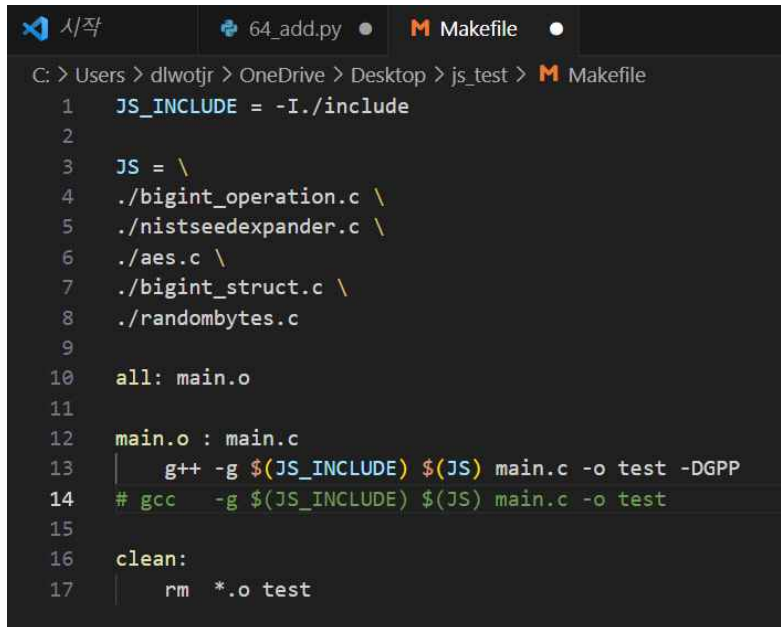
```
unlimit@DESKTOP-0A3UL9S /mnt/c/Users/hojin/Code/js_test $ valgrind --undef-value-errors=no ./test
==2983== Memcheck, a memory error detector
==2983== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2983== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==2983== Command: ./test
==2983==
==2983==
==2983== HEAP SUMMARY:
==2983==   in use at exit: 1,931,400 bytes in 113,972 blocks
==2983== total heap usage: 54,183,576 allocs, 54,069,604 frees, 1,092,448,844 bytes allocated
==2983==
==2983== LEAK SUMMARY:
==2983==   definitely lost: 1,367,664 bytes in 56,986 blocks
==2983==   indirectly lost: 563,736 bytes in 56,986 blocks
==2983==   possibly lost: 0 bytes in 0 blocks
==2983==   still reachable: 0 bytes in 0 blocks
==2983==   suppressed: 0 bytes in 0 blocks
==2983== Rerun with --leak-check=full to see details of leaked memory
==2983==
==2983== For lists of detected and suppressed errors, rerun with: -s
==2983== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

### 1-4 코드 가독성이 좋은가?(2/2)

코드의 가독성은 두 가지의 판단 기준이 있다고 생각합니다. 첫 번째는 주석이 많이 달려있는가? 두 번째는 슈도 코드에 맞춰 작성되었나? 입니다. 첫 번째 판단 기준인 주석은 개별 함수마다 독시젠 형식의 주석을 달고, 함수 내부에 별도의 주석을 추가로 달아 충족한다고 생각했습니다. 두 번째 판단 기준인 슈도코드에 맞춰 작성하는 것은, 최대한 진행 과정에 맞춰 작성하였고, 알고리즘 과정 중 사용되는 연산을 변형하지 않고 이해하기 쉬운 형태로 작성하여 충족한다고 생각하여 2점을 부여했습니다.

### 1-5 컴파일러 종속이 없는가?(2/2)

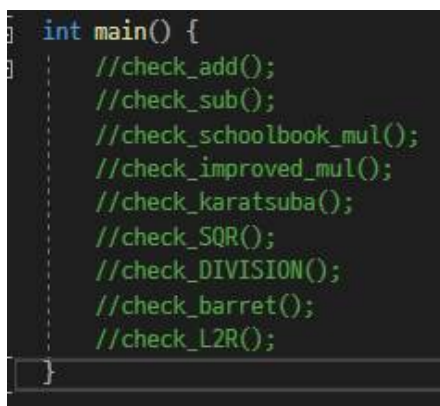
리눅스 환경에서 gcc, g++과 visual studio의 msvc에서 모두 돌아감을 확인하였으며 아래의 사진과 같이 makefile로도 구동이 가능합니다. 확인 결과 오류는 발견되지 않았습니다. 따라서 컴파일러 종속이 없다고 생각합니다. 각 환경에서의 코드를 따로 제출하였습니다.



```
시작 64_add.py ● M Makefile ●
C: > Users > dlwotjr > OneDrive > Desktop > js_test > M Makefile
1 JS_INCLUDE = -I./include
2
3 JS = \
4 ./bigint_operation.c \
5 ./nistseedexpander.c \
6 ./aes.c \
7 ./bigint_struct.c \
8 ./randombytes.c
9
10 all: main.o
11
12 main.o : main.c
13 | g++ -g $(JS_INCLUDE) $(JS) main.c -o test -DGPP
14 # gcc -g $(JS_INCLUDE) $(JS) main.c -o test
15
16 clean:
17 | rm *.o test
```

### 1-6 코드 사용이 용이한가?(2/2)

AES DRBG(리눅스에서는 다른 버전을 사용하였습니다)를 사용하여 랜덤값에 대하여 10000씩의 테스트를 진행하였습니다. 어떤 파일에 저장을 하는 과정은 없으며, 항상 랜덤값을 사용하였습니다. 해당 결과물을 파이썬 코드 형태로 작성하여 연산이 틀릴 시에는 exit()하게 만들었으며 오류는 발견되지 않았습니다. 따라서 테스트 반복횟수가 충분히 진행되었다고 생각합니다. 또한, 아래의 사진과 같이 main.c에 있는 main 함수 안에서 테스트하고 싶은 함수를 쉽게 선택할 수 있습니다. 단순 실행 시 pass만이 뜨는 것을 확인하실 수 있습니다. 각각의 함수의 구동 및 예제를 원하실 경우 main.c 파일의 해당 주석처리를 풀어주시면 해당 함수를 확인하실 수 있습니다.



```
int main() {
    //check_add();
    //check_sub();
    //check_schoolbook_mul();
    //check_improved_mul();
    //check_karatsuba();
    //check_SQR();
    //check_DIVISION();
    //check_barret();
    //check_L2R();
}
```

### 1-7 8/32/64비트 단위 연산처리를 모두 지원하는가?(2/2)

저희의 라이브러리에서 8/16/32/64를 모두 지원함을 확인하였습니다. 해당 결과를 확인하기 위해서는 "bigint\_struct.h"파일의 typedef uint32\_t word;에 대하여 uint8\_t, uint32\_t, uint64\_t에 대해 바꾸어 모두 확인해 보았습니다. check\_pyton\_file에서 확인 가능합니다.

### 1-8 다음 연산을 지원하는가?(12/12)

-덧셈 => 지원합니다(2)

-뺄셈 => 지원합니다(2)

-곱셈 => 지원합니다(2)

-제곱 => 지원합니다(2)

-나눗셈 => 지원합니다(2)

-모듈러지수승 => 지원합니다(2)

## 2. 구현정확성(14/14점)

### 2-1 구현 정확성 검증을 위해 충분히 많은 테스트를 했는가?(2/2)

AES DRBG를 사용하여 랜덤한 길이와 랜덤한 값에 대하여 10,000번씩 테스트를 여러번 진행하였습니다. 또한, 각 uint8\_t, uint32\_t, uint64\_t에 대해서도 동일하게 검증을 진행하였습니다. 또한, 서로 함수를 작성 완료했을 경우마다 크로스 체크하기로 하여 최소 100,000번의 검증을 하였습니다. 따라서 구현 정확성 검증을 위해 충분히 많은 테스트를 했다고 생각합니다.

### 2-2 다음 연산이 정확한가?(12)

(2-1)에서 서술했듯이 각 함수별 자료형마다 최소 100,000번의 검증을 하였고 틀린 부분이 없었으며, 다음 서술된 연산들이 모두 정확하다고 생각합니다.

- 8비트: 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승
- 32비트: 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승
- 64비트: 1 곱셈, 1 제곱, 1 나눗셈, 1 모듈러지수승

### 3. 성능/ 우수성(14/16점)

#### 3-1 Karatsuba 곱셈을 지원하는가? (2/2)

지원합니다.

#### 3-2 워드 단위 처리 긴나눗셈을 지원하는가?(0/2)

지원하지 않습니다.

#### 3-3 Barret Reduction을 지원하는가? (2/2)

지원합니다

#### 3-4 라이브러리 성능을 보여주는 예시를 제공하는가?(10/10)

저희 조는 조금 바른 방법으로 저희의 성능을 보여주려고 합니다. 다른 조와 다르게 저희 조는 정적 bignumber 구현을 함께 진행하였습니다. 동적 bignumber 구현은 여러 경우에 대해 고려할 조건 등이 생겨나며, 동적 할당시 heap 영역에서 데이터가 할당되고 해제되므로 정적 bignumber 구현에 비해 매우 느릴 것으로 예측 하였습니다. 따라서, 정적 bignumber 연산을 구현하여 성능 비교를 진행해보고 싶었고, 정적 bignumber의 상황은 ECDH의 p-256 상황을 가정하여 작성하였습니다. 따라서 이에 대한 성능 비교도 포함되어 있습니다.

측정 방법은 window 상에서 <intrin.h>헤더의 \_\_rdtsc()함수를 사용하였습니다. 또한, 길이 정보를 정적구현과 동일하게 uint32\_t의 8개 배열로 고정시켜 주었습니다. 사용 ide는 visualstudio 2022 x64 release 모드입니다. 또한, 10000번의 평균값을 사용하였고 최적화 옵션 -O2를 사용하여 측정하였습니다.

아래의 성능 표는 동적 할당을 사용한 bignumber 연산 성능 표입니다.

연산	덧셈	뺄셈	Schoolbook 곱셈	항상 곱셈	Karatsuba 곱셈	제곱	나눗셈	Barrett 감산	L2R 감산지수승
성능 (cc)	356	347	24,431	2,253	50,302	3,668	238,572	449,846	42,289,385

동적 할당 성능 측정표 (32bit word \*8)

성능 표에서 눈여겨 볼만한 결과는 곱셈 연산 중에서 Improved 곱셈이 가장 빠른 것입니다. 또한, Karatsuba 곱셈은 Schoolbook 곱셈보다 더 느린 것을 확인 할 수 있습니다. 이러한 이유는 두 가지로 볼 수 있는데, 첫 번째는 Karatsuba의 구조적 특징입니다. Karatsuba는 4개의 곱셈 연산을 3개의 곱셈 연산으로 변경하는 대신, 덧셈과 뺄셈의 수가 더욱 많아지게 됩니다. 따라서 수가 커질수록 Karatsuba의 성능이 더 나을 것입니다. 두 번째는 구현상의 문제입니다. 재귀적인 표현에서 오는 연산부하량이 존재하여 이를 반복문 형태로 풀어서 설계하면 성능이 더 나아질 것으로 예측됩니다. 이처럼 256비트의 작은 배열에서는 오히려 스쿨북

곱셈보다도 느린 것을 볼 수 있었습니다. 따라서 곱셈의 경우 워드 크기를 늘려 다시 한번 성능을 측정해 보았습니다.

다음 성능 측정표의 경우 100~110워드 크기의 경우입니다. 환경은 동일합니다.

연산	Shoolbook 곱셈	Improved 곱셈	Karatsuba 곱셈
성능 (cc)	31,819,934	259,849	9,254,900

동적 할당 성능 측정표 (32bit word \*100~110)

워드의 개수를 늘려서 측정한 경우 Karatsuba의 성능이 Schoolbook에 비해 더욱 나아진 것을 확인할 수 있었습니다. 하지만 위의 경우에도 Improved Shoolbook 곱셈이 가장 성능이 좋은 것을 확인 할 수 있습니다. 하지만, 이전 결과에 비해 그 격차가 줄어든 것을 확인했고 워드 개수를 더 늘릴 경우 Improved Shoolbook 곱셈보다도 빨라질 것을 예상할 수 있습니다.

연산	덧셈	뺄셈	Shoolbook 곱셈	Karatsuba 곱셈	제곱
성능 (cc)	18	45	132	438	207

정적 할당 성능 측정표 (32bit word \*8)

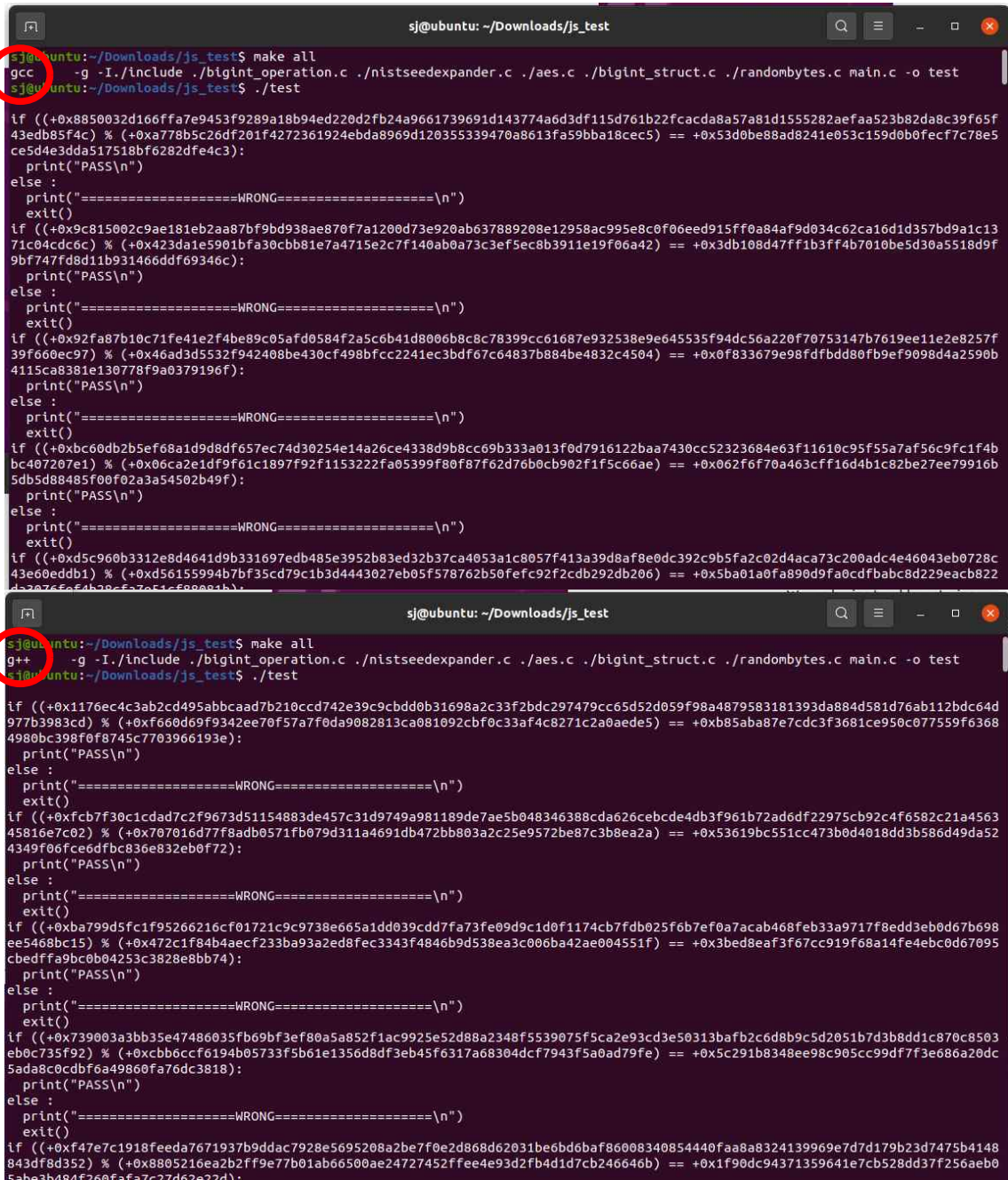
해당 측정표는 정적인 함수의 성능 측정표이며 동적 할당성능 분석 당시와 동일한 조건에서 측정을 진행하였습니다. 모든 연산이 눈에 띄게 빠름을 볼 수 있으며 암호를 static으로 짜는 이유에 대해 생각해 볼 수 있었습니다. 따라서 ECDH를 작성하는 최종 목표에서도 정적 bignumber 연산 라이브러리를 사용하였습니다.

결론적으로 내부 연산간의 성능 비교와 그에 대한 이유를 고찰하였고, 외부 라이브러리와 비교하는 대신 정적 라이브러리를 만들어 비교하였으므로 성능에 대한 예시를 적절하게 보여주었다고 생각하여 10점을 부여했습니다.

#### 4. 평가 점수에 대한 근거를 충분히 제시하는가?(0/0점)

(4-1)평가 점수에 대한 근거를 충분히 제시하는가?

다음과 사진들과 파일로 평가 점수들에 대한 근거를 충분히 제시하였다고 생각합니다.



```
sj@ubuntu: ~/Downloads/js_test
$ make all
gcc -g -I./include ./bigint_operation.c ./nistseedexpander.c ./aes.c ./bigint_struct.c ./randombytes.c main.c -o test
$ ./test

if ((+0x8850032d166ffa7e9453f9289a18b94ed220d2fb24a9661739691d143774a6d3df115d761b22fcacda8a57a81d1555282aefaa523b82da8c39f65f
43edb85f4c) % (+0xa778b5c26df201f4272361924ebda8969d120355339470a8613fa59bba18cec5) == +0x53d0be88ad8241e053c159d0b0fecf7c78e5
ce5d4e3dda517518bf6282dfe4c3):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0x9c815002c9ae181eb2aa87bf9bd938ae870f7a1200d73e920ab637889208e12958ac995e8c0f06eed915ff0a84af9d034c62ca16d1d357bd9a1c13
71c04cdc6c) % (+0x423da1e5901bfa30cbb81e7a4715e2c7f140ab0a73c3ef5ec8b3911e19f06a42) == +0x3db108d47ff1b3ff4b7010be5d30a5518d9f
9bf747fd8d11b931466ddf69346c):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0x92fa87b10c71fe41e2f4be89c05afd0584f2a5c6b41d8006b8c878399cc61687e932538e9e645535f94dc56a220f70753147b719ee11e2e8257f
39f660ec97) % (+0x46ad3d5532f942408be430cf498bfcc2241ec3bdf67c64837b884be4832c4504) == +0x0f833679e98fddfbd80fb9ef9098d4a2590b
4115ca8381e130778f9a0379196f):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0xabc60db2b5ef68a1d9d8df657ec74d30254e14a26ce4338d9b8cc69b333a013f0d7916122baa7430cc52323684e63f11610c95f55a7af56c9fc1f4b
bc407207e1) % (+0x06ca2e1df9f61c1897f92f1153222fa05399f80f87f62d76b0cb902f1f5c66ae) == +0x062f6f70a463cff16d4b1c82be27ee79916b
5db5d88485f0f02a3a54502b49f):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0xd5c960b3312e8d4641d9b331697edb485e3952b83d32b37ca4053a1c8057f413a39d8af8e0dc392c9b5fa2c02d4aca73c200adc4e46043eb0728c
43e60edd01) % (+0xd56155994b7bf35cd79c1b3d443027eb05f578762b50f9c92f2c2b292db206) == +0x5ba01a0fa890d9fa0cdfbabc8d229eac822
d3207656f4b28c7a7c51c68081b1):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0x7c7b30c1cdad7c2f9673d51154883de457c31d9749a981189de7ae5b048346388cda626cebcde4db3f061b72ad6df22975cb92c4f6582c21a4563
45816e7c02) % (+0x707016d77f8adb0571fb079d311a4691db472bb803a2c25e9572be87c3b8ea2a) == +0x53619bc551cc473b0d4018dd3b586d49da52
4349f06fce6dfbc836e832eb0f72):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0xba799d5fc1f95266216cf01721c9c9738e665a1dd039cdd7fa73fe09d9c1d0f1174cb7fdb025f6b7ef0a7acab468feb33a9717f8edd3eb0d67b698
ee5468bc15) % (+0x472c1f84b4aecf233ba93a2ed8fec3343f4846b9d538ea3c006ba42ae004551f) == +0x3bed8eaf3f67cc919f68a14fe4ebc0d67095
cbcdfa9bc0b04253c3828e8bb74):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0x739003a3bb35e47486035fb69bf3ef80a5a852f1ac9925e52d88a2348f5539075f5ca2e93cd3e50313bafb2c6d8b9c5d2051b7d3b8dd1c870c8503
eb0c735f92) % (+0xcbb6ccf6194b05733f5b61e1356d8df3eb45f6317a68304dcf7943f5a0ad79fe) == +0x5c291b8348ee98c905cc99df7f3e686a20dc
5ada8c0cdfb6a49860fa76dc3818):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
if ((+0xf47e7c1918feeda7671937b9ddac7928e5695208a2be7f0e2d868d62031be6bd6baf86008340854440faa8a8324139969e7d7d179b23d7475b4148
843df8d352) % (+0x8805216ea2b2ff9e77b01ab66500ae2472f452f7ee493d2fb4d1d7cb246646b) == +0x1f90dc94371359641e7cb528dd37f256aeb0
5abe3b484f260fafa7c27d6e22d):
    print("PASS\n")
else :
    print("=====WRONG=====n")
    exit()
```

-리눅스 환경에서 gcc/g++실행사진



```
선택 C:\Users\wcoala_guest\Desktop\WWIWClassW3-2WApplication\WAirplane_300km_vs\Wx64WRelease\WAirplane_300km_vs.exe
if ((-0x9f32afea2ea5af7dfcbad33b15a5dd7f27069966129b8f46cc47cbc7c5420668) + (-0xd4e7d7ca68116975ca3afe28) == -0x9f32afea2ea5af7dfcbad33b15a5dd7f27069966e78367113459353d8f7d0490):
    print("PASS\n")
else:
    print("=====WRONG=====\\n")
    exit()
if ((-0x3dc80a82) + (-0xf3a9f195dc9634b40f79bd45f1722bf5cd4a90f9e4f7dc3cb1a10479cacab6a8) == -0xf3a9f195dc9634b40f79bd45f1722bf5cd4a90f9e4f7dc3cb1a1047a0892c12a):
    print("PASS\\n")
else:
    print("=====WRONG=====\\n")
    exit()
if ((+0x0bb08a304bd6e7368a10df0f17f90edf43d5bda5aa2f4e8a8ce6466de05f1272d52df240) + (+0xee057281ceaa4d4) == +0x0bb08a304bd6e7368a10df0f17f90edf43d5bda5aa2f4e8a8ce6466ecf3f699af2189714):
    print("PASS\\n")
else:
    print("=====WRONG=====\\n")
    exit()
if ((+0xb597f28e88f950e4021f67b83622829e) + (+0xf5a258ae) == +0xb597f28e88f950e4021f67b92bc4db4c):
    print("PASS\\n")
else:
    print("=====WRONG=====\\n")
    exit()
if ((-0xb9a08696) + (+0xae07f625) == -0x0b989071):
    print("PASS\\n")
else:
    print("=====WRONG=====\\n")
    exit()
if ((+0x6fa643f5c40f5cf0607de9eb05f584a471d37f385e3d5c97836258ce2dd6999f6c5e7093) + (+0xa107ac57292efbd9663944288dcef89f
```

-visual studio 2022 실행사진

-오류가 없으며 경고 또한 없음을 볼 수 있습니다.

-완성도, 컴파일러 종속성이 없음

-자료형 변경에도 구애받지 않음

```
563 int main() {
564     check_add();
565     //check_sub();
566     //check_schoolbook_mul();
567     //check_improved_mul();
568     //check_karatsuba();
569     //check_SQR();
570     //check_DIVISION();
571     //check_barret();
572     //check_L2R();
573 }
```

-main사진

-주석처리만 지우고 원하는 함수를 실행 가능하며 간단한 주석 지우기로 성능까지 보실 수 있습니다.

-코드 사용 용이성

