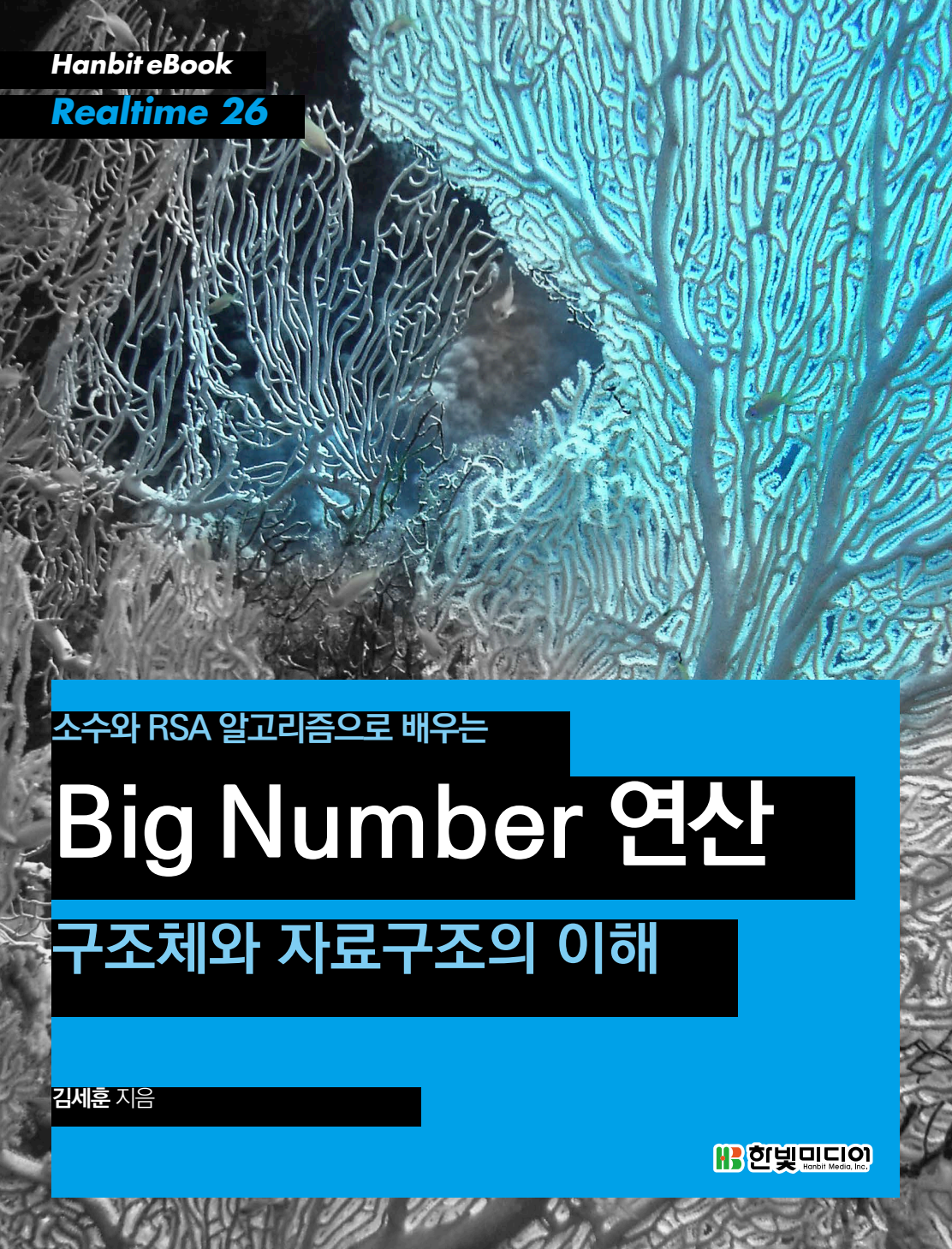

Hanbit eBook
Realtime 26
소수와 RSA 알고리즘으로 배우는
Big Number 연산
구조체와 자료구조의 이해
김세훈 지음
한빛미디어
Hanbit Media, Inc.

소수와 RSA 알고리즘으로 배우는

Big Number 연산

구조체와 자료구조의 이해

소수와 RSA 알고리즘으로 배우는 **Big Number 연산 – 구조체와 자료구조의 이해**

초판발행 2013년 05월 21일

지은이 김세훈 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-607-4 15000 / **정가** 9,900원

책임편집 배용석 / **기획** 이중민 / **편집** 김연숙
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 박진희
마케팅 박상용, 박주훈, 정민하

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ **김세훈**

동국대학교에서 수학을 전공하고, University of Waterloo에서 컴퓨터 과학을 전공해 두 개의 학사 학위를 받았다. 현재 (주)에임투지 기술 연구소에서 선임연구원으로 재직 중이다. 70살이 되었을 때도 멋진 프로그래머라는 말을 듣는 것이 꿈이며 지금도 그 꿈을 만들어 가는 중이다.

저자 서문

이 책은 프로그래밍을 공부하기 위한 'How-to Series'의 첫 번째 주제며, 구조체와 자료구조를 이해하여 컴퓨터가 가진 한계를 소프트웨어로 어떻게 극복할 수 있는지를 알아본다. 예를 들어 계산기로 '1 / 3'의 연산을 실행하면 0.3333333333이라는 결과가 도출된다. 그러나 '0.3333333333 × 3'의 연산을 실행하면 1이 되지 않고 0.9999999999라는 결과가 나온다. 즉, 계산기가 스스로 처리할 수 있는 수의 한계가 존재한다는 뜻이다. 이 책에서 설명하는 Big Number는 이러한 수의 한계를 극복하기 위한 노력의 일부라 할 수 있다.

우주로 로켓을 쏘려면 슈퍼컴퓨터로는 연산해야 한다고 들은 적이 있다. 그렇다면 슈퍼컴퓨터는 우리가 사용하는 일반적인 컴퓨터와 무엇이 다를까? 어쩌면 하드웨어 성능이 월등히 뛰어나서 연산을 빠르게 할 뿐, 소프트웨어에는 큰 변화가 없을지도 모른다. 그렇다면 하드웨어의 한계를 소프트웨어로 극복해야만 한다. 따라서 Big Number는 하드웨어의 한계를 소프트웨어로 극복하려는 방법의 하나일 수 있으며, 여러분이 더 나은 소프트웨어를 만들 수 있게 도와주는 시작일 수 있다.

Big Number 연산은 사칙 연산과 고급 연산을 다루는데, 정수만으로 연산을 하는 만큼 알고리즘의 대부분은 누구나 쉽게 이해할 수 있다. 즉, 이 책에서는 쉬운 연산 과정을 코드로 어떻게 구현하는지를 다룬 후, 사칙 연산으로는 큰 소수를 구하는 연산을 실행하고 고급 연산은 현재 가장 많이 사용하는 공개키 암호화 알고리즘인 RSA의 연산을 실행한다.

이 책의 프로그래밍 언어는 C에 기반을 두고 있다. 그리고 가장 중요한 내용은 구조체를 이용해 새로운 자료형Data Type을 만드는 것이다. 즉, 구조체structure로 C의 자료형에는 존재하지 않는 새로운 자료형을 만드는 것이다.

사실 자료형을 만드는 구조체가 발전해 C++의 클래스class가 되었다. 그만큼 프로그래밍에서 구조체가 얼마나 중요한지를 알 수 있다. 필자의 기준에서 C를 잘 이해한다는 것은 프로그램에서 사용하는 자료형을 가장 먼저 이해할 수 있어야 한다는 것을 말한다. 따라서 이 책에서 다루는 새로운 자료형을 이해하고 나면, 어떠한 자료형도 자유자재로 만들 수 있을 것이다.

Big Number 연산의 응용은 크게 두 가지 주제로 나눌 수 있다. 첫째는 '소수Prime Number'고, 둘째는 암호 알고리즘 'RSA'의 구현이다. 소수의 개념은 중학교 수학 수준이며, 'RSA'는 고등학교 수학 수준이다. 단지 여기에 C 기반의 프로그래밍 기법이 가미된 것으로 모르는 부분이 있을 때는 C 입문서를 보면서 이해하면 된다.

프로그래밍은 프로그래밍 언어를 완전히 익힌 후 하는 것이 아니라 프로그래밍 언어의 도움을 받는 것이다. 외우는 분야가 아니고 이해하는 일이 중요하다. 필요한 정보는 책이나 인터넷을 통해 얻을 수 있으므로 이 책을 보기 위해 C 입문서를 정독할 필요는 없다. 혹시 C 입문서를 다 읽고도 프로그래밍을 구현하지 못하겠다면 주제를 정하고 무작정 프로젝트를 진행해보기 바란다. 내가 아는 프로그래밍이란 모든 것을 알고 하는 것이 아니라 그때마다 필요한 것을 맞추어 조립하는 장난감 같은 것이다. 이제 이 책을 통해 내 자식 같은 프로그램을 출가시키려 한다. 내 코드가 어떤 누군가의 인생에 조금이나마 도움이 될 수 있기를 바란다.

마지막으로 필자의 첫 책인 Big Number 연산을 어머니 문언연 여사에게 바친다.

집필을 마치며

김세훈

대상 독자 및 도서 구성

초급 **초중급** 중급 중고급 고급

이 책의 독자는 C 기초를 이해한 초 · 중급자를 대상으로 하며 알고리즘 학습에 도움을 주는 기본적인 수학 함수들을 코드로 어떻게 구현하는지를 설명한다. Big Number라는 주제는 펜으로 계산하기 어려운 큰 수에 대한 것이지만, 여기서 다루는 알고리즘은 초등학교의 사칙 연산부터 시작하므로 쉽게 이해를 할 수 있을 것이다. 큰 소수를 구하는 일은 사칙 연산만으로 계산이 이루어지므로 간단할 수 있으며, 고급 연산은 현재 가장 많이 사용하는 공개키 암호화 알고리즘인 RSA의 이해를 돕기 위해 다룬다. 하지만 RSA를 위한 고급 연산도 결국은 고등학교 수학의 범위를 벗어나지는 않는다. 실제 RSA 연산은 이 책에서 다루는 방법으로 코드를 구현하지 않지만 RSA를 이해할 때 Big Number의 고급 연산으로 많은 도움을 받을 수 있을 것이다.

이 책에 수록된 소스 코드는 특정 플랫폼에 종속되지 않으므로 윈도우 또는 리눅스에서 실행할 수 있으며, 'How-to Series'는 프로그래밍 방법론에 초점을 맞추어 쓴 책이다. 일방적인 지식의 전달이라기보다는 '이러한 방법도 있구나'라고 이해할 수 있기를 바란다.

그리고 이 책에서는 설명하지 않지만, 이 책의 소스 코드를 C++로 구현한 예제 파일을 함께 제공한다. 이 책의 내용을 끝까지 공부한 다음 C++로 구현한 소스 코드를 스스로 분석해본다면 독자 여러분의 프로그래밍 실력도 한층 더 성장할 것으로 기대한다.

예제 테스트 환경 및 예제 파일 다운로드

사용 프로그램	설명
Visual Studio 2008 이상	코드 예제는 윈도우 환경에서 테스트했다
모든 운영체제	운영체제에 종속되지 않는 프로그램이다

● 예제 파일 다운로드

http://www.hanb.co.kr/exam/2607

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook 입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | 왜 Big Number인가?

프로그래밍 언어는 우리가 일상적으로 사용하는 언어와 비슷하다. 각 나라의 언어가 다양하듯 프로그래밍 언어도 여러 가지 종류가 있으며, 그중 하나의 언어를 선택해 책을 쓰듯 우리는 프로그래밍 언어 하나를 선택하여 프로그램을 만들고 있다.

그런데 일상 언어에서 중요한 것이 문법이 아니라 내용의 전달이듯 프로그래밍 언어에서 정의한 연산자들을 사용하는 것은 기초일 뿐이다. 중요한 것은 함수 사이의 데이터 전달이다. 이러한 의미로 보면 프로그램을 이해하는 데 있어 가장 중요한 것은 전달하는 데이터가 무엇이냐를 아는 것인데, 만약 회사에서 C로 만든 프로그램이 있을 때 전달하는 데이터가 무엇인지 잘 알려면 구조체structure가 어떻게 이루어져 있는지 파악해두는 것이 좋다.

프로그래밍 언어의 기초를 다루는 책에서는 대부분 구조체를 간단히 다루는데, 필자는 실무에서 가장 중요한 것이 구조체라는 점을 강조하고 싶다. 구조체가 발전해 클래스class가 되었으며 C에서 구조체를 잘 다루게 되면 프로그램의 구조를 잘 설계할 수 있다. 물론 프로그램을 만드는 일은 기본적으로 함수를 생성해 나가는 작업이지만 함수를 만들기 전에 구조체를 만들어야 내용을 담아 전달하기 위한 도구로 함수를 사용한다. 즉, 필자가 이 책에서 Big Number를 다루는 이유는 구조체 개념을 이해하는 일이 앞으로의 프로그래밍 인생에 많은 도움이 되기 때문이다.

빅데이터 시대에 접어들면서 이러한 일은 더 빈번하게 발생할 것이다. 하지만 지금부터 설명하는 Big Number는 메모리memory가 허락하는 한 어떠한 수도 표현할 수 있다. 즉, Big Number는 대량의 데이터를 처리할 때 수반되는 하드웨어의 한계를 소프트웨어로 극복할 수 있는 기초 개념이다.

2 | 새로운 자료형 정의

Big Number의 핵심은 구조체로 새로운 자료형을 정의하는 일이다. 이 새로운 자료형을 정의하다 보면 중급 프로그래머로 도약하는 데 필요한 알고리즘을 이해할 수 있게 될 뿐만 아니라 메모리 중심의 자료구조를 이해하면서 큰 데이터를 자유롭게 다룰 수 있는 원리를 이해할 수도 있게 된다.

지금부터는 Big Number를 저장하기 위해 구조체로 새로운 자료형을 만들어서 필요한 연산을 실행할 것이다. 이 책에서는 10진수로 저장하는 구조체와 2진수로 저장하는 구조체 두 가지를 다룬다. 지금부터 살펴보도록 하자.

2.1 포인터

구조체를 잘 사용하려면 먼저 포인터를 제대로 이해할 수 있어야 한다.

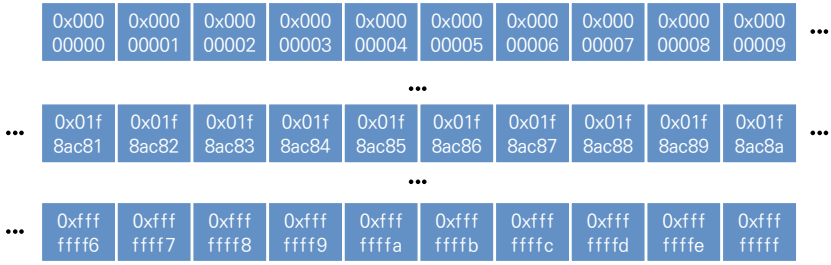
포인터를 잘 사용한다는 것은 하드웨어의 한계를 파악하고 소프트웨어로 하드웨어를 효율적으로 사용할 수 있다는 의미이기도 하다. 자바Java 같은 객체 지향 프로그래밍 언어는 포인터 연산을 없애 프로그래머를 편하게 해주었다고 한다. 하지만 이러한 객체 지향 프로그래밍 언어는 포인터 연산만 없앴을 뿐 객체Object를 포인터 개념으로 사용한다. C/C++가 어려운 이유 중 하나는 포인터 때문이며, C/C++로 프로그램을 만들어본 사람은 다른 프로그래밍 언어를 쉽게 배울 수 있지만 포인터를 사용해보지 않은 프로그래머는 C/C++를 어렵게 생각한다. 사실 포인터를 자유자재로 사용할 수 있다면 다른 프로그래밍 언어를 다루는 것은 따분한 일이라고 생각할 정도다. 포인터를 잘 다룬다는 것은 메모리를 잘 이해한다는 것이므로 먼저 메모리에 관해 알아야 한다.

1차원 메모리RAM 구조를 생각해보자. 물론 하드웨어 구조상으로는 2차원이지만, 여기서 다루는 메모리 개념은 순차적인 것으로서 자신의 주소를 가지므로 1차원 구조라고 생각해도 무방하다. 그림 2-1은 주소를 가진 1차원 행렬로 각 셀cell은 1바이트를 가지게 된다.

그림 2-1 순차적인 1차원 메모리 구조

위와 같은 메모리 구조는 2차원 배열이어도 순차적인 번호를 갖게 되며, 순차적인 번호는 address라고 불리는 메모리의 주소다. 실제 메모리 관리는 운영체제가 담당하므로 프로그래머는 메모리 구조를 자세히 알 필요는 없다. 단지 운영체제가 할당하는 메모리 주소만 알면 메모리를 읽고 쓸 수 있다. 그림 2-2는 32비트 운영체제에서 어떻게 주소를 지정하는지를 보여준다.

그림 2-2 주소로 순차 표현한 메모리 구조

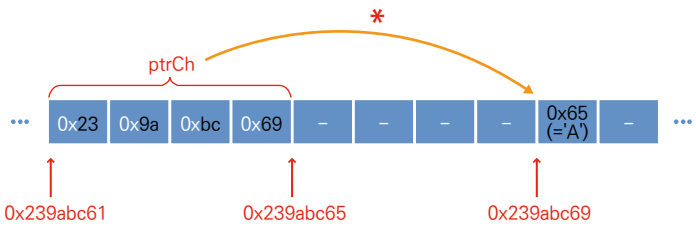


그림에서 볼 수 있듯이 32비트 운영체제에서는 2^{32}바이트(약 4GB)의 메모리만 관리할 수 있다. 그래서 일반 컴퓨터에서 4GB 이상의 메모리를 사용하려면 64비트 운영체제를 사용해야 한다. 우리가 만드는 프로그램은 메모리의 첫 부분부터 사용하는 것이 아니라 운영체제가 메모리의 중간 부분을 사용하도록 허가하는 것이다.

즉, 프로그래머가 만든 프로그램은 운영체제가 허가한 메모리 안에서만 동작하고, 포인터는 말 그대로 '가리키는 것'으로 메모리를 가리키며, 이 메모리의 주소를 가지고 동작한다고 보면 된다.

어떤 메모리 공간에 데이터가 존재한다면 포인터는 데이터가 존재하는 메모리의 주소를 가리킨다. 즉, 어떤 변수가 포인터 변수라면 해당 변수는 메모리의 주소값만을 가지는 것이며, 32비트(4바이트)의 공간만을 가진다. 즉, 모든 포인터 변수의 크기는 4바이트다. 예를 들어 char *ptrCh = 'A';라고 입력한 코드는 그림 2-3과 같은 메모리 구조를 가지게 된다.

그림 2-3 포인터 변수의 메모리 구조



포인터는 변수 앞에 '*'(애스터리스크)를 입력해 선언하며 '가리킨다'는 의미로 해석하면 이해하기 쉽다. 위 그림에서 볼 수 있듯이 ptrCh 변수는 4바이트의 메모리 공간에 주소값 0x239abc69를 가진다. 그런데 지시자인 '*'가 ptrCh 변수 앞에 붙으면 ptrCh 변수 안에 저장한 주소를 읽은 후 메모리 안의 내용을 가져온다.

따라서 포인터를 이해한다는 것은 위 *ptrCh라는 포인터 변수와 포인터가 가리키는 실제 변수 ptrCh를 구분할 줄 안다는 것이다. 즉, char *ptrCh는 ptrCh 변수 크기가 char 자료형의 크기인 1바이트가 아니라 주소값을 저장하는 4바이트고, *ptrCh(즉, 포인터가 가리키는 대상)의 자료형이 char라는 의미다.

포인터 기호 '*'와 항상 같이 사용하는 것은 메모리 주소를 나타내는 '&'(앰퍼샌드)다. 이 기호를 변수 앞에 붙이면 주소값을 가져오라는 의미다. 다음 코드를 살펴보면 알 수 있다.

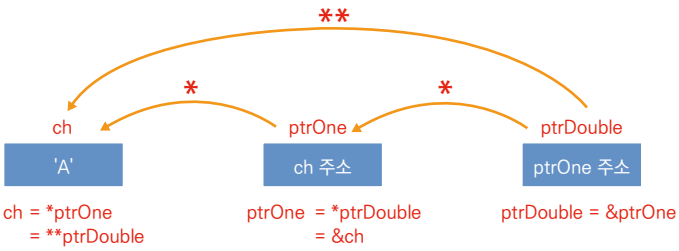
코드 2-1 '&'의 사용 예

```
char ch, *ptrCh;  // 문자를 저장하는 변수 'ch'와 포인터 변수 'ptrCh' 선언
ch = 'A';         // 'ch' 변수에 'A'라는 문자를 삽입
ptrCh = &ch;      // 'ptrCh' 변수에 'ch'의 주소값을 삽입
```

위와 같이 '&'를 이용해 변수의 주소값을 포인터 변수에 대입하면 포인터 변수는 ch를 가리킨다. 즉, ptrCh = &ch; 구문으로 포인터 변수(ptrCh)에 생명을 주었다고 이해하자(ptrCh = &ch; 구문을 선언하기 전의 ptrCh 변수는 임의의 값을 저장하는데, 이는 프로그램에서 아무 쓸모가 없는 데이터다).

이번에는 '*'를 두 번 쓰는 이중 포인터를 알아보기로 하자. 이중 포인터는 말 그대로 두 번 가리켜야 원하는 데이터를 얻는다고 이해하면 된다. 즉, 한 번 가리킨 곳은 주소값을 가지며, 다시 한 번 더 가리킨 곳에 원하는 데이터가 있는 것이다. 이중 포인터는 잘 사용하지는 않지만 포인터를 더 잘 이해하는 데는 많은 도움이 된다. 그림 2-4는 이중 포인터를 어떻게 사용하는지 보여준다.

그림 2-4 이중 포인터



코드 2-2를 보면 이중 포인터를 이해할 수 있다. 코드만 보고 메모리가 어떻게 할당되는지를 생각할 수 있으면 포인터를 충분히 이해했다고 생각해도 좋다.

코드 2-2 이중 포인터의 이해

```
#include <stdio.h>

int main()
{
    char **ptrDouble, *ptrOne, ch = 'A';

    ptrOne = &ch;
    ptrDouble = &ptrOne;

    printf("ptrDouble address : 0x%p \n\n", &ptrDouble);

    printf("ptrDouble value : 0x%p \n", ptrDouble);
    printf("ptrOne address : 0x%p \n\n", &ptrOne);

    printf("ptrOne value : 0x%p \n", ptrOne);
    printf("ch address : 0x%p \n\n", &ch);

    printf("**ptrDouble value: %c \n", **ptrDouble);
    printf(" *ptrOne value: %c \n", *ptrOne);
    printf(" ch value: %c \n\n", ch);
}
```

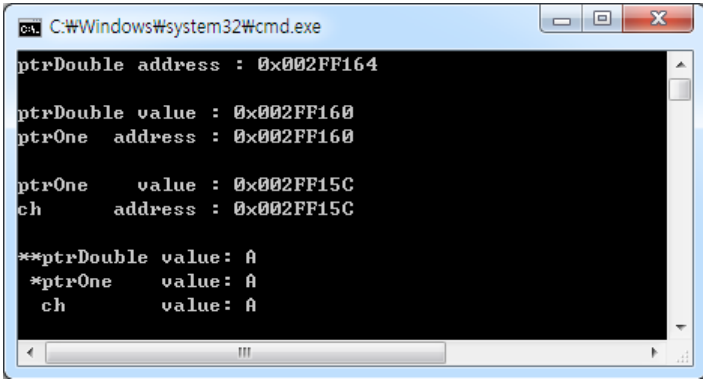
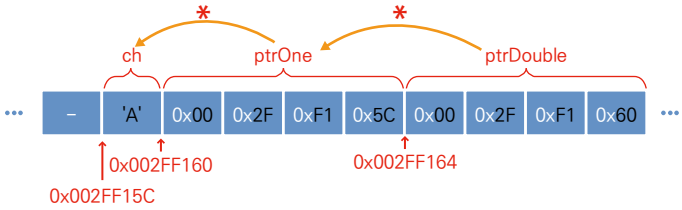


그림 2-5는 실행 결과가 메모리에서 어떻게 구조화되는지를 보여준다. 단일 포인터(ptrOne 변수)와 이중 포인터(ptrDouble 변수)는 주소값을 가진 단순 변수임을 알 수 있다. 그리고 포인터 지시자인 '*'에 의해 다른 곳을 찾아간다.

그림 2-5 코드 2-2 실행 결과에 따른 메모리 구조



정리하면 포인터가 메모리 주소를 저장하는 변수라고 이해하는 순간, 메모리 구조를 이해할 수 있으며 포인터를 자유자재로 사용할 수 있다. 우리가 흔히 접하는 '참조에 의한 호출Call-By-Reference'은 대상을 메모리에 그대로 두고 주소값을 전달하여 대상을 제어한다고 이해하면 된다. 사실 포인터를 이해하고 나면 다른 프로그래밍 언어가 재미없어질 것이다.

포인터는 Big Number 연산을 이해하는 데 가장 중요한 내용이다. 즉, 큰 수를 만들게 되면 메모리에 수를 저장하고 Big Number는 포인터로 메모리의 주소만을 기억하게 된다. 그리고 이 메모리 주소를 부르는 방식으로 연산을 실행해 실제 처리해야 하는 데이터양을 줄여 연산 속도를 빠르게 만들기도 한다.

2.2 구조체

일반 C 입문서에서는 구조체structure를 비중 있게 설명하지 않는다. 생각보다 간단하기 때문이다. 그러나 실무에서 프로그래밍을 하다 보면 구조체는 상당히 중요하며 아주 많이 사용된다는 것을 알게 된다. 또한 Big Number의 모든 수는 구조체로 표현하는데, 이때 큰 수 하나가 하나의 구조체라고 생각하면 이해하기 쉽다.

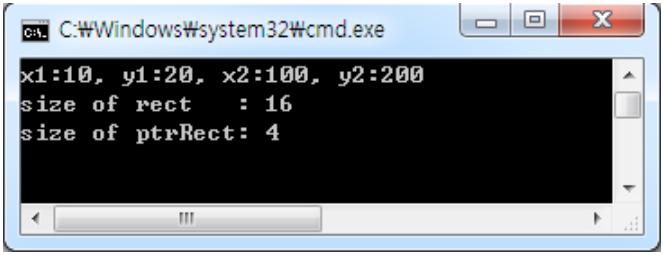
어떤 프로그램을 이해한다는 것은 소스 코드에 포함된 구조체를 완벽히 이해하는 것이다. 구조체는 프로그래머가 만들 수 있는 새로운 자료형이며, 구조체 개념이 발전해 클래스class가 되었다. 즉, 객체 지향 프로그래밍Object-Oriented Programming, OOP이라는 개념은 구조체를 기반으로 만들어졌다고 말할 수 있다.

> **NOTE** 여러분이 어떤 회사에 입사한다면 책에서 보던 기본 자료형(int, char, float, bool 등)이 없는 소스 코드를 볼 수도 있다. 그렇다고 난감해하지 말자. 구조체가 그 회사에 특화되도록 소스 코드를 바꾸어 놓은 것이다. 즉, 회사에서 구조체로 새로운 자료형을 만들어서 사용하는 것이며, 어떤 구조체가 만들어졌는지를 파악하고 나면, 소스 코드를 분석하는 데 큰 문제가 없을 것이다.

이 책에서 설명하는 구조체는 기존의 자료형에 기반을 두고 새로운 자료형을 만드는 것이라고 이해하자. 코드 2-3은 정수형(int)이 모여 점(Point)이 되고, 점(Point)가 모여 사각형(Rectangle)이 되는 구조체의 예제다.

코드 2-3 구조체 예제 - Point & Rect

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

struct Rect {
    Point ptLeftTop;
    Point ptRightBottom;
} rect, *ptrRect; // --- ❶

int main()
{
    // --- ❷
    rect.ptLeftTop.x = 10;
    rect.ptLeftTop.y = 20;
    rect.ptRightBottom.x = 100;
    rect.ptRightBottom.y = 200;

    ptrRect = &rect;

    printf("x1:%d, y1:%d, x2:%d, y2:%d \n",
        // --- ❸
        ptrRect->ptLeftTop.x, ptrRect->ptLeftTop.y,
        ptrRect->ptRightBottom.x, ptrRect->ptRightBottom.y);

    // --- ❹
    printf("size of rect   : %d \n", sizeof(rect));
    printf("size of ptrRect: %d \n", sizeof(ptrRect));
}
```



소스 코드를 살펴보면 다음과 같다.

❶에서는 구조체 변수를 선언했다. rect는 변수로서 메모리에 구조체 공간의 크기(16바이트)를 할당하며, ptrRect는 구조체를 가리키는 포인터 변수로서 메모리의 주소값을 저장하는 4바이트의 공간만 할당한다. 이처럼 구조체를 정의하고 변수를 바로 선언하는 코드는 생각보다 많이 사용한다.

❷에서는 구조체 안에 있는 변수 x와 y에 접근하려고 '.' 기호를 사용한다.

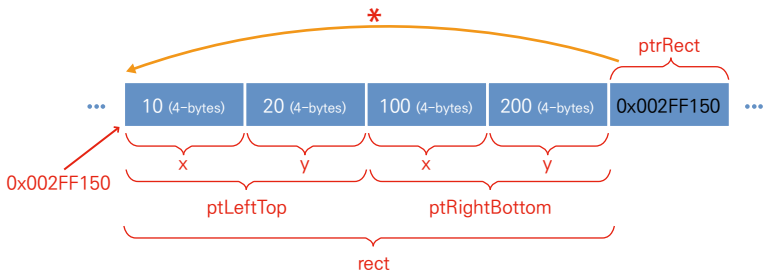
❸에서는 구조체를 가리키는 포인터 변수인 prtRect에서 구조체가 존재하는 메모리로 접근하려고 화살표 모양인 '->' 기호를 사용한다.

❹에서는 rect 변수와 ptrRect 포인터 변수의 크기를 출력한다.

코드 2-3에서 중요한 점은 구조체가 메모리에 어떻게 할당되는지 아는 것이다. 위 소스 코드에 있는 rect와 ptrRect 변수는 그림 2-6처럼 메모리에 할당한다.

또한 rect와 ptrRect 변수는 전역 변수이므로 코드 2-3을 실행하면 메모리의 스택 Stack에 할당해 프로그램 실행이 끝날 때까지 존재하게 된다.

그림 2-6 구조체 변수 rect와 구조체 포인터 변수 ptrRect의 메모리 할당



구조체를 메모리에 할당할 때는 구조체 안에 존재하는 변수의 순서가 중요하다. 구조체로 자료형을 변환해야 하는 경우가 발생하기 때문이다.

2.3 malloc( ) 함수와 free( ) 함수

컴파일러는 우리가 만든 소스 코드를 기계어로 바꾼 후 실행 파일로 만든다. 실행 파일을 실행하면 운영체제는 알아서 메모리를 할당하고 그 안에서 프로그램이 동작한다. 이때 운영체제에서 사용을 허가한 메모리는 프로그램 하나를 위해 여러 개로 나눈다.

나누는 영역은 크게 보면 코드가 존재하는 영역과 프로그램이 동작하는 동안 데이터가 기록되고 지워지는 영역이다. 코드가 존재하는 영역은 프로그램을 실행하는 동안은 변화가 없으나 변수 등의 데이터 영역을 위해 사용하는 영역은 일반적으로 스택Stack과 힙Heap이라는 두 부분으로 나누어지며 프로그램이 실행되는 동안 크기가 수시로 변한다. 예를 들어 스택은 프로그램에서 함수를 실행하면 함수 안에 존재하는 변수값을 저장하고, 힙은 프로그래머가 인위적으로 메모리를 할당하여 사용한다(사실 스택과 힙 외에도 전역 변수(또는 정적 변수static variable)를 저장하는 '데이터'라는 영역도 있으나 이해를 쉽게 하기 위해 생략했다).

그림 2-7은 프로그램을 실행할 때 메모리 영역이 어떻게 나누어지는지를 보여준다.

그림 2-7 프로그램이 사용하는 메모리 영역의 구조

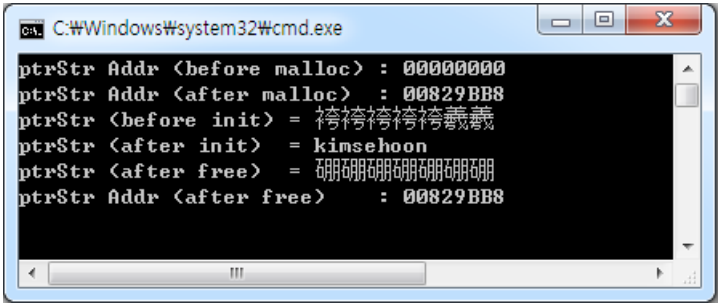
앞으로 설명하게 될 malloc() 함수는 힙 영역을 사용하는 것이며, 메모리 용량이 허락하는 한 원하는 만큼 프로그래머가 사용할 수 있다.

'메모리 할당'을 영어로는 'Memory Allocation'이라고 표현한다. 그래서 메모리를 할당한다는 뜻으로 함수 이름을 malloc이라고 지었다. 그리고 할당한 메모리를 없애주는 함수는 '자유롭게 놔준다'는 의미로 free라고 이름 지었다.

코드 2-4 malloc() 함수와 free() 함수

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;

    char str[10] = "kimsehoon";
    char *ptrStr;

    printf("ptrStr Addr (before malloc) : %p\n", ptrStr); // --- ❶
```

```
    ptrStr = (char *)malloc(10); // --- ❷

    printf("ptrStr Addr (after malloc)  : %p\n", ptrStr);

    printf("ptrStr (before init) = %s \n", ptrStr); // --- ❸

    // --- ❹
    for(i = 0; i < 10; i++)
        ptrStr[i] = str[i];

    printf("ptrStr (after init) = %s \n", ptrStr);

    free(ptrStr); // --- ❺

    printf("ptrStr (after free) = %s \n", ptrStr);
    printf("ptrStr Addr (after free) : %p\n", ptrStr); // --- ❻
}
```



소스 코드의 중요 부분을 살펴보자.

❶에서의 ptrStr 포인터 변수는 malloc() 함수 사용 전에는 가리키는 곳이 없으므로 임의의 값으로 초기화되었다.

❷에서는 힙 영역에 10바이트를 할당한다. malloc() 함수의 실행 결과는 항상 포인터 변수에 저장하며, (char *)등과 같이 자료형 변환을 해주어야 한다. 따라서 ptrStr 포인터 변수는 할당한 메모리의 제일 앞 번지수를 저장하게 된다.

❸에서는 malloc() 함수로 메모리 영역을 할당받았지만 어떠한 값도 입력하지 않았기에 쓰레기값Garbage Value으로 초기화되었다.

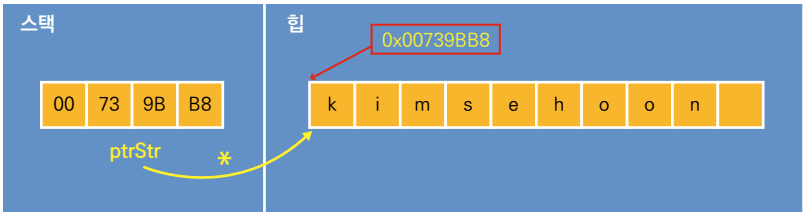
❹에서는 malloc() 함수로 할당받은 힙 영역에 데이터를 삽입한다.

❺에서는 malloc() 함수로 할당받았던 메모리 영역을 free() 함수로 해제한다. 그래서 ptrStr 변수가 가리키는 곳은 다시 쓰레기값을 가진다.

❻에서는 free() 함수로 힙 영역을 삭제해도 ptrStr 변수값에는 변화가 없으므로 힙 영역을 계속 가리킨다.

그림 2-8은 코드 2-4의 실행 결과에 따른 메모리 구조를 표현한 것이다. 실제 프로그램에서 스택과 힙 두 영역을 함께 사용함을 알 수 있다.

그림 2-8 malloc() 함수에 의한 ptrStr 지역 변수의 메모리 구조



지역 변수인 ptrStr은 스택 영역에 저장했으며, malloc() 함수로 할당한 메모리 공간은 힙 영역에 자리한다. ptrStr 변수는 힙 영역에 할당한 메모리 공간을 가리키는 주소값만을 저장한다.

프로그래밍하다 보면 malloc() 함수를 많이 사용하게 되는데, 힙 영역에 할당한 메모리 공간을 free() 함수로 해제하지 않으면 프로그램이 사용하는 메모리가 계속 커지게 된다. 코드가 길면 프로그래머의 실수로 free() 함수를 생략하는 경우도 발생하는데, 처음에는 프로그램에 문제가 생기지 않기 때문에 무심코 지나쳤다가 나중에 어려움을 겪을 수도 있다. 이러한 이유로 메모리 할당은 종종 어렵게 느껴지곤 한다.

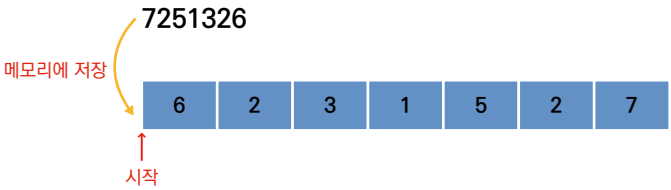
Big Number에서 다루는 수는 크기가 정해진 것이 아니므로 수를 저장하는 메모리 크기가 일정하지 않다. 따라서 수를 생성할 때 malloc() 함수로 수를 저장하는 메모리를 확보해야 하며, 연산 중에 필요 없는 수는 free() 함수로 메모리의 공간을 삭제해야 한다.

2.4 구조체를 이용한 새로운 자료형

이제부터 본격적으로 Big Number를 살펴보겠다. 먼저 구조체로 새로운 자료형을 만드는 원리를 살펴보자.

정수의 크기를 판단하는 기준은 오른쪽이다. 즉, 7251326이라는 숫자는 '$7\times10^6 + 2\times10^5 + 5\times10^4 + 1\times10^3 + 3\times10^2 + 2\times10^1 + 6\times10^0$'으로 표현하며, 오른쪽으로부터 얼마만큼 떨어져 있느냐가 해당 숫자의 크기를 결정한다. 그렇다면 메모리 관점에서는 어떻게 표현할까? 메모리는 값을 저장할 위치가 결정되면 오른쪽으로 증가한다. 즉, 숫자를 메모리에서 표현하려면 숫자 오른쪽 부분을 메모리의 왼쪽 부분에 위치시킨다는 뜻이다. 그리고 수학에서는 소수점이 중요한데, 고정된 소수점의 위치를 프로그램상에서 메모리의 시작 위치로 생각하면 된다. 실제 프로그래밍 언어에서 이를 고려할 필요는 없지만, 우리가 만드는 새로운 자료형은 그림 2-9와 같이 숫자를 표현한다고 기억하자.

그림 2-9 숫자를 메모리상에 표현



위 그림과 같이 10진수의 한 자리 숫자를 메모리상 1바이트만큼의 공간에 입력한다면 10진수의 표현에 따라서 메모리의 크기도 정해질 것이다. 1바이트에서 표현할 수 있는 숫자는 0~255일 테지만, 여기서는 1바이트에서 표현하는 숫자를 0~10까지로 한정한다고 가정하자. 새로운 자료형을 만들 때의 장점은 프로그래머 마음대로 모든 것을 정의할 수 있다는 점이다. 따라서 그림 2-9는 코드 2-5와 같이 선언할 수 있다.

코드 2-5 숫자 7251326을 새로운 자료형으로 표현

```
unsigned char decimal[7] = { 0x06, 0x02, 0x03, 0x01, 0x05, 0x02, 0x07 };
```

그렇다면 Big Number 연산을 위한 새로운 자료형은 어떻게 정의해야 할까? 정수의 특징에 따라야 한다. 먼저 음수인지 양수인지를 표시하는 sign 변수가 필요하고, 값을 저장하고 있는 메모리 공간도 필요하다. 그런데 값을 저장하는 공간은 숫자의 크기에 따라 크기가 유동적일 것이다.

그렇다면 메모리상에서 숫자를 담는 공간을 어떻게 표시해야 할까? 해결 방법은 메모리 공간을 할당할 때 시작하는 곳을 알고, 크기 정보를 가지면 된다. 그래서 새로운 자료형인 BIG_NUMBER라는 구조체는 코드 2-6과 같이 선언한다.

코드 2-6 정수 연산을 위한 BIG_NUMBER 구조체

```
struct BIG_NUMBER {
    unsigned char *ptrSpace;  // ptrSpace는 저장 공간의 시작 번지를 저장
    int size;                 // 저장 공간의 크기(바이트 크기)를 저장
    bool sign;                // 부호 변수(false:양수, true:음수)
};
```

코드 2-6에서는 정수만을 다룬다. 그렇다면 실수를 위한 새로운 자료형은 어떻게 만들까? 소수점의 위치를 저장하는 변수를 추가하면 된다. 프로그래밍에서의 실수 연산은 정수 연산을 실행한 후 소수점 위치를 바꿔주면 된다. 실수 연산을 위한 구조체는 코드 2-7이며 소수점 위치를 저장하는 변수만 추가했다.

코드 2-7 실수 연산을 위한 BIG_FLOAT 구조체

```
struct BIG_FLOAT {
    unsigned char *ptrSpace;  // ptrSpace는 저장 공간의 시작 번지를 저장
    int size;                 // 저장 공간의 크기(바이트 크기)를 저장
    int decimalPoint;         // 소수점 위치를 저장
    bool sign;                // 부호 변수(false:양수, true:음수)
};
```

Big Number 연산에서 실수는 다루지 않을 것이다. 하지만 Big Number 연산을 이해한다면 실수 연산을 구현하는 일도 어렵지 않을 것이다.

2.5 BIG_DECIMAL 구조체

이번에는 앞서 배운 개념을 살려 실제로 이 책에서 사용할 구조체를 만들어보자. 여기서는 Big Number를 10진수로 표현하려고 BIG_DECIMAL이라는 구조체를 정의한다.

이 구조체는 Big Number에 해당하는 모든 수를 표현하게 되는데, 실제 수는 메모리에 저장하며 구조체 안의 포인터가 실제 수를 가리킨다. 구조체의 구현은 코드 2-8과 같다.

코드 2-8 BIG_DECIMAL 구조체

```
struct BIG_DECIMAL { // --- ❶
    unsigned char *digit; // --- ❷
    int size; // --- ❸
    bool sign; // --- ❹
};
```

❶에서는 BIG_DECIMAL이라는 구조체를 정의한다.

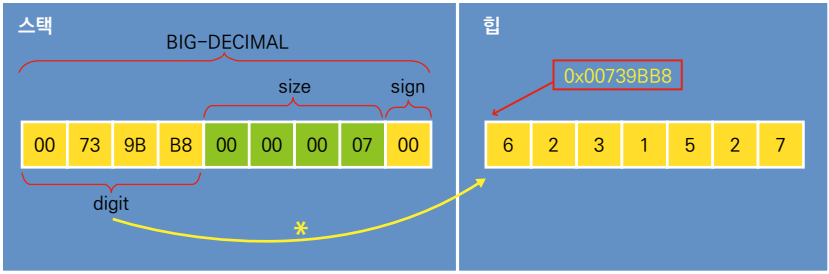
❷에서 digit은 malloc() 함수로 힙 영역에 할당한 주소의 시작 번지를 가지는 포인터 변수며, malloc() 함수에 할당한 공간에 실제 10진수 값을 저장한다. 10진수의 각 자릿수에 해당하는 digit이라는 포인터 변수 하나는 1바이트 크기를 저장하므로 char로 선언했다.

❸에서 size 변수는 digit 포인터 변수가 가리키는 힙 영역에 할당한 메모리 공간의 바이트 크기를 저장한다. 즉, 10진수 숫자의 총 개수라고 보면 된다.

❹에서 sign 변수는 정수 부호를 나타내며 음수는 true, 양수는 false로 표현한다.

BIG_DECIMAL 구조체는 그림 2-10과 같이 스택과 힙 영역 모두에 값을 가지므로, BIG_DECIMAL 구조체의 크기가 항상 9바이트더라도 힙 영역에 할당한 공간을 고려한다면 실제 크기는 유동적이라고 보는 것이 좋다. 그림 2-10은 7251326이라는 10진수가 BIG_DECIMAL 구조체에서 어떻게 표현되는지를 보여준다.

그림 2-10 메모리상에서의 BIG_DECIMAL 구조체



위 그림에서 BIG_DECIMAL 구조체는 스택 영역에 있는 9바이트 크기의 메모리에 저장한다.

처음 4바이트는 포인터 변수인 digit에 해당하며, 실제 값을 저장한 힙 영역의 주소를 저장한다. 구조체의 두 번째 변수인 size는 힙 영역에 할당한 메모리 공간의 크기를 저장한다. 즉, 위 그림에서 size 변수값이 7이면 힙 영역에서 7바이트의 공간을 사용한다고 보면 된다. 마지막으로 sign 변수값이 0이면 BIG_DECIMAL 구조체로 표현하는 숫자가 양수임을 나타낸다.

코드 2-9는 BIG_DECIMAL 구조체로 실제 값을 생성하는 CreateDecimal() 함수로, 문자열과 크기를 매개변수로 받아서 BIG_DECIMAL 구조체로 표현하는 양의 정수를 만든다.

코드 2-9 BIG_DECIMAL 구조체를 이용해 실제 값을 생성하는 CreateDecimal() 함수

```
BIG_DECIMAL CreateDecimal(unsigned char *str, int size) // --- ❶
{
    BIG_DECIMAL decimal;

    decimal.digit = (unsigned char *)malloc(size); // --- ❷
```

```
    // --- ❸
    for(int i = 0; i < size; i++)
        decimal.digit[i] = str[size-i-1] - 48;

    // --- ❹
    decimal.size = size;
    decimal.sign = false;

    return decimal; // --- ❺
}
```

❶에서는 BIG_DECIMAL 구조체를 이용하는 함수를 선언하며, 매개변수로 문자열과 10진수의 크기를 입력받는다. 함수의 사용법은 CreateDeciama((unsigned char*)"7251326", 7)처럼 매개변수로 10진수 문자열(7251326)을 저장하고, 두 번째로 문자열의 길이(7)를 저장한다.

❷에서는 BIG_DECIMAL 구조체를 이용해 실제 값을 저장하려는 공간을 malloc() 함수로 할당한다.

❸에서는 ❷에서 할당받은 공간에 매개변수로 입력받은 값을 삽입한다. for문을 통해 매개변수 문자열의 오른쪽 값은 할당받은 공간의 왼쪽부터 채워지게 된다. 48을 빼는 이유는 '0'이라는 문자의 아스키코드 값이 48이기 때문이다(문자로 받는 값을 실제 값으로 바꿔줄 때는 아스키코드 값을 가지고 처리한다). 즉, 0~9라는 문자열의 실제 값이 48~57이므로 48을 빼주어 실제 값을 0~9로 만들어주는 것이다(이해가 안 되는 분은 'APPENDIX A. 아스키코드' 부분을 참고하기 바란다).

❹에서는 BIG_DECIMAL 구조체의 size 변수와 sign 변수의 값을 입력한다.

❺에서는 반환하는 값이 구조체므로 메모리 관점에서 본다면 BIG_DECIMAL 구조체 크기인 9바이트를 복사한다고 보면 된다. 실제 값을 저장한(malloc() 함수로

할당한) 힙 영역은 지워지지 않고 값을 저장하며, 반환하는 BIG_DECIMAL 구조체 안 힙 영역에 있는 메모리 주소를 저장해 전달하는 것이다.

코드 2-10은 BIG_DECIMAL 구조체로 생성한 decimal이라는 구조체 변수값을 콘솔에 출력하는 함수로, decimal 변수값을 10진수 문자열로 출력한다.

코드 2-10 BIG_DECIMAL 구조체 출력(콘솔)

```
void printDecimal(BIG_DECIMAL decimal) // --- ❶
{
    int i;

    // --- ❷
    if(decimal.sign)
        printf("-");

    // --- ❸
    for(i = decimal.size-1; i >= 0; i--)
        printf("%c", decimal.digit[i]+48);
    printf("\n");
}
```

❶의 printDecimal() 함수는 BIG_DECIMAL 구조체 변수 decimal의 값을 콘솔에 출력한다. 이러한 출력 함수는 초기에 꼭 만들어야 한다. 왜냐하면 테스트 목적으로 가장 많이 사용하기 때문이다. 보통 테스트 코드 작성에 다소 인색한 편인데, 실제 프로그래밍에서는 테스트 코드를 많이 작성할수록 좋다.

❷에서는 decimal 변수의 부호를 출력한다. sign 변수값이 1이면 음수고, 0이면 양수다.

❸에서는 10진수의 왼쪽부터 차례대로 출력한다. 48을 더해주는 이유는 실제 값

을 우리가 알아볼 수 있는 문자열로 출력하기 위해서다. 즉, 실제 값인 0~9에 48을 더해 48~57의 값을 만들면 숫자를 나타내는 아스키 문자로 출력된다.

decimal 변수값의 크기가 클 때는 파일에 출력할 필요가 있다. 코드 2-11은 콘솔 대신 파일에 출력하려는 값을 기록하는 함수로, 콘솔 출력에 사용된 printf() 함수를 파일로 출력하는 fprintf() 함수로 대체한 것 외에는 소스 코드 내용이 같다.

코드 2-11 BIG_DECIMAL 구조체 출력(파일)

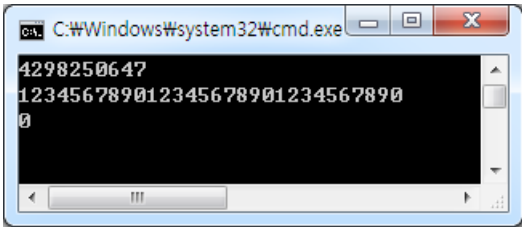
```
void fprintDecimal(FILE *fp, BIG_DECIMAL decimal)
{
    int i;

    if(decimal.sign)
        fprintf(fp, "-");
    for(i = decimal.size-1; i >= 0; i--)
        fprintf(fp, "%c", decimal.digit[i]+48);
    fprintf(fp, "\n");
}
```

코드 2-12는 decimal 구조체 변수값을 출력하는 테스트 코드다. 실제 프로그램을 구현할 때는 위에서 설명한 소스 코드보다 테스트 코드를 먼저 작성하는 것이 좋다. 왜냐하면 프로그램을 다 만든 후에 수정하는 것보다는 문제가 있는지를 테스트하면서 프로그램을 만들어 가는 방법이 좋기 때문이다.

코드 2-12 BIG_DECIMAL 구조체 테스트

```
int main()
{
    BIG_DECIMAL decimal;
```

```
    decimal = CreateDecimal((unsigned char *)"4298250647",10); // --- ❶
    printDecimal(decimal);

    decimal = CreateDecimal(
        (unsigned char *)"12345678901234567890123456789O",30
    ); // --- ❶
    printDecimal(decimal);

    decimal = CreateDecimal((unsigned char *)"0",1); // --- ❶
    printDecimal(decimal);

    // --- ❷
    FILE *fp;
    if((fp=fopen("result.txt", "wt")) == NULL)
        printf("file open error. \n");

    fprintDecimal(fp, decimal);
        fclose(fp);
}
```



❶에서는 BIG_DECIMAL 구조체를 이용하는 CreateDecimal() 함수를 사용한다. 10진수의 문자열을 넣고 자료형 변환을 해주며 숫자 길이를 입력한다.

❷에서는 출력하려는 값을 저장할 파일을 생성한다.

NOTE BIG_DECIMAL 구조체를 이해했다면 여러분도 Big Number 연산을 직접 구현할 수 있다. 필자는 이해를 돕고 구현을 쉽게 하려고 10진수의 각 자릿수에 해당하는 digit이라는 포인터 변수 각각을 1바이트에서 표현했으나 다른 구조체를 만들어 Big Number를 표현할 수도 있으며 어디까지나 프로그래머가 마음대로 정할 문제다. 즉, 이 책에 소개한 것이 정답은 아니라는 뜻이다.

프로그래밍은 소스 코드를 작성하는 일만 말하는 것이 아니다. 소스 코드 작성은 극히 일부분에 해당하는 작업일 뿐이며, 어떻게 구조화할지 생각하는데 소스 코드 작성 시간의 몇 배 혹은 몇십 배를 소요할 수도 있다. 따라서 어떻게 연산을 실행할지를 스스로 생각해보는 것도 좋은 훈련이다.

여러분이 어떤 회사에 입사한다면 책에서 보던 기본 자료형(int, char, float, bool 등)이 없는 소스 코드를 볼 수도 있다. 그렇다고 난감해하지 말자. 구조체가 그 회사에 특화되도록 소스 코드를 바꾸어 놓은 것이다. 즉, 회사에서 구조체로 새로운 자료형을 만들어서 사용하는 것이며, 어떤 구조체가 만들어졌는지를 파악하고 나면, 소스 코드를 분석하는 데 큰 문제가 없을 것이다.

2.6 BIG_BINARY 구조체

여기서 구현하려는 Big Number는 어쩌면 2진수로 표현할 필요가 없을지도 모른다. 하지만 빠른 지수 연산exponential calculation을 하려면 2진수를 이용해야 하므로 2진수를 표현하는 BIG_BINARY라는 구조체를 만들었다.

NOTE 현대 암호학에서는 2진수만 사용한다고 생각할 수도 있다. 왜냐하면 컴퓨터가 사용하는 기계어란 결국 2진수며 현대 암호학은 반드시 컴퓨터를 이용해야 하기 때문이다(컴퓨터가 처리하는 모든 연산은 사실 2진수 계산이다).

5장에서 다룰 지수 연산에서 2진수가 얼마나 유용하게 사용되는지를 알면 놀랄 것이다(필자는 개인적으로 인수분해를 하지 않고도 빠른 지수 연산을 할 수 있음을 알고 난 후에 2진수를 사랑하게 되었다).

코드 2-13은 Big Number를 2진수로 표현하기 위해서 BIG_BINARY라는 구조체를 구현했다.

코드 2-13 BIG_BINARY 구조체

```
struct BIG_BINARY { // --- ❶
    unsigned char *byte; // --- ❷
    int size; // --- ❸
};
```

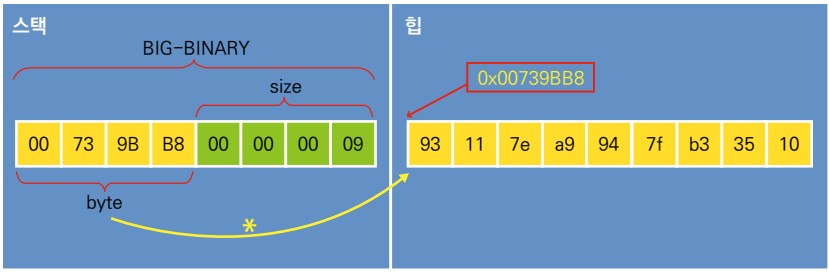
❶에서는 BIG_BINARY 구조체를 정의한다.

❷의 byte 변수는 malloc() 함수로 힙 영역에 할당한 주소의 시작 번지를 갖는 포인터 변수며, malloc() 함수로 할당한 공간에는 2진수 값을 저장한다.

❸의 size 변수는 byte 변수가 가리키는 공간의 크기를 저장한다.

2진수는 부호가 필요하지 않기에 부호를 정의하는 구문은 구조체에 포함하지 않았으며, 10진수 Big Number는 2진수 Big Number로 변환할 수 있다. BIG_DECIMAL 구조체와 마찬가지로 BIG_BINARY 구조체는 그림 2-11과 같이 스택과 힙 영역에 데이터를 가지며, 스택 영역에 있는 BIG_BINARY 구조체의 크기가 항상 8바이트라도 힙 영역에 할당한 공간은 숫자 크기에 따라서 유동적인 크기를 가진다. 또한 그림 2-11은 정수 29901748196066971278 7(0x1035b37f94a97e1193)을 갖는 BIG_BINARY 구조체의 메모리 구조를 표현하는 것이기도 하다.

그림 2-11 BIG_BINARY 구조체의 메모리 구조



위 그림에서 BIG_BINARY 구조체는 스택 영역에 있는 8바이트의 메모리에 저장한다. 처음 4바이트는 byte 포인터 변수의 영역이며, 실제 값을 저장한 힙 영역의 주소를 저장한다. 구조체의 두 번째 변수인 size는 힙 영역에 할당한 공간의 크기를 저장한다. 즉, 위 그림에서 size 변수값이 9면 힙 영역에서 9바이트의 공간을 사용한다고 보면 된다. 힙 영역에 저장한 값은 실제 2진수 데이터인데, BIG_DECIMAL 구조체와 마찬가지로 왼쪽을 기준으로 오른쪽으로 증가하는 구조며, 실제 값의 역순으로 저장한다(조금 자세히 설명하자면 자바의 정수형(int)은 다른 프로그래밍 언어와 달리 역순으로 값을 저장한다).

코드 2-14는 BIG_BINARY 구조체로 실제 값을 생성하고 출력하는 함수들이다. CreateBinary() 함수는 2진수 데이터와 데이터의 바이트 크기를 매개변수로 받아서 BIG_BINARY 구조체로 실제 값을 생성하는데, 실제 잘 사용하지는 않는다. 왜냐하면 CreateBinary() 함수를 사용하는 것보다 BIG_DECIMAL 구조체를 BIG_BINARY 구조체로 변환Convert해서 사용하기가 쉽고 효과적이기 때문이다.

필자도 BIG_BINARY 구조체로 직접 실제 값을 생성한 적은 거의 없고, 항상 BIG_DECIMAL 구조체를 이용해 실제 값을 생성한 후 BIG_BINARY 구조체로 바꾸어서 연산에 응용했다. BIG_DECIMAL 구조체를 BIG_BINARY 구조체로 변

환하는 일은 Big Number 연산의 사칙 연산을 구현한 후에 만들어야 하므로 '5.1 진수 변환'에서 다룰 것이다.

코드 2-14 BIG_BINARY 구조체를 이용해 실제 값을 생성하는 CreateBinary() 함수

```
BIG_BINARY CreateBinary(unsigned char *bytes, int length) // --- ❶
{
    BIG_BINARY binary;

    binary.byte = (unsigned char *)malloc(length); // --- ❷

    // --- ❸
    for(int i = 0; i < length; i++)
        binary.byte[i] = bytes[length-i-1];

    binary.size = length; // --- ❹

    return binary; // --- ❺
}
```

❶에서는 BIG_BINARY 구조체를 이용해서 실제 값을 생성하는 CreateBinary() 함수를 선언하며, 매개변수로 bytes라는 포인터 변수와 bytes 변수의 개수인 정수형 length 변수를 입력받는다.

❷에서는 BIG_BINARY 구조체를 이용해서 생성한 binary 변수값을 저장할 공간을 malloc() 함수로 할당한다.

❸에서는 ❷에서 할당한 공간에 매개변수로 입력받은 데이터를 삽입한다. for문에서는 매개변수로 받은 bytes 변수의 오른쪽 값을 할당받은 공간의 왼쪽부터 채운다.

❹에서는 BIG_BINARY 구조체의 size 변수값을 입력한다.

❺에서 반환하는 값이 구조체이므로 메모리 관점에서 본다면 BIG_BINARY 구조체 크기인 8바이트를 복사한다고 보면 된다. 실제 값을 저장한(malloc() 함수로 할당한) 힙 영역은 지워지지 않고 값을 저장하며, 반환하는 BIG_DECIMAL 구조체 안 힙 영역에 있는 메모리 주소를 저장해 넘기는 것이다.

코드 2-15의 printBinary() 함수는 콘솔에 BIG_BINARY 구조체로 생성한 실제 값을 출력하는 함수다. 값을 저장하는 bytes 변수 개수가 N개라면 1바이트에 8비트가 존재하므로 출력하는 2진수 문자 개수는 '8 × N'개다.

코드 2-15 BIG_BINARY 구조체 출력(콘솔)

```
void printBinary(BIG_BINARY binary) // --- ❶
{
    int i, j;
    unsigned char flag;

    // --- ❷
    for(i = binary.size-1; i >= 0; i--)
    {
        flag = 0x80;

        for(j = 0; j < 8; j++)
        {
            if(binary.byte[i] & flag) // --- ❸
                printf("1");
            else
                printf("0");

            flag >>= 1; // --- ❹
        }
    }
```

```
    printf("\n");
}
```

❶은 BIG_BINARY 구조체로 생성한 binary 변수값을 콘솔에 출력하는 함수다.

❷에서는 BIG_BINARY 구조체로 생성한 값을 2진수(0 또는 1)로 출력한다. 여기서 사용한 flag 변수는 1바이트 안에서 한 비트만 1이고 나머지는 0이며, 1인 비트 위치에 있는 값을 출력한다. 즉, 처음 flag 변수의 값은 비트를 나타내는 10000000이고, for문을 한 번 실행할 때마다 ❹에서 1이 오른쪽으로 하나씩 이동하므로 다섯 번 실행했다면 flag 변수의 비트값은 00000100이 되고, ❸에서와 같이 AND 연산을 실행해 BIG_BINARY 구조체로 생성한 binary 구조체 변수에서 flag 변수가 가리키는 해당 위치의 비트값을 출력하는 것이다(이 부분을 정확히 이해할 수 없다면 'APPENDIX A. 비트 연산자'를 다시 살펴보기 바란다).

또한 여기서는 2진수 출력을 0과 1만으로 만들었는데, 필요하다면 8진수나 16진수로도 출력하게끔 구현할 수 있다. 위의 함수를 이해했다면 필요한 함수를 쉽게 만들 수 있으리라 생각한다.

코드 2-16의 fprintBinary() 함수는 BIG_BINARY 구조체로 생성한 binary 변수값을 파일로 출력하는 함수인데, 코드 2-11의 printBinary() 함수와 같은 알고리즘을 사용하기에 설명은 생략한다.

코드 2-16 BIG_BINARY 구조체 출력(파일)

```
void fprintBinary(FILE *fp, BIG_BINARY binary)
{
    int i, j;
    unsigned char flag;

    for(i = binary.size-1; i >= 0; i--)
```

```
        {
            flag = 0x80;

            for(j = 0; j < 8; j++)
            {
                if(binary.byte[i] & flag)
                    fprintf(fp, "1");
                else
                    fprintf(fp, "0");

                flag >>= 1;
            }
        }
        fprintf(fp, "\n");
    }
```

코드 2-17은 BIG_BINARY 구조체 변수 binary의 값을 출력하는 테스트 코드로, 입력하는 값이 어떤 형태든 (unsigned char *)를 사용해 자료형 변환을 한다.

코드 2-17 BIG_BINARY 구조체 테스트

```
int main() // --- ❶
{
    BIG_BINARY binary;

    int bytes = -1; // --- ❷
    binary = CreateBinary((unsigned char *) &bytes, 4); // --- ❸
    printBinary(binary);

    char str[9] = { 0x10, 0x35, 0xb3, 0x7f, 0x94, 0xa9, 0x7e, 0x11, 0x93 };
    binary = CreateBinary((unsigned char *) str, 9); // --- ❹
    printBinary(binary);
}
```

❶의 main() 함수는 테스트를 위해서 만들었다.

❷에서 bytes 변수는 제일 왼쪽 비트를 부호 비트로 쓴다. 그래서 -1은 '2의 보수[2] Complement' 법칙에 따라 실제 값은 0xffffffff다. 실제 결과는 위 그림에서 볼 수 있듯이 2진수 값인 11111111111111111111111111111111로 출력한다('2의 보수'에 대한 자료는 인터넷을 검색하면 쉽게 얻을 수 있기에 여기서는 다루지 않는다).

❸과 ❹에서는 매개변수의 자료형이 무엇인지를 아는 일이 중요하지 않다. 왜냐하면 (unsigned char *)를 사용해 자료형 변환을 하기 때문이다. ❸에서는 매개변수가 정수형(int)이고 ❹에서는 매개변수가 문자열(string)이며, 어떠한 자료형을 입력하더라도 이처럼 자료형 변환을 해주면 된다.

NOTE 앞으로 BIG_BINARY 구조체를 많이 사용하지는 않을 것이다. BIG_BINARY 구조체는 사실 이해를 할 수는 있어도, 이를 사용할 때는 굉장히 머리가 복잡해질 수도 있기 때문이다. 하지만 프로그램의 이론적인 부분을 자세히 이해해야 하므로 소개한 것이다.

우리는 프로그래밍을 하는 것이지 펜으로 수학 문제를 푸는 것이 아니다. 그래서 컴퓨터가 인식하는 2진수에 친숙해지는 일은 앞으로의 프로그래밍을 위해 필요하다고 생각한다. 예를 들어 여러분이 네트워크에 관한 프로그래밍을 하게 된다면 2진수에 친숙하다는 것이 얼마나 큰 힘이 되는지를 알게 될 것이다.

2.7 비교 함수

Big Number 연산은 비교 함수(==, 〉=)에서 시작한다. 보통 사칙 연산을 가장 먼저 배우는 것 같지만 수가 '큰지'와 '작은지' 등을 비교하는 능력은 사칙 연산을 배우기 전에 익히게 되는 기본 능력이다. 실제로도 사칙 연산을 하기 전에 비교 연산이 먼저 이루어져야만 사칙 연산을 구현할 수 있다. Big Number 연산에서 구현할 비교 함수 두 가지는 '같은가?(==)'와 '큰가?(〉=)'다. 여러 가지 비교 함수를 만들 수도 있지만 이 두 가지 비교 연산만으로도 나머지 연산을 모두 처리할 수 있다.

첫 번째로 '같은가?(==)'에 해당하는 함수를 살펴보기로 하자. C를 기반으로 하기 때문에 IsEqual() 함수라고 이름 지었으며, C++로 구현하려면 operator 명령어를 사용해 이 함수를 구현하면 된다.

코드 2-18 비교 함수 1 - IsEqual() 함수

```
bool IsEqual(BIG_DECIMAL *A, BIG_DECIMAL *B) // --- ❶
{
    // --- ❷
    if(A->size != B->size)
        return false;

    // --- ❸
    for(int i = 0; i < A->size; i++)
        if(A->digit[i] != B->digit[i])
            return false;

    return true;
}
```

❶에서는 BIG_DECIMAL 구조체 포인터 변수인 A와 B를 매개변수로 입력받아 크

기가 같으면 true를, 다르면 false를 반환하는 IsEqual() 함수를 선언한다. 매개변수는 포인터 변수이므로, BIG_DECIMAL 구조체 크기인 9바이트가 아니라 포인터 크기인 4바이트다. 이처럼 포인터 변수를 사용해서 매개변수를 전달받으면 메모리 사용을 줄일 수 있는 장점이 있다. 실제 프로그램을 만들 때도 구조체 크기가 상당히 클 때는 포인터 변수로 전달할 때가 많다.

❷에서는 두 개의 매개변수에 의해 포인트 된 개체의 멤버인 size 변수(저장 공간의 크기를 저장하는 변수)와 다를 때 false를 반환한다.

❸에서는 두 개의 매개변수에 의해 포인트 된 개체의 멤버인 모든 digit 변수(10진수 각 자릿수의 숫자를 저장하는 변수)를 비교한 다음 하나라도 다르면 false를 반환한다.

다음은 '큰가?(>=)'에 해당하는 함수를 살펴보기로 하자. '>(Bigger Than)'이 아닌 '>=(Is Bigger Than Or Is Equal To)'를 구현하는 함수다. 함수 이름은 IsBigger()인데, C++에서 구현할 경우 이 함수를 operator 명령어로 구현하면 된다.

코드 2-19 비교 함수 2 - IsBigger() 함수

```
bool IsBigger(BIG_DECIMAL *A, BIG_DECIMAL *B) // --- ❶
{
    int i;

    // --- ❷
    if(A->size > B->size)
        return true;
    else if(A->size < B->size)
        return false;
    else
    {
        // --- ❸
```

```
        for(i = A->size-1; i >= 0; i--)
        {
            if(A->digit[i] > B->digit[i])
                return true;
            else if(A->digit[i] < B->digit[i])
                return false;
        }
    }

    return true;
}
```

❶에서는 매개변수로 BIG_DECIMAL 구조체 변수 A와 B를 입력받아 첫 번째 매개변수 A가 두 번째 매개변수 B보다 크거나 같으면 true를 반환하고, 작으면 false를 반환하는 IsBigger() 함수를 선언한다.

❷에서는 두 개의 매개변수에 의해 포인트 된 개체의 멤버인 size 변수만으로 크기를 판단한다.

❸에서는 가장 오른쪽 digit 변수부터 차례로 비교해 크기를 판단한다.

비교 함수의 구현은 간단하다. 특별한 연산을 실행하는 것이 아니라 digit 변수 각각을 비교하므로 이해하기도 쉽다. 앞으로 구현할 연산은 다소 복잡할 수도 있는데 2장에서 소개한 구조체 모두를 이해했다면 생각보다 어렵지는 않을 것이다.

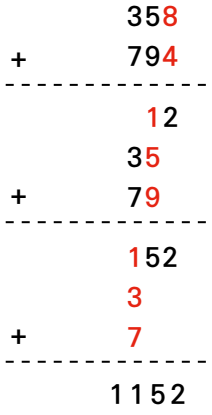
3 | 사칙 연산

Big Number 연산의 사칙 연산 알고리즘은 우리가 아는 사칙 연산을 그대로 따른다. 그러므로 3장에서 다루는 내용은 알고리즘을 코드로 어떻게 구현하는지 이해하는 것이 중요하다. 소스 코드를 살펴보기 전에 어떻게 구현하는 것이 좋은지 스스로 생각해보기 바란다. 여담이지만 다른 사람이 만든 소스 코드를 살펴보기만 하면 자신만의 독창적인 프로그래밍에 방해가 된다. 따라서 직접 구현해보다가 막히는 부분에서 이 책의 소스 코드를 참조하기를 권한다.

3.1 더하기 연산

일반적으로 더하기 연산은 가장 오른쪽 자리부터 하나씩 더하는 것을 반복하고 10 이상일 때 한 자리 높은 숫자에 1을 더하는 것이다. 그림 3-1만 보아도 초등학교 산수 시간의 추억에 잠길 것이다.

그림 3-1 더하기 연산



그럼 BIG_DECIMAL 구조체의 더하기 연산은 어떤 방식으로 이루어질까?

BIG_DECIMAL 구조체에서 가장 낮은 자리의 수는 digit[0]일 것이고, 그 다음은 digit[1]이며, 이를 일반화하면 digit[i] (0 <= i < size)가 될 것이고, i 변수를 for문으로 실행하면 될 것이다.

따라서 BIG_DECIMAL 구조체에서의 더하기 연산은 result = PLUS(A, B)처럼 표현할 수 있는데, 이 연산 안에서는 코드 3-1과 같은 소스 코드를 실행할 것이다 (temp 변수는 i-1 번째 연산에서 받는 값으로 i-1 번째 값이 10 이상이면 1이 주어지고, 10 이하면 0을 temp 변수에 저장한다).

코드 3-1 BIG_DECIMAL 구조체의 일반화 구문

```
// if 0<=i<n, temp = 0 or 1
result.digit[i] = (A.digit[i] + B.digit[i] + temp) % 10
```

BIG_DECIMAL 구조체 변수 A와 B를 더할 때 중요한 것은 두 개 변수값의 크기 (size 변수값)다. 즉, 코드로 구현하려면 digit 변수값의 크기가 큰 것과 작은 것을 구분해주어야 한다는 뜻이다. 따라서 먼저 크기가 작은 것을 기준으로 더하기 연산을 실행하고, 큰 수의 나머지 부분으로 더하기 연산을 실행해야 한다.

그림 3-2를 보면 쉽게 이해할 것이다. A 값은 51359고, B 값이 99962763일 때 A와 B를 더한 결과인 result 변수값은 100014122임을 알 수 있다.

그림 3-2 BIG_DECIMAL 구조체의 더하기 연산 메모리 구조

A와 B라는 두 개의 구조체 변수를 더할 때는 A 변수와 B 변수의 size 변수값 중 어느 것이 큰 값을 가졌는지 모르므로 biggerNum이라는 포인터 변수를 사용해 큰 값을 가리키게 해야 한다. 위 그림에서는 B 변수의 size 변수값이 크므로 biggerNum 변수는 B 변수의 메모리 주소값을 저장하게 된다.

더하기 연산에는 두 개의 for문을 사용하며, 첫 번째 for문은 작은 값(A 변수의 size 변수값)을 위해서 사용하고, 두 번째 for문은 큰 값(B 변수의 size 변수값)의 나머지 부분을 계산하는 데 사용한다. 더하기 결과 값을 저장하는 result 변수의 크기(result 변수의 size 변수값)는 B 변수의 size 변수값보다 1바이트 더 큰 공간을 초기에 할당한다. 그림 3-2를 이해했다면 이제 코드 3-2를 살펴보도록 하자.

코드 3-2 BIG_DECIMAL 구조체의 더하기 연산

```
BIG_DECIMAL PLUS(BIG_DECIMAL *A, BIG_DECIMAL *B)
{
    BIG_DECIMAL result; // --- ❶

    // --- ❷
    unsigned int min, max;

    BIG_DECIMAL *biggerNum = A->size > B->size ? A : B;
```

```
min = A->size > B->size ? B->size : A->size;
max = A->size > B->size ? A->size : B->size;

// --- ❸
unsigned int size = max + 1;
result.digit = (unsigned char *)malloc(size);

unsigned int i = 0; // --- ❹
unsigned char temp = 0; // --- ❺

for(;i < min; i++) // --- ❻
{
    // --- ❼
    result.digit[i] = A->digit[i] + B->digit[i] + temp;
    if(result.digit[i] > 0x09)
        temp = 0x01;
    else
        temp = 0x00;
    result.digit[i] %= 0x0A; // --- ❽
}
for(; i < max; i++) // --- ❾
{
    // --- ❿
    result.digit[i] = biggerNum->digit[i] + temp;
    if(result.digit[i] > 0x09)
        temp = 0x01;
    else
        temp = 0x00;
    result.digit[i] %= 0x0A;
}

// --- ⓫
if(temp)
{
```

```
            result.digit[i] = temp;
            result.size = size;
        }
        else
            result.size = size - 1;

        result.sign = 0;

        return result;
    }
```

❶에서는 결과 값을 저장하는 result 구조체 변수를 선언한다. BIG_DECIMAL 구조체의 지역 변수이므로 스택 영역에 9바이트를 할당할 것이다.

❷에서는 매개변수로 입력한 BIG_DECIMAL 구조체 변수 A와 B의 size 변수값 크기를 비교해 크기가 작은 size 변수값을 min 변수에 저장하고 크기가 큰 size 변수값은 max 변수에 저장한다. min과 max 변수는 ❻과 ❽ 두 개의 for문을 실행할 때 사용한다.

❸에서는 결과 값을 저장하는 공간을 할당하는데, A와 B 변수 중에서 size 변수값이 큰 것보다 1바이트 더 큰 공간을 할당한다.

❹에서는 BIG_DECIMAL 구조체 변수 A와 B의 digit 변수를 위한 인덱스로 i 변수를 사용하는데, i 변수는 0으로 초기화한 후 함수가 끝날 때까지 계속 증가한다. ❻과 ❽의 for문에서 초기화식에 아무것도 기입하지 않는 이유가 여기에 있다.

❺의 temp 변수는 0 또는 1을 갖는 변수로, 두 digit 변수를 더한 다음 10 이상이면 temp 변수에 1을 저장하고, 10보다 작으면 temp 변수에 0을 저장한다.

❻의 for문은 for(i = 0; i 〈 10; i++)와 같다. 즉, for (초기화식; 조건식; 증감)의 형태에서 '초기화식', '조건식' 그리고 '증감'은 생략이 가능하다. 여기서는 '초기화

식' 부분을 생략한 것이며, 이렇게 생략하는 일은 의외로 많다.

❼은 첫 번째 for문의 실행 내용으로 하위 digit 변수부터 상위 digit 변수를 차례대로 더한다. temp 변수에 입력되는 값은 상위 digit 변수에 더해지는 값이다. 즉, A와 B 변수에 의해 포인트 된 개체의 멤버인 digit 변수를 더해 9보다 크면 상위 digit 변수에 1이 더해지는 연산을 하려고 temp 변수를 사용한 것이다

❽에서는 나머지 연산자 (%)를 이용해 한 자리 숫자만 해당 digit 변수에 저장한다.

❾의 인덱스 i 변수는 위 for문에서 사용한 값을 그대로 받는다. 즉, 첫 번째 for문에서 A와 B 변수 중 size 변수 크기가 작은 값 중심으로 연산을 했다면, 두 번째 for문부터는 size 변수 크기가 큰 값 중심으로 연산이 이루어진다. 따라서 A와 B 변수가 아닌 앞에서 포인터 변수로 사용된 biggerNum 변수를 사용한다. ❷에서 포인터 변수 biggerNum은 A와 B 변수 중에서 size 변수가 큰 값을 가리키게 초기화했었다는 사실을 기억하자.

❿에서는 ❼에서와 거의 같은 연산을 실행한다. 다른 점이 있다면 size 변수 크기가 작은 값을 중심으로 연산을 마친 다음 size 변수 크기가 큰 값을 중심으로 나머지 부분을 연산한다는 점이다. 따라서 하위 digit 변수로부터 받는 값인 temp 변수만으로 연산을 실행한다.

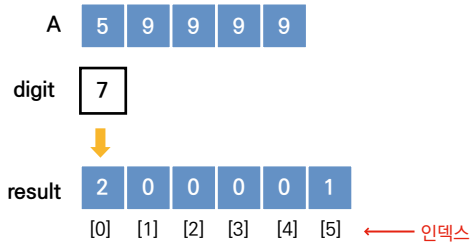
⓫은 결과 값을 저장하는 BIG_DECIMAL 구조체 변수 result의 최상위 digit 변수값을 저장한다. 하위 digit 변수에서 1을 받아야 한다면 temp 변수값이 1일 것이고, if문은 참이다. 1을 받지 않는다면 result 변수의 최상위 digit 변수값을 저장할 필요가 없다. 또한 최상위 digit 변수값이 1이면 result 변수는 ❸에서 지정한 size 변수값을 그대로 받을 것이고, 최상위 digit 변수값이 0이면 result 변수가 받는 size 변수값은 하나 줄어들 것이다. 왜냐하면 정수에서 0으로 시작하는 값은 없기 때문이다.

NOTE 코드 3-2를 설명하면서 쉽게 설명한다는 것이 참 어렵다고 생각했다. 이 책을 쓰면서 사실 초등학생도 이해할 수 있는 설명서를 작성하는 것을 목표로 삼았지만, 생각을 글로서 정리하는 일이 쉽지 않음을 느낀다. 과거를 돌이켜보면 영어를 잘 못하는 필자는 원서를 읽을 때, 설명은 거의 읽지 못하고 소스 코드만 보면서 이해하려고 했다. 영어를 해석하는 일에 더 많은 시간과 에너지를 소모했기 때문이다. 하지만 소스 코드를 보고 이해하려 했던 행동이 결국은 프로그래밍 능력을 키우는 데 도움이 되기도 했다. 즉, 독자들에게 강조하려는 것은 소스 코드 설명보다 소스 코드 자체를 보고 이해할 수 있다면 더 좋다는 사실이다.

이번에는 BIG_DECIMAL 구조체에서 한 자리 숫자의 일반 자료형 digit 변수 하나만을 더하는 PlusDigit() 함수를 살펴보자. 이 함수를 만든 이유는 소수Prime Number를 구하거나 간단한 연산을 할 때 요긴하게 사용할 수 있기 때문이다.

그림 3-3은 BIG_DECIMAL 구조체 변수 A(99995)에 한 자리 숫자의 일반 자료형 값을 가진 digit 변수(7)를 더해 결과 값으로 result 구조체 변수(= 100002)를 얻는 연산이다.

그림 3-3 BIG_DECIMAL 구조체에 한 자리 숫자의 일반 자료형 변수 digit을 더하는 연산의 메모리 구조



BIG_DECIMAL 구조체 두 개를 더하는 일을 이해했다면 한 자리 숫자의 일반 자료형인 digit 변수 하나만 더하는 일은 의외로 간단하다. 이는 코드 3-3을 살펴보면 알 수 있을 것이다.

코드 3-3 BIG_DECIMAL 구조체에 일반 자료형 변수를 더하는 PlusDigit() 함수

```
BIG_DECIMAL PlusDigit(BIG_DECIMAL *A, unsigned char digit) // --- ❶
{
    BIG_DECIMAL result;

    // --- ❷
    unsigned int size = A->size + 1;
    result.digit = (unsigned char *)malloc(size);

    int i;
    unsigned char temp;

    // --- ❸
    result.digit[0] = A->digit[0] + digit;
    temp = result.digit[0] / 0x0A;
    result.digit[0] %= 0x0A;

    // --- ❹
    for(i = 1; i < A->size; i++)
    {
        result.digit[i] = A->digit[i] + temp;
        temp = result.digit[i] / 0x0A;
        result.digit[i] %= 0x0A;
    }

    // --- ❺
    if(temp)
    {
        result.digit[i] = temp;
        result.size = size;
    }
    else
        result.size = A->size;
```

```
    result.sign = A->sign;

    return result;
}
```

❶에서는 매개변수로 BIG_DECIMAL 구조체 변수 A와 1바이트의 일반 자료형 변수인 digit을 입력받아 더한 값인 result 변수를 반환하는 PlusDigit() 함수를 선언한다.

❷에서는 malloc() 함수로 A 변수보다 1바이트 큰 result 변수의 결과 값을 저장할 공간(size 변수값)을 할당한다.

❸에서는 A 변수의 최하위 digit 변수와 매개변수로 입력받은 1바이트 digit 변수를 더한다. temp 변수는 상위 digit 변수에 더해지는 값으로 0 또는 1을 가진다.

❹에서는 A 변수의 최하위 digit 변수 연산으로부터 파생된 temp 변수로 A 변수와 나머지 digit 변수의 더하기 연산을 실행한다.

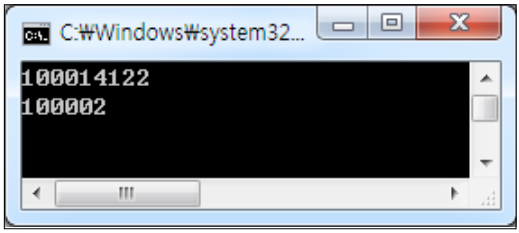
❺에서는 result 변수의 최상위 digit 변수에 값을 채우고 size 변수값을 정한다. 즉, 하위 digit 변수에서 넘겨지는 값이 있으면(temp 변수값이 1이면) result 변수의 최상위 digit 변수값은 1이 된다.

NOTE 코드 3-3에서 매개변수로 입력하는 digit 변수는 0~9여야만 한다. 그런데 상용화되거나 다른 프로그램에서 호출하는 함수로 사용하려면 반드시 예외 처리를 해주어야 한다. 즉, 0~9 이외의 입력값에서는 에러가 발생해야 한다. 그러나 여기서는 공부하는 목적의 소스 코드이므로 예외 처리까지는 구현하지 않았다. 소스 코드가 복잡해질 수 있기 때문이다. 하지만 실무에서는 예외 처리를 많이 하는데, 이는 소스 코드가 길어졌을 때 어느 부분에서 문제가 생겼는지를 알기 위해서이며 심할 때는 실제 구현하려는 소스 코드만큼 예외 처리 구문을 작성할 수도 있다.

다음은 Big Number 연산의 더하기 함수를 테스트한 예제다. 함수를 어떻게 호출했는지 살펴보기로 하자. 어렵지 않은 코드이므로 설명은 생략한다.

코드 3-4 더하기 연산 테스트

```
int main()
{
    BIG_DECIMAL A, B, result;

    A = CreateDecimal((unsigned char *)"51359", 5);
    B = CreateDecimal((unsigned char *)"99962763", 8);

    result = PLUS(&A, &B);
    printDecimal(result);

    A = CreateDecimal((unsigned char *)"99995", 5);

    result = PlusDigit(&A, 7);
    printDecimal(result);
}
```



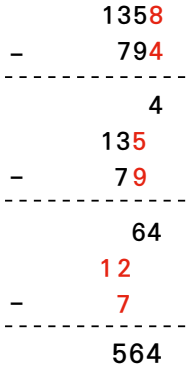
프로그래밍에서 테스트만큼 중요한 것이 없다. 위에서는 간단하게 테스트를 했지만, 여러분이 직접 구현을 할 때는 테스트를 위한 main() 함수에 다양한 테스트 구문을 작성한 후 PLUS()와 PlusDigit() 함수를 구현해보기 바란다.

NOTE 프로그래밍이란 함수를 다 만든 후에 테스트를 하는 것이 아니라, 함수를 구현하면서 끊임없이 테스트를 해야 한다. 필자는 10명의 개발자보다 1명의 전문 테스터를 두는 것이 효과적일 때가 많다고 생각한다. 그만큼 테스트는 중요하다.

3.2 빼기 연산

더하기와 마찬가지로 빼기 연산도 그림 3-4처럼 가장 낮은 자리의 숫자부터 순차로 빼기 연산을 하면 된다.

그림 3-4 빼기 연산



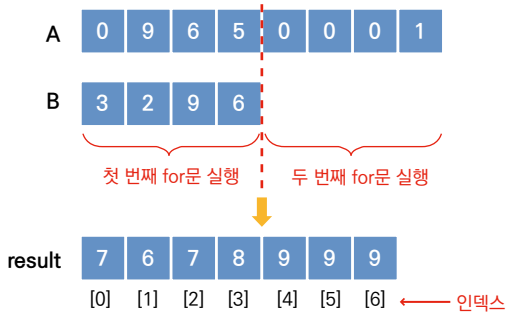
그럼 BIG_DECIMAL 구조체에서는 어떨까? 가장 낮은 digit[0]부터 빼기 연산을 실행하면 될 것이고, 이때 부호(sign 변수)가 음수로 바뀔 수도 있을 것이다. 다음은 빼기 연산을 각 digit 변수에서 어떻게 실행할지 일반화한 구문이다. result = MINUS(A, B)의 내부에서 처리하는 구문이며, temp 변수는 하위 digit 변수에서 상위 digit 변수로부터 값을 빌릴 때 1을 빼주기 위해 사용하는 값이다.

코드 3-5 빼기 연산의 일반화 구문

```
result.digit[i] = A.digit[i] - B.digit[i] - temp  //if 0<=i<n, temp = 0 or 1
```

두 개의 양수가 저장된 변수 A에서 변수 B를 뺄 때 B 변수값이 더 크면 결과 값은 음수일 것이고, A 변수값이 더 크면 양수일 것이다. 빼기 연산을 쉽게 구현하기 위해 A 변수가 B 변수보다 항상 크다고 가정하자. 즉, A 변수의 size 변수값이 B 변수의 size 변수값보다 항상 크거나 같다고 가정하고 BIG_DECIMAL 구조체 변수의 연산을 실행해보자. 그림 3-5는 A 변수값이 10005690이고 B 변수값이 6923일 때, 빼기 연산의 결과인 result 변수값이 9998767임을 보여준다.

그림 3-5 BIG_DECIMAL 구조체의 빼기 연산 메모리 구조



더하기 연산에서처럼 두 개의 for문을 사용하는데, 첫 번째 for문은 size 변수값이 작은 B 변수를 중심으로 연산을 실행하고, 두 번째 for문은 나머지 부분의 연산을 실행한다. 결과 값을 저장하는 result 변수의 size 변수값은 A 변수의 size 변수값으로 초기화하면 된다. 코드 3-6에는 빼기 연산을 위해 함수 두 개를 사용했는데, 첫 번째인 MINUS() 함수는 일반적인 연산이고, 두 번째인 MinusAbsolute() 함수는 A - B 연산에서 A 변수가 항상 B 변수보다 크다는 가정 아래 연산을 실행한다. 빼기 연산은 입력받는 매개변수가 항상 양수값이라는 가정을 했으며, 음수라면 좀 더 많은 경우의 수를 만들어야 한다. 그러나 여기서는 학습이 목적이므로 최대한 간단하게 구현했으며, 개인적으로 더 멋진 함수를 원하는 독자라면 코드 3-6에 더 많은 조건을 추가하면 된다.

코드 3-6 BIG_DECIMAL 구조체끼리 빼기 연산을 하는 MINUS()와 MinusAbsolute() 함수

```
// A 변수와 B 변수는 항상 양수다.
BIG_DECIMAL MINUS(BIG_DECIMAL *A, BIG_DECIMAL *B) // --- ❶
{
    BIG_DECIMAL result;

    // --- ❷
    if(IsBigger(A, B))
    {
        result = MinusAbsolute(A, B);
        result.sign = 0;
    }
    else
    {
        result = MinusAbsolute(B, A);
        result.sign = 1;
    }
    return result;
}

// A 변수는 B 변수보다 항상 크다.
BIG_DECIMAL MinusAbsolute(BIG_DECIMAL *A, BIG_DECIMAL *B) // --- ❸
{
    BIG_DECIMAL result;

    result.digit = (unsigned char *)malloc(A->size); // --- ❹

    int i = 0; // --- ❺
    unsigned char temp = 0; // --- ❻

    for(; i < B->size; i++) // --- ❼
    {
```

```
        // --- ❽
        if(A->digit[i] >= (B->digit[i] + temp))
        {
            result.digit[i] = A->digit[i] - B->digit[i] - temp;
            temp = 0;
        }
        else
        {
            result.digit[i] = A->digit[i] + 10 - B->digit[i] - temp;
            temp = 1;
        }
    }
    for(; i < A->size; i++) // --- ❾
    {
        // --- ❿
        if(A->digit[i] >= temp)
        {
            result.digit[i] = A->digit[i] - temp;
            temp = 0;
        }
        else
        {
            result.digit[i] = A->digit[i] + 10 - temp;
            temp = 1;
        }
    }

    result.size = A->size;

    // --- ⓫
    while(!result.digit[i-1] && i>1)
    {
```

```
            result.size--;
            i--;
        }

        result.sign = 0;

        return result;
    }
```

❶에서는 BIG_DECIMAL 구조체 변수 A와 B를 매개변수로 받아서 빼기 연산을 실행한 후 결과 값인 result 변수를 반환하는 MINUS() 함수를 선언한다.

❷에서 A 변수가 B 변수보다 크면 결과는 양수고, B 변수보다 작으면 음수다. IsBigger() 함수는 '2.7 비교 함수'에서 구현했으며, 두 변수의 크기를 비교한다. MinusAbsolute() 함수는 ❸에서 구현한다.

❸의 MinusAbsolute() 함수는 MINUS() 함수의 보조 함수로, 매개변수 A는 B보다 항상 커야 한다.

❹에서는 결과 값의 크기(size 변수)를 힙 영역에 할당하는데, 빼기 연산이므로 최대 크기는 A 변수값과 같다.

❺에서는 BIG_DECIMAL 구조체 변수 A와 B 변수의 digit 변수를 위한 인덱스로 사용되는 i 변수를 선언했다. 0으로 초기화하며 for문 두 개를 실행하는 동안 계속 증가한다(❼과 ❾ for문에서 두 개의 초기화식에 아무것도 기입하지 않는 이유가 여기에 있다).

❻의 temp 변수는 0 또는 1을 갖는 변수로, 하위 digit 변수에서 빼기 연산을 실행할 때 값이 모자라 상위 digit 변수로부터 값을 빌리면 temp 변수에 1을 저장하고, 값을 빌려주지 않으면 0을 저장한다.

❼에서는 첫 번째 for문으로 작은 size 변수값을 가진 B 변수를 기준으로 연산을 실행한다.

❽에서는 각각의 digit 변수에 빼기 연산을 실행했을 때의 값이 빼는 값보다 큰지 작은지에 따라서 다른 연산을 실행한다. 즉, 빼는 값인 A->digit[i]가 더 크다면 상위 digit 변수로부터 10을 빌려 오지 않아도 되지만 작다면 상위 digit 변수로부터 10을 빌려 와서 연산을 실행해야 한다. if문은 상위 digit 변수에서 값을 빌리지 않고 연산을 실행한 것이고, else문은 상위 digit 변수에서 10만큼 빌려 와서 연산을 실행했다. 상위 digit 변수에서 10을 빌려 오면 temp 변수는 1로 설정해서 상위 digit 변수의 연산을 실행할 때 temp 변수에 영향을 미치게 된다.

❾는 두 번째 for문으로 빼는 값인 A의 나머지 부분에 대한 연산을 실행한다. 즉, B 변수의 size 변수값보다 큰 위치에 있는 digit 변수의 연산이므로 temp 변수만으로 연산을 실행한다.

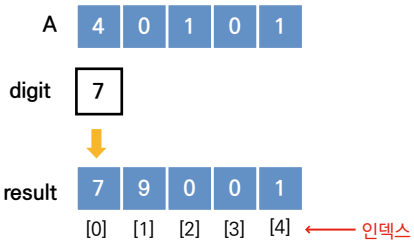
❿의 하위 digit 변수의 연산에서는 10을 빌렸는지의 여부에 따라 연산을 실행해야 하는데, 10을 빌렸다면 상위 digit 변수는 1을 빼주게 된다. 하위 digit 변수에서 10을 빌렸는지의 여부에 따라서 temp 변수값이 1 또는 0이 된다. 또한 temp 변수값이 1일 때, 현재의 digit 변수에서 빼는 값 A->digit[i]가 temp 변수보다 작다면 상위 digit 변수로부터 10을 빌려 오게 된다.

⓫에서 높은 자리의 digit 변수값이 0이라면 이 값은 무시해야 한다. 예를 들어 결과 값이 00001234라면 앞부분의 0000은 무시해야 하며 실제 size 변수값은 8이 아니라 4가 되어야 한다. 이처럼 빼기 연산 후 앞부분의 값이 0일 때는 BIG_DECIMAL 구조체 변수의 size 변수값을 조정함으로써 최종 결과 값을 가진다.

다음은 BIG_DECIMAL 구조체의 변수에서 한 자리 숫자의 일반 자료형인 digit 변수를 빼는 연산을 살펴보기로 하자. Big Number 연산에서 많이 활용하지는 않지

만 함수의 일관성을 맞추려고 구현했다. 그림 3-6은 10104에서 7을 빼서 10097이라는 값을 얻는 연산이다.

그림 3-6 BIG_DECIMAL 구조체 변수에서 한 자리 숫자의 일반 자료형 변수 digit을 빼는 연산의 메모리 구조



BIG_DECIMAL 구조체에서 한 자리 숫자의 digit 변수(0~9)를 빼는 함수인데, MINUS() 함수를 이해했다면 지금부터 소개할 MinusDigit() 함수는 금방 이해할 수 있을 것이다. 왜냐하면 첫 번째 for문 대신 최하위 digit 변수인 A->digit[0]으로 연산을 실행했기 때문이다.

코드 3-7 BIG_DECIMAL 구조체에서 한 자리 숫자의 일반 자료형 변수 digit을 빼는 MinusDigit() 함수

```
BIG_DECIMAL MinusDigit(BIG_DECIMAL *A, unsigned char digit)
{
    BIG_DECIMAL result;

    result.digit = (unsigned char *)malloc(A->size);

    int i;
    unsigned char temp = 0x00;

    // --- ❶
```

```
if(A->digit[0] >= digit)
    result.digit[0] = A->digit[0] - digit;
else
{
    result.digit[0] = A->digit[0] + 10 - digit;
    temp = 1;
}

for(i = 1; i < A->size; i++)
{
    if(A->digit[i] >= temp)
    {
        result.digit[i] = A->digit[i] - temp;
        temp = 0;
    }
    else
    {
        result.digit[i] = A->digit[i] + 10 - temp;
        temp = 1;
    }
}

result.size = A->size;

while(!result.digit[i-1] && i>1)
{
    result.size--;
    i--;
}

result.sign = false;
```

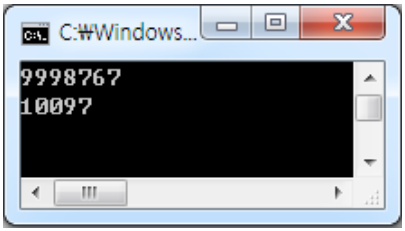
```
    return result;
}
```

코드 3-7은 앞에서 설명한 MinusAbsolute() 함수와 거의 같으므로 설명을 생략하겠다. 다른 점은 ❶ 부분으로, 최하위 digit 변수인 A->digit[0]의 연산을 가장 먼저 실행한다는 것이다.

코드 3-8은 위에서 설명한 함수를 테스트한다.

코드 3-8 빼기 연산 테스트

```
int main()
{
    BIG_DECIMAL A, B, result;

    A = CreateDecimal((unsigned char *)"10005690", 8);
    B = CreateDecimal((unsigned char *)"6923", 4);

    result = MINUS(&A, &B);
    printDecimal(result);

    A = CreateDecimal((unsigned char *)"10104", 5);

    result = MinusDigit(&A, 7);
    printDecimal(result);
}
```



3.3 곱하기 연산

곱하기 연산도 그림 3-7과 같이 곱하는 값을 한 자리 숫자 단위로 쪼개서 계산한 후 모두 더하는 구조다.

그림 3-7 곱하기 연산

```
    11111
×     123
-------
    33333
   222220
+ 1111100
-------
  1366653
```

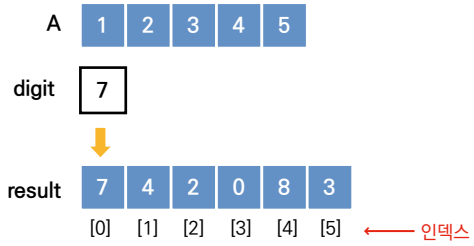
곱하기 연산도 원리는 간단하기에 긴 설명은 하지 않는다. 그렇다면 어떻게 구현하는 것이 좋을까? 더하기 연산과 빼기 연산의 구현은 간단했지만 곱하기 연산부터는 좀 복잡할 수 있다. 코드 3-9는 result = MULTIPLY(A, B)라는 연산 안에서 처리하는 구문이며, i 변수는 0부터 순차로 증가한다.

코드 3-9 곱하기 연산의 일반화 구문

```
result = PLUS (result, MULTYPLY(A, B->digit[i]))
```

곱하기 연산은 digit 변수 단위로 연산을 실행한 후 결과 값 result 변수에 계속 값을 더하도록 구현하면 된다. 먼저 BIG_DECIMAL 구조체 변수에 한 자리 숫자의 일반 자료형인 digit 변수(0~9)를 곱하는 연산을 구현해야 한다. 그림 3-8은 54321에 7을 곱해 380247이라는 결과 값을 얻는 연산의 구조다.

그림 3-8 BIG_DECIMAL 구조체와 한 자리 숫자의 일반 자료형 변수 digit을 곱하는 연산의 메모리 구조



위 그림과 같이 result 변수의 최대 크기는 A 변수의 size 변수값보다 하나 더 큰 값이다. 또한 연산은 A 변수의 제일 왼쪽 digit 변수부터 차례대로 곱하기를 실행한다. 코드 3-10은 BIG_DECIMAL 구조체 변수와 한 자리 숫자의 일반 자료형인 digit 변수를 곱하는 MultiplyDigit() 함수다.

코드 3-10 BIG_DECIMAL 구조체 변수와 일반 자료형 digit 변수 하나를 곱하는 MultiplyDigit() 함수

```
BIG_DECIMAL MultiplyDigit(BIG_DECIMAL *A, unsigned char digit) // --- ❶
{
    BIG_DECIMAL result;

    // --- ❷
    unsigned int size = A->size + 1;
    result.digit = (unsigned char *)malloc(size);

    int i; // --- ❸
    unsigned char temp = 0; // --- ❹

    for (i=0;i<A->size;i++) // --- ❺
    {
        result.digit[i] = (A->digit[i] *digit) + temp; // --- ❻

        temp = result.digit[i] / 0x0A; // --- ❼
        result.digit[i] %= 0x0A; // --- ❽
```

```
    }

    // --- ❾
    if (temp)
    {
        result.digit[i] = temp;
        result.size = size;
    }
    else
        result.size = A->size;

    result.sign = false; // --- ❿

    return result;
}
```

❶에서는 BIG_DECIMAL 구조체 변수 A와 1바이트의 일반 자료형 digit 변수를 입력받아 결과 값인 result 변수를 반환하는 MultiplyDigit() 함수를 선언한다.

❷에서는 결과 값을 저장하는 result 변수의 size 변수값을 A 변수보다 1바이트 더 큰 값으로 설정한다.

❸의 i 변수는 인덱스로 사용하며 0부터 순차로 증가하면서 곱하기 연산을 실행하는 데 사용한다.

❹의 temp 변수는 하위 digit 변수에서 상위 digit 변수로 전달하는 값을 저장하는 임시 저장 공간으로, '8 × 9 = 72'라는 연산이라면 temp 변수는 7을 저장한다.

❺에서는 A 변수 각각의 digit 변수를 연산하려고 for문을 사용하는데, 인덱스 i 변수는 0부터 증가하므로 A 변수의 하위 digit 변수부터 차례대로 연산한다.

❻에서는 A 변수의 digit 변수 중 하나인 A->digit[i]와 매개변수로 입력받은 digit

변수를 곱하고, 이전 연산에서 넘겨받은 temp 변수(상위 digit 변수로 전달하는 올림 값)를 더한다.

❼에서는 다음 상위 digit 변수로 전달하는 값을 temp 변수에 저장한다. 예를 들어 '8 × 9 = 72'라는 계산의 결과 값인 72에서 10으로 나눈 정수 부분의 7을 temp 변수에 저장한다. 이 값은 다음 상위 digit 변수 연산에서 더하게 된다.

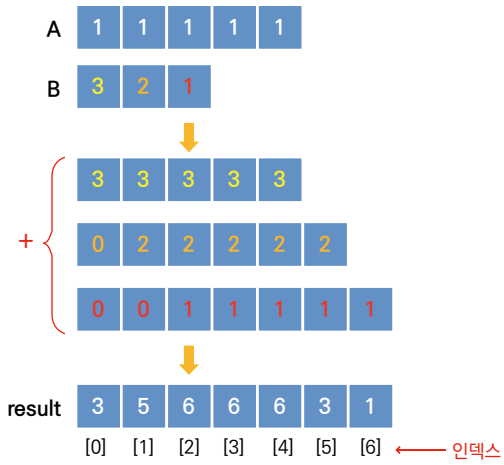
❽에서는 현재 digit 변수값인 result.digit[i]에 결과 값을 저장하려고 나머지 연산을 실행한다. 예를 들어 '8 × 9 = 72'라는 연산의 결과 값 72를 10으로 나눈 나머지인 2를 result.digit[i]에 저장한다. digit 변수는 0~9의 숫자만을 가질 수 있기 때문이다.

❾는 결과 값 result 변수의 최상위 digit 변수에 결과를 저장하는 연산이다. for 문이 A 변수를 중심으로 연산을 실행했다면, if~else문은 A 변수의 size 변수값을 벗어난 result 변수의 최상위 digit 변수 중 하나에 결과를 저장한다. 예를 들면 '54321 × 7 = 380247'이라는 연산에서 결과 값의 최상위 digit 변수인 3을 저장한다고 생각하면 된다. 최상위 digit 변수에 저장할 값이 있을 때는 if문 안의 내용을 실행하고 그렇지 않으면 else문처럼 A 변수의 size 변수값만을 result 변수에 저장한다.

❿에서 곱하기 연산은 항상 양수만을 계산한다는 가정하에 만들어졌으므로 부호는 양수(false)가 된다.

digit 변수 하나를 곱하는 MultiplyDigit() 함수를 구현했으므로, BIG_DECIMAL 구조체 변수의 곱하기 연산은 MultiplyDigit() 함수를 사용하면 된다. 이제는 곱하는 값이 1바이트의 digit 변수가 아닌 BIG_DECIMAL 구조체 변수며, 그림 3-9처럼 각각의 digit 변수를 나눠서 연산을 실행한 후 BIG_DECIMAL 구조체 변수를 더하면 된다. 그림에서는 '11111 × 123 = 1366653'이라는 연산을 어떻게 실행하는지를 보여준다.

그림 3-9 BIG_DECIMAL 구조체끼리의 곱하기 연산 메모리 구조



곱하기 연산에서는 자릿수를 맞춰주는 일이 중요한데, 위 그림처럼 B 변수의 digit 변수 위치에 따라 곱한 결과 값의 하위 digit 변수를 0으로 채워야 하는 경우가 발생한다. 예를 들면 '11111 × 100 = 1111100'이라는 연산은 '11111 × 1 = 11111'의 MultiplyDigit() 함수 연산을 실행한 후 결과 값 뒤에 00을 넣어 1111100이라는 값을 만들어야 한다.

코드 3-11은 MultiplyDigit() 함수를 이용해서 BIG_DECIMAL 구조체 변수를 곱하는 MULTIPLY() 함수를 살펴볼 수 있다. 함수 안에서 사용되는 tempDigit 변수는 MultiplyDigit() 함수를 실행한 결과 값을 저장하며, temp 변수는 tempDigit 변수값에 0을 채워서 자릿수를 맞추기 위한 임시 저장 공간이다. 중요한 점은 free() 함수를 사용해 메모리 누수를 없애는 것인데, 계산을 위해 힙 영역에 할당한 공간을 계산이 끝난 후에 free() 함수를 사용해 없애준다. 메모리 관점에서 보면 malloc() 함수와 free() 함수를 확실히 이해할 수 있는 좋은 예제다.

코드 3-11 BIG_DECIMAL 구조체 변수끼리 곱하는 Multiply() 함수

```
BIG_DECIMAL MULTIPLY(BIG_DECIMAL *A, BIG_DECIMAL *B)  // --- ❶
{
    BIG_DECIMAL result, temp, tempDigit; // --- ❷
    unsigned char *ptrTemp; // --- ❸

    result = CreateDecimal((unsigned char *)"0", 1); // --- ❹

    temp.digit = (unsigned char *)malloc(A->size + B->size); // --- ❺

    int i, j;

    for(i = 0; i < B->size; i++) // --- ❻
    {
        tempDigit = MultiplyDigit(A, B->digit[i]); // --- ❼

        // --- ❽
        for(j = 0; j < i; j++)
            temp.digit[j] = 0x00;
        for( ; j < tempDigit.size+i; j++)
            temp.digit[j] = tempDigit.digit[j-i];
        temp.size = tempDigit.size + i; // --- ❾

        free(tempDigit.digit); // --- ❿

        // --- ⓫
        ptrTemp = result.digit;
        result = PLUS(&result, &temp);
        free(ptrTemp);
    }
    free(temp.digit); // --- ⓬

    return result;
}
```

❶에서는 BIG_DECIMAL 구조체 변수 A와 B를 매개변수로 받아서 곱하기 연산을 실행한 후 결과 값인 BIG_DECIMAL 구조체 변수 result를 반환하는 MULTIPLY() 함수를 선언한다.

❷에서 result 변수는 결과 값을 저장하는 변수고, tempDigit 변수는 digit 변수 하나를 곱하는 MultiplyDigit() 함수를 실행해 얻은 결과를 임시로 저장하는 변수다. temp 변수는 tempDigit 변수값으로 하위 digit 변수에 0을 채워서 자릿수를 맞춘 값을 임시 저장하는 변수다.

❸의 ptrTemp는 포인터 변수로 힙 영역의 주소를 가리키며 free() 함수로 해당 메모리 공간을 없애려고 사용한다. ptrTemp 변수는 ⓫에서 사용하며, PLUS() 함수를 실행한 후 result 변수는 새로운 메모리 공간을 갖게 되므로 이전 메모리 공간을 지워주어야 한다. 이때 ptrTemp 포인터 변수로 이전 공간을 가리키게 해 PLUS() 함수 실행을 끝낸 후에는 free(ptrTemp)로 이전 메모리 공간을 지운다.

❹에서 결과 값을 저장하는 result 변수는 0으로 초기화하며, 이후에 digit 변수 단위로 계산한 값을 더하게 된다.

❺에서는 임시 저장 공간으로 사용하는 temp 변수의 메모리 공간을 할당한다. 이때 최대 크기는 매개변수 A와 B의 size 변수를 더한 값이다.

❻에서 i 변수는 곱하는 값인 B 변수의 digit 변수들을 위한 인덱스로 사용한다. B 변수 각각의 digit 변수들로 MultiplyDigit() 함수를 실행하는 데 사용한다.

❼은 B 변수의 digit 하나를 가지고 A 변수와 곱하기 연산을 한다. MultiplyDigit() 함수 안에서는 malloc() 함수를 실행해 메모리를 할당하기 때문에 ⓾에서처럼 free() 함수를 사용해 반드시 할당한 메모리를 없애야 한다. 그래야만 메모리 누수를 막을 수 있다.

❽에서는 B 변수의 digit 변수로 연산을 실행한 후 해당 digit 변수가 가진 자릿수에 따라서 0을 채워 실제 곱하기 연산을 처리해야 한다. ❽ 안의 첫 번째 for문은 temp 변수의 하위 digit 변수들에 0을 채우는 것이고, 두 번째 for문은 tempDigit 변수값들을 temp 변수에 저장하는 것이다. 예를 들어 '1234 × 2000 = 2468000' 연산에서 '1234 × 2 = 2468'을 실행한 후 2468을 tempDigit 변수에 저장했다면 temp 변수는 000을 채운 후 2468을 앞에 넣어주어 2468000을 만든다.

❾에서 temp.size는 tempDigit 변수의 size 변수값에 하위 digit 변수에 존재하는 0의 개수를 더한 값이다.

❿에서 tempDigit 변수는 사용을 완료했으므로 메모리 공간을 없애주어야 한다.

⓫에서는 PLUS() 함수로 result = result + temp를 실행해 각각의 digit 변수 단위로 계산된 값을 결과 값인 result 변수에 더해준다. 이때 result 변수는 힙 영역에 새로운 공간을 할당받으며, 연산 전에 result 변수가 가진 메모리 공간은 연산 후에 free() 함수와 포인터 변수 ptrTemp를 사용해 지워야 한다.

⓬에서는 함수 실행을 끝내기 전 임시 저장 공간으로 사용한 temp 변수의 메모리 공간을 지운다.

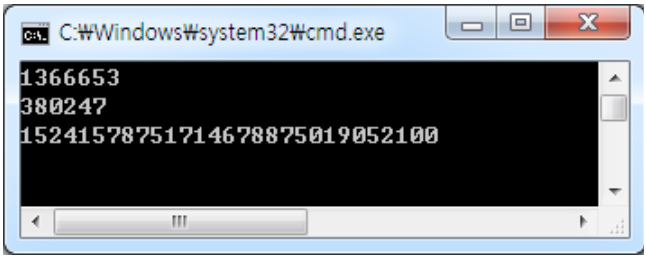
위 함수에서는 알고리즘도 중요하지만 무엇보다 중요한 것은 연산 중 할당한 메모리 공간을 지우는 것이다. 이는 메모리 누수를 방지하기 위함인데, 알고리즘이 잘못되었다면 잘못된 결과가 발생하므로 쉽게 수정할 수 있다.

하지만 메모리 누수 현상은 찾아내기가 상당히 까다롭다. 코드를 작성할 때 신경 쓰지 않으면 당장은 문제가 발견되지 않더라도 나중에 심각한 문제를 일으킬 수도 있고 심지어는 문제를 찾기도 어려워질 수 있다. 프로그램을 실행했는데 한 달 후에 프로그램이 멈춰버리면 얼마나 난감하겠는가?

코드 3-12는 지금까지 구현한 함수를 테스트하는 예제다.

코드 3-12 곱하기 연산 테스트

```
int main()
{
    BIG_DECIMAL A, B, result;

    A = CreateDecimal((unsigned char *)"11111", 5);
    B = CreateDecimal((unsigned char *)"123", 3);

    result = MULTIPLY(&A, &B);
    printDecimal(result);

    A = CreateDecimal((unsigned char *)"54321", 5);

    result = MultiplyDigit(&A, 7);
    printDecimal(result);

    A = CreateDecimal((unsigned char *)"12345678901234567890", 20);
    B = CreateDecimal((unsigned char *)"1234567890", 10);

    result = MULTIPLY(&A, &B);
    printDecimal(result);
}
```



곱하기 연산부터는 구현이 다소 어렵게 느껴질 수 있다. 하지만 알고 나면 아주 간단하다. 모를 때는 막막하지만 알고 나면 별거 아닌 것이 프로그래밍이다. 아마도 1년 후에 여러분이 이 책을 다시 본다면 너무나도 유치하다고 생각할지도 모른다.

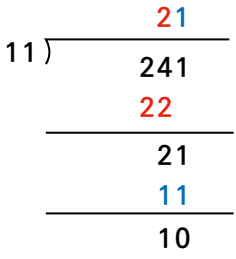
3.4 나누기 연산

나누기 연산은 결과 값으로 실수(소수점을 가진 수)를 받는 때가 많다. 하지만 이 책에서 다루는 나누기 연산은 정수를 결과로 받는다. 즉, 소수점 오른쪽 부분을 버린다는 의미다

왜냐하면 BIG_DECIMAL 구조체를 만들 때 소수를 표현하지 않는다고 가정했기 때문이기도 하지만 엄밀히 따지면 암호학의 기초가 되는 대학 수학의 정수론 Number Theory은 실수 범위를 다루지 않기 때문이기도 하다.

나누기 연산은 어려운 연산이 아니다. 그림 3-10은 '241 / 11 = 21'이라는 연산의 실행 예인데 나머지 10은 버린다. 즉, 결과 값 21만을 취한다.

그림 3-10 나누기 연산

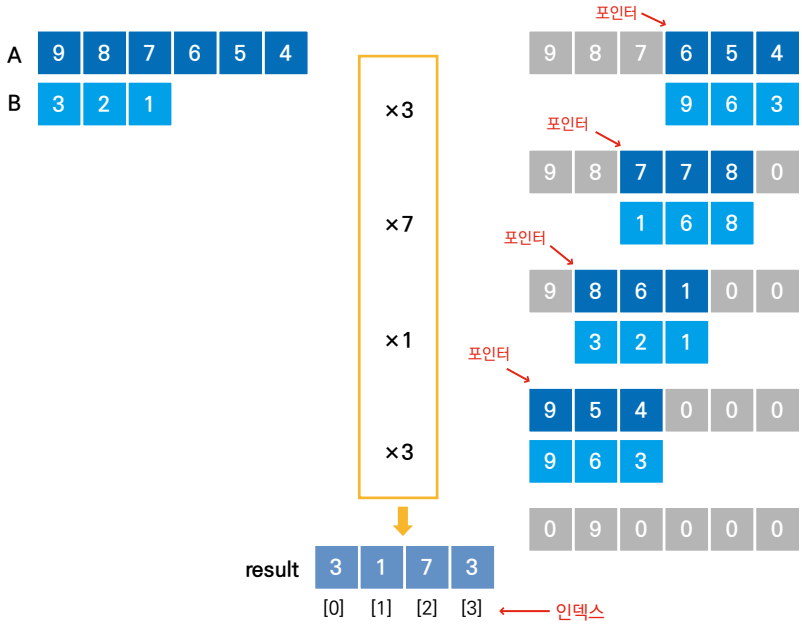


나누기 연산은 Big Number 연산에서 가장 복잡한 연산의 하나다. 이전 연산은 하위 digit 변수부터 연산을 실행하면 되지만, 나누기 연산은 상위 digit 변수부터 연산을 실행해야 하기 때문이다. 이러한 연산을 실행하려면 포인터를 적절히 사용해야 한다.

그림 3-11은 BIG_DECIMAL 구조체로 '456789 / 123 = 3713'이라는 연산을 실행하는 방법을 보여주는 예다. 그림에서 중요하게 살펴봐야 할 부분은 포인터의 위치로, 맨 위 포인터의 위치는 중간에 존재하며 아래로 내려갈수록 포인터의 위치가 왼쪽으로 한 digit 변수만큼씩 이동한다. 즉, 포인터를 사용해 나누는 수를 작은 수로 만들어서 빼기 연산을 실행하는 것으로, 이를 마지막 하위 digit 변수까지 반복적으로 실행하면 나누기 연산의 결과를 얻을 수 있다.

나누기 연산의 결과는 나누는 수에 몇을 곱하는지로 결과 값을 얻는데, 그림에서는 노란색 사각형 안의 값을 통해 결과 값을 가진다. 즉, Big Number의 나누기 연산에서 가장 중요한 것은 포인터를 이용한 자릿수 계산이고, 여기에 빼기 연산을 실행하면 된다고 생각하면 된다.

그림 3-11 BIG_DECIMAL 구조체의 나누기 연산 메모리 구조



그럼 위 나누기 연산을 어떻게 구현하면 좋을까? 다음은 위 그림을 어떻게 구현할지에 관한 순서를 정리한 것이다.

① 위 그림에서 설명하는 것 중 가장 중요한 포인터 위치를 초기화해야 한다. A 변수의 원래 포인터 위치에 A->size에서 B->size 값을 뺀 만큼 더해 오른쪽으로 이동하면 된다. 즉, A 변수의 원래 포인터 위치는 최하위 digit 변수값인 9를 가리키지만, 초기화된 포인터 위치는 오른쪽으로 세 번 이동해 6을 가리킨다.

② 포인터 위치를 초기화한 후 새롭게 만들어진 수 456에서 나누는 값 321에 3을 곱한 369를 빼준다. 이때 369는 123의 배수면서 456보다는 작은 수 중 가장 큰 수다. 이를 코드로 구현하려면 456에서 123을 빼주는 연산을 세 번 반복하도록 구현해야 한다. 그리고 세 번 반복한 결과를 나누기 연산의 결과값인 result 변수의 digit 변수 하나에 넣어주면 된다.

③ 포인터 위치를 왼쪽으로 digit 변수값 하나만큼 옮긴다. 이렇게 새로운 값을 만들어 ②와 같은 연산을 실행하게 되며, 포인터 위치가 A 변수의 원래 포인터 위치와 같아질 때까지 반복적으로 실행해야 한다.

나누기 연산은 자릿수를 정확히 알고 프로그래밍해야 한다. 그림 3-11에서 포인터가 가장 중요하다고 했는데, 이는 포인터가 자릿수에 변화를 주는 데 사용되기 때문이다. 코드 3-13은 BIG_DECIMAL 구조체의 나누기 연산을 구현한 것이다. 두 개의 함수 DIVIDE()와 MinusForDivide()를 구현했는데, MinusForDivide() 함수는 나누기 연산만을 위해 만들었다. 즉, 나누기 연산에서는 이전에 만들어진 빼기 연산을 사용할 수가 없다. 왜냐하면 빼기 연산은 나누는 값에 영향을 주지 않지만, MinusForDivide() 함수는 나누는 값인 매개변수 값에 변화를 주기 때문이다(즉, '참조에 의한 호출' 기법으로 변화를 준다). 그럼 코드 3-13을 자세히 살펴보도록 하자.

코드 3-13 BIG_DECIMAL 구조체끼리 나누기 연산을 하는 DIVIDE() 함수

```
BIG_DECIMAL DIVIDE(BIG_DECIMAL *A, BIG_DECIMAL *B) // --- ❶
{
    BIG_DECIMAL result;
    int i, j, count; // --- ❷

    // --- ❸
    if(!IsBigger(A,B))
    {
        result = CreateDecimal((unsigned char *)"0", 1);
        return result;
    }

    // --- ❹
    BIG_DECIMAL numerator;

    numerator.digit = (unsigned char *)malloc(A->size);
    numerator.size = A->size;

    for(j = 0; j < numerator.size; j++)
        numerator.digit[j] = A->digit[j];

    // --- ❺
    i = A->size - B->size;
    result.digit = (unsigned char *)malloc(i+1);

    for(j = 0; j <= i; j++)
        result.digit[j] = 0x00;

    unsigned char* ptrForOrigin = numerator.digit; // --- ❻

    // --- ❼
```

```
numerator.digit += (numerator.size - B->size);
numerator.size = B->size;

for( ; i >= 0; i--) // --- ❽
{
    count = 0; // --- ❾

    // --- ❿
    while (IsBigger(&numerator, B))
    {
        count++;
        MinusForDivide(&numerator, B);
    }
    result.digit[i] = (unsigned char)count; // --- ⓫

    // --- ⓬
    numerator.digit--;

    if(numerator.digit[numerator.size] != 0x00)
        numerator.size++;
}

free(ptrForOrigin); // --- ⓭

// --- ⓮
i = A->size - B->size;

for( ; i > 0; i--)
    if(result.digit[i] != 0x00)
        break;

result.size = i + 1;
result.sign = 0;
```

```
    return result;
}

// --- ⓯
void MinusForDivide(BIG_DECIMAL *A, BIG_DECIMAL *B)
{
    int i;
    unsigned char temp = 0;

    for(i = 0; i < B->size; i++)
    {
        if(A->digit[i] >= (B->digit[i] + temp))
        {
            A->digit[i] = A->digit[i] - B->digit[i] - temp;
            temp = 0;
        }
        else
        {
            A->digit[i] = A->digit[i] + 0x0A - B->digit[i] - temp;
            temp = 0x01;
        }
    }
    for( ; i < A->size; i++)
    {
        if(A->digit[i] >= temp)
        {
            A->digit[i] = A->digit[i] - temp;
            temp = 0;
        }
        else
        {
            A->digit[i] = A->digit[i] + 0x0A - temp;
```

```
            temp = 0x01;
        }
    }

    while(!A->digit[i-1] && i>1)
    {
        A->size--;
        i--;
    }
}
```

❶은 BIG_DECIMAL 구조체 변수 A와 B를 매개변수로 받아 나누기 연산을 실행하는 DIVIDE() 함수로, 함수 실행 후에도 매개변수에는 변화가 없다.

❷의 i와 j 변수는 인덱스로 사용하며, count 변수는 빼기 연산을 몇 번 실행했는지를 계산해 결과 값에 저장하려고 사용한다.

❸에서 나누는 값인 B 변수가 A 변수보다 크면 나누기 결과 값은 0이 된다.

❹에서 numerator 변수는 나누는 값을 가지는데, BIG_DECIMAL 구조체 변수 중 나누는 변수값인 A 변수값을 numerator라는 변수에 복사한다. 왜냐하면 함수를 실행하는 동안 나누는 값이 변하기 때문이다. 즉, 매개변수 A에 영향을 주지 않기 위함이다.

❺에서는 결과 값 result 변수의 메모리 공간을 할당한다. 최대 크기는 A 변수의 size 변수값에서 B 변수의 size 변수값을 뺀 것보다 하나 더 큰 공간이다. 예를 들어 '456789 / 123 = 3713' 연산에서 size 변수값을 계산해보면 456789의 size 변수값(digit 변수의 총 개수)은 6이고 123의 size 변수값은 3이므로 '6 - 3 = 3'이다. 그런데 결과 값 3713의 size 변수값은 4이므로 3에 1을 더하면 된다. for문에서는 결과 값 result 변수의 모든 digit 변수를 0으로 초기화해준다.

❻의 ptrForOrigin 변수는 연산을 위해 만들어진 numerator 변수의 digit 변수 메모리 공간을 함수 실행이 끝나기 전에 지우기 위해 사용한다. ⓭에서 연산이 끝나기 전 free() 함수로 힙 영역의 공간을 지워준다. 물론 이유는 메모리 누수를 방지하기 위해서다.

❼에서는 포인터 개념을 적용했다. 즉, 제일 처음에 나누는 값을 만드는 부분으로 numerator 변수의 digit 변수에 특정 값을 더해 포인터 위치를 오른쪽으로 옮긴 효과를 준다. 예를 들어 '3456789 / 12'라는 연산을 실행하면 3456789의 size 변수값(digit 변수의 총 개수)은 7이고 12의 size 변수값은 2이므로 5(=7-2)만큼 소수점(포인터 위치) 위치를 이동해 3456789라는 값을 34로 바꾼다. 포인터 위치만을 이동해 값에 변화를 주고, size 변수값을 바꿔준다. 결과적으로 numerator 변수는 새로운 BIG_DECIMAL 구조체 변수로 바뀌었다.

❽에서는 for문 안에서 나누기 연산의 가장 중요한 부분을 실행한다. 즉, 빼기 연산을 실행하면서 결과 값을 얻으며, for문을 실행하면서 나누는 수를 계속 변화시킨다. 나누는 수를 계속 변화시킨다는 것은 포인터 위치를 움직인다는 것과 같기 때문에, ⓬처럼 numerator 변수의 digit 변수값에서 1을 뺄수록 포인터 위치가 한 칸씩 왼쪽으로 움직이게 된다.

❾의 count 변수는 결과 값 result 변수의 digit 변수 하나에 값을 저장하는 데 사용한다. 0으로 초기화한 후 빼기 연산을 실행할 때마다 1씩 증가하게 된다.

❿에서는 numerator 변수가 B 변수보다 크면 빼기 연산을 계속 실행하면서 count 변수값을 하나씩 증가시킨다. MinusForDivide(&numerator, B); 구문이 중요한데, '참조에 의한 호출'로 &numerator라는 매개변수를 사용해 함수를 실행한 후 numerator 변수값은 변하게 된다. 나누기 연산에서 사용하는 빼기 연산을 실행하는 MinusForDivide() 함수는 ⓯에 구현했다.

⓫에서는 빼기 연산을 실행할 때마다 증가한 count 변수값을 결과 값 result 변수의 digit 변수 하나에 저장한다.

⓬에서는 numerator 변수값에 변화를 주는데, numerator 변수의 digit 변수값을 1만큼 감소시켜 포인터 위치를 옮겨주며 if문에서 size 변수값을 결정하는데, 최상위 digit 변수값이 0이 아니면 포인터 위치를 옮겼기 때문에 numerator 변수의 size 변수값을 1 증가시켜야 한다.

⓭에서는 연산을 위해 임시로 만들어 사용한 numerator 변수를 힙 영역에서 지워준다.

⓮는 결과 값인 result 변수값 앞쪽에 0이 있을 때 결과 값에서 삭제하는 연산이다. 예를 들어 결과 값이 01234라면 앞의 0을 삭제해 1234라는 결과 값을 얻는다. result.size = i + 1; 구문과 result.sign = 0; 구문으로 결과 값 result 변수의 크기와 부호를 결정한다.

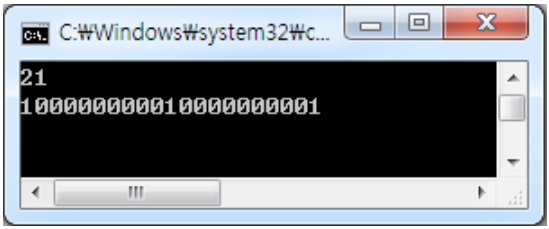
⓯는 나누기 연산만을 위해 사용하는 빼기 연산을 실행하는 MinusForDivide() 함수다. 함수를 실행한 후 매개변수 A는 값에 변화가 생긴다. 예를 들어 '234 - 123 = 111'이라는 연산을 실행하면, 매개변수 A는 처음에 234를 가지지만 함수 실행 종료 후 A 변수값은 111로 바뀐다('호출에 의한 참조'). 이 함수는 BIG_DECIMAL 구조체의 MINUS() 함수와 알고리즘이 같다. 단지 결과 값을 매개변수 A에 저장한다는 점만 다르다. 소스 코드를 자세히 알고 싶다면 코드 3-6 MINUS() 함수의 설명을 다시 한번 읽어보길 바란다.

나누기 연산자에서는 digit 변수 하나로 나누는 함수를 구현하지 않는데, 이유는 사용할 일이 없기 때문이다.

코드 3-14에서는 나누기 연산을 구현한 함수를 테스트한다.

코드 3-14 나누기 연산 테스트

```
int main()
{
    BIG_DECIMAL A, B, result;

    A = CreateDecimal((unsigned char *)"241", 3);
    B = CreateDecimal((unsigned char *)"11", 2);

    result = DIVIDE(&A, &B);
    printDecimal(result);

    A = CreateDecimal((unsigned char *)"123456789012345678901234567890", 30);
    B = CreateDecimal((unsigned char *)"1234567890", 10);

    result = DIVIDE(&A, &B);
    printDecimal(result);
}
```



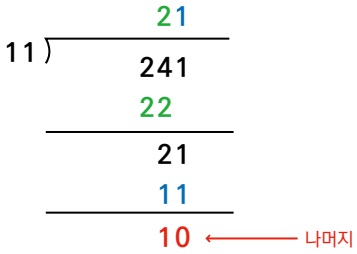
Big Number의 사칙 연산만으로도 재미있는 일을 많이 할 수 있다. 손으로 계산하기 싫은 큰 수를 컴퓨터가 알아서 계산해주니 생각하기에 따라서는 상당히 다양하게 응용해서 사용할 수 있다. 그리고 지금까지의 소스 코드를 이해하면서 직접 만들어본 독자라면 C/C++에 어느 정도 자신감이 생기는 부수적인 효과도 얻게 될 것으로 생각한다.

3.5 나머지 연산

나머지 연산(%)은 나누기 연산의 연장선에 있다. 알고리즘을 살펴보면 나누기 연산과 거의 같으며, 나눈 몫을 취하는 것이 아닌 연산의 마지막에 남는 나머지를 결과로 갖는다는 점만 다르다.

그림 3-12는 '241 % 11 = 10'이라는 연산의 실행 방법을 나타낸다.

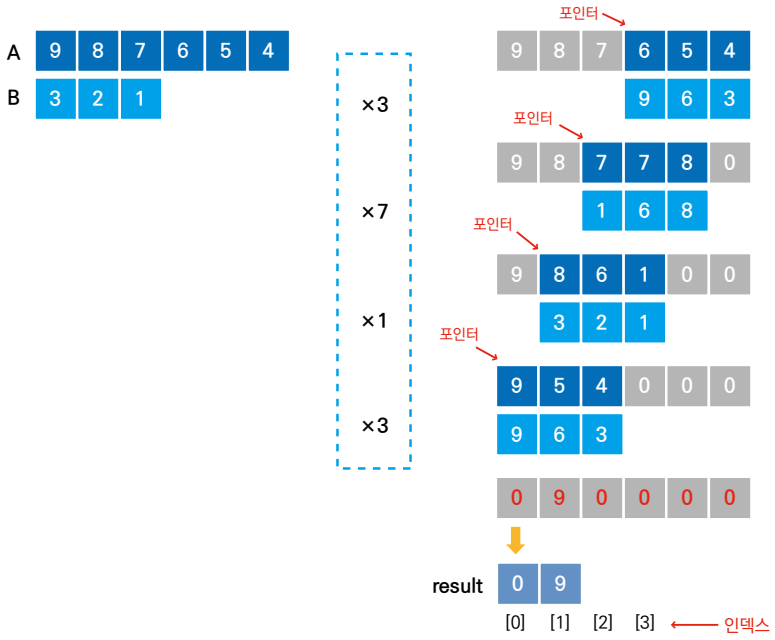
그림 3-12 나머지 연산



나머지 연산의 구현은 나누기 연산과 같다. 그림 3-13은 Big Number 구조체로 '456789 % 123 = 90'이라는 연산을 실행하는 방법을 설명한 것이다. 나누기 연산에서와 같은 방법으로 포인터로 수에 변화를 주어 나누기 연산을 실행한다. 단, 연산을 마지막까지 실행한 후에는 나누는 수인 B 변수보다 작은 수를 결과로 가진다.

> **NOTE** 지금까지 살펴본 나누기 연산은 쉬운 개념임에도 소스 코드로 구현할 때는 다소 까다로웠다. 그러나 포인터를 확실히 이해했다면 그다지 어렵지 않을 것으로 생각한다. C/C++를 자유롭게 사용한다는 것은 프로그램이 메모리 안에서 어떤 구조를 가지는지 이해하면서 포인터를 자유자재로 사용하는 순간이라 생각한다.

그림 3-13 BIG_DECIMAL 구조체의 나머지 연산 메모리 구조



다음에 그림 3-13을 어떻게 구현할지 순서대로 정리했다.

① 나누기 연산을 실행한다(기억이 나지 않으면 나누기 연산의 설명을 다시 읽어보기 바란다).

② 나누는 수는 메모리에서 나누는 수보다 작은 값으로 변환되었으며, 상위 digit 변수들은 0 값을 가진다. 따라서 BIG_DECIMAL 구조체 변수의 size 변수값(digit 변수의 총 개수)을 조정해, 0을 가진 상위 digit 변수들을 제외한 값을 결과로 반환한다. 즉, 000090의 결과에서 상위 digit 변수에 저장된 0을 제외한 90을 결과로 가진다.

나머지 연산도 나누기 연산과 마찬가지로 자릿수를 정확히 알고 프로그래밍해야

한다. 그리고 나누기 연산에서 구현한 MinusForDivide() 함수와 같은 알고리즘을 사용했다. 코드 3-15를 살펴보도록 하자.

코드 3-15 BIG_DECIMAL 구조체끼리 나머지 연산을 하는 MODULAR() 함수

```
BIG_DECIMAL MODULAR(BIG_DECIMAL *A, BIG_DECIMAL *M) // --- ❶
{
    BIG_DECIMAL result; // --- ❷
    int i; // --- ❸

    // --- ❹
    result.digit = (unsigned char *)malloc(A->size);
    for(i = 0; i < A->size; i++)
        result.digit[i] = A->digit[i];
    result.size = A->size;
    result.sign = 0;

    // --- ❺
    if(!IsBigger(A, M))
        return result;

    unsigned char* ptrForOrigin = result.digit; // --- ❻

    result.digit += (result.size - M->size);
    result.size = M->size;

    // --- ❼
    for(i = (A->size - M->size); i >= 0; i--)
    {
        while(IsBigger(&result, M))
            MinusForDivide(&result, M);

        result.digit--;

        if(result.digit[result.size] != 0)
```

```
            result.size++;
        }

        result.digit = ptrForOrigin; // --- ❽

        // --- ❾
        i = A->size - 1;

        while(i > 0)
        {
            if(result.digit[i] != 0x00)
                break;
            i--;
        }
        result.size = i+1;

        return result;
    }
```

---

❶에서는 매개변수로 BIG_DECIMAL 구조체 변수 A와 M을 입력받아 나머지 연산을 실행하는 MODULAR() 함수를 선언한다.

❷에서 결과 값을 저장하기 위한 result 변수는 나누는 매개변수 A와 같은 값으로 초기화한다.

❸의 i 변수는 인덱스로 사용한다.

❹는 나누는 result 변수를 초기화하는 구문으로 A 변수와 같은 값을 가지게 된다.

❺에서는 A 변수가 M 변수보다 작을 때 연산을 더 실행하지 않고 결과 값을 result 변수에 반환한다.

❻에서 연산을 실행하면 result 변수의 digit 변수는 변하게 된다. 즉, 그림 3-13

에서 설명한 포인터 개념이 적용되어 digit 변수값이 증가하거나 감소한다. 따라서 처음의 digit 변수값을 기억하기 위해 ptrForOrigin 변수를 사용하며, 연산이 끝난 후에는 ❽처럼 원래 digit 변수값을 result 변수에 저장하게 된다.

❼은 나누기 연산을 구현한 것으로, 자세한 설명은 코드 3-13에서 설명했으므로 생략한다.

❽에서는 나누기 연산을 종료한 후 결과 값 result 변수가 초기의 digit 변수값을 가진다. digit 변수는 힙 영역을 가리키며 힙 영역의 상위 digit들은 0 값을 가진다.

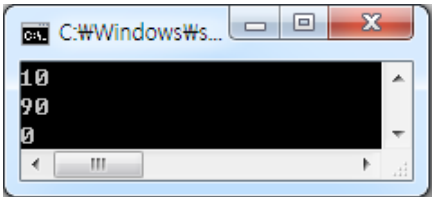
❾에서는 size 변수만 조정해 결과 값 result 변수의 앞부분에 있는 0을 없애준다. 예를 들어 결과 값은 000090을 가지므로 필요 없는 앞부분의 0을 지워주어 90으로 만든다.

코드 3-16은 나머지 연산을 구현한 함수를 테스트한다.

코드 3-16 나머지 연산 테스트

```
int main()
{
    BIG_DECIMAL A, B, result;

    A = CreateDecimal((unsigned char *)"241", 3);
    B = CreateDecimal((unsigned char *)"11", 2);

    result = MODULAR(&A, &B);
    printDecimal(result);

    A = CreateDecimal((unsigned char *)"456789", 6);
    B = CreateDecimal((unsigned char *)"123", 3);

    result = MODULAR(&A, &B);
    printDecimal(result);
```

```
    A = CreateDecimal((unsigned char *)"12345678901234567890123456789O", 30);
    B = CreateDecimal((unsigned char *)"1234567890", 10);

    result = MODULAR(&A, &B);
    printDecimal(result);
}
```



NOTE 사칙 연산 못지않게 많이 사용하는 것이 나머지 연산이다. 또한 암호학에서는 나머지 연산이 없으면 현대 암호학을 구현할 수 없다는 사실도 기억하자.

지금까지 설명한 연산들은 초등학교에서 배우는 간단한 연산이기에 내용을 이해하기 쉬웠으며 단지 프로그램으로 구현하는 방법이 다소 생각을 많이 하게 했을 것이다. 지금까지와 같은 방법으로 다음 장에서 다루게 되는 소수는 알고리즘을 이해한 후 사칙 연산의 코드를 가지고 소수 알고리즘을 구현한다.

뿐만 아니라 계속되는 고급 연산과 RSA도 지금까지의 방법과 동일하게 알고리즘을 먼저 이해한다면 어렵지 않게 코드를 구현할 수도 있다. 특히 고급 연산에서 지수 연산을 수행하게 되는데, 지수 연산을 하게 되면 결과 값이 기하급수적으로 커지게 되므로 Big Number 연산이 왜 유용한지를 알 수 있을 것이다.

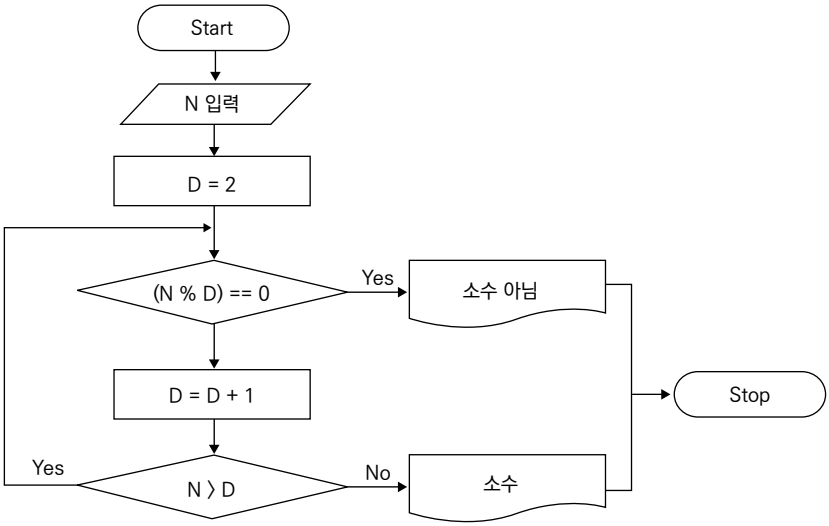
4 | 소수

소수Prime Number란 1과 그 수 자신 이외의 어떠한 자연수로도 나눌 수 없는 1보다 큰 양의 정수를 말한다. Big Number 연산에서는 5장에서 다루는 고급 연산을 이해하는 데 적합한 주제이기 때문에 소수를 다룰 것이다. 4장에서는 소수를 깊이 있게 다루지 않지만 Big Number 연산으로 소수를 판별하는 알고리즘을 비롯해 큰 소수를 구하는 방법을 간단히 다뤄보고자 한다.

4.1 소수 알고리즘

어떤 수가 소수인지를 판별하려면 그 수보다 작은 수로 계속 나눠보고 나머지가 0이 나오면 소수가 아닌 것으로 판단한다. 순서도Flow Chart로 살펴보면 다음과 같다.

그림 4-1 소수 판별 순서도 1



위 순서도를 이용하는 알고리즘을 구현해 소스 코드로 직접 구현해도 된다. 하지만 처리 속도 면에서 썩 좋은 알고리즘은 아니다. 왜냐하면 2로 나누어 소수인지 판별을 시도했으면 모든 짝수를 가지고 나머지 연산을 할 필요가 없어지는 등의 문제점이 있기 때문이다.

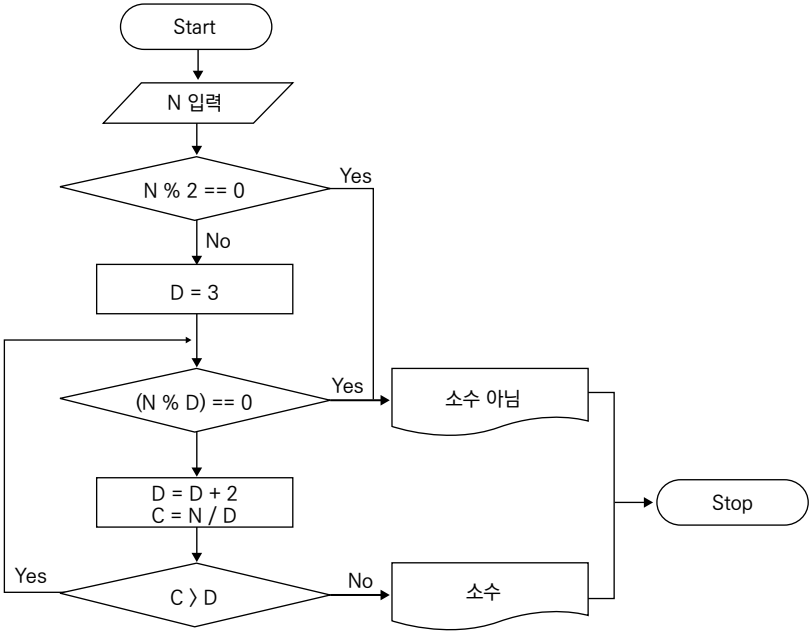
사실 가장 좋은 알고리즘은 판별하기 위해서는 N보다 작은 소수만으로 판단하는 것이 가장 빠르다. 즉, 19라는 수가 소수인지 판별하기 위해서는 19보다 작은 소수들인 {2, 3, 5, 7, 11, 13, 17}로만 나누는 것이 가장 빠르다는 의미다.

하지만 이런 방식은 N이 소수임을 알기 위해 N보다 작은 모든 소수를 알아야 하고, 모든 소수를 미리 데이터로 저장했어야 하며, N이 커질수록 많은 소수를 저장하는 데 상당히 많은 메모리가 필요하다.

그렇다면 N보다 작은 모든 소수를 나타내는, 나누는 값인 D는 어떻게 설정해야 효과적일까? 필자는 처음에 2로 나누어서 소수인지 판별한 다음 D 값을 홀수로만 설정해서 다시 판별을 시도했다.

또한 나누는 값인 D의 범위를 한정해 그림 4-2와 같은 소수 판별 알고리즘을 만들었다. 아마 필자가 만든 알고리즘보다 더 좋은 소수 판별 알고리즘이 있을 수도 있다. 따라서 더 빠르게 실행하고 싶은 독자라면 자신만의 새로운 알고리즘을 만들어 보기를 권장한다.

그림 4-2 소수 판별 순서도 2



소수를 판별하는 알고리즘이 좀 복잡해진 것 같이 보일 수도 있다. 나누는 수 D의 범위를 한정시키려고 C를 추가했으며 C는 D에 종속되어 유동적으로 값을 정하게 되기 때문이다.

소수를 판별하는 알고리즘은 속도를 우선시하는데, 이유는 수가 클 경우에는 소수를 판별하는 데 걸리는 시간이 상당히 길어지기 때문이다. 따라서 그림 4-2를 바탕으로 BIG_DECIMAL 구조체 변수로 소수를 판별하는 IsPrimeNumber() 함수를 코드 4-1과 같이 구현했다.

코드 4-1 BIG_DECIMAL 구조체 변수로 소수를 판별하는 IsPrimeNumber() 함수

```
bool IsPrimeNumber(BIG_DECIMAL *A) // --- ❶
{
    BIG_DECIMAL denominator, max, result; // --- ❷
    unsigned char *ptrForFree;

    // --- ❸
    if(A->size == 1 && A->digit[0] == 0x02)
        return true;

    // --- ❹
    if((A->digit[0] ^ 0x01) & 0x01)
        return false;

    denominator = CreateDecimal((unsigned char *)"3", 1); // --- ❺

    max = DIVIDE(A, &denominator); // --- ❻

    while(IsBigger(&max, &denominator)) // --- ❼
    {
        // --- ❽
        result = MODULAR(A, &denominator);

        if(result.size == 1 && result.digit[0] == 0)
        {
            free(result.digit);
            free(denominator.digit);
            free(max.digit);
            return false;
        }

        free(result.digit);

        ptrForFree = denominator.digit;
        denominator = PlusDigit(&denominator, 0x02); // --- ❾
```

```
            free(ptrForFree);

            ptrForFree = max.digit;
            max = DIVIDE(A, &denominator); // --- ❿
            free(ptrForFree);

        }
        free(denominator.digit);
        free(max.digit);

        return true;
    }
```

❶에서는 매개변수로 입력받은 BIG_DECIMAL 구조체 변수 A가 소수인지를 판별하는 IsPrimeNumber() 함수를 선언한다. 소수면 true를 반환한다

❷에서 denominator 변수는 나누는 수이며, max 변수는 나누는 수의 최대 범위를 한정하는 데 사용한다. result 변수는 소수를 판별하는 나머지 연산의 결과 값을 저장하게 된다.

❸에서는 판별하려는 값이 2면 true를 반환한다. 즉, A 변수가 2면 소수다.

❹에서는 A 변수가 짝수인지를 판별하여, 짝수라면 false를 반환한다. 짝수라면 최하위 digit 변수값이 0이므로 최하위 digit 변수값만을 판별해내어 소수가 아님을 판별할 수 있다. 예를 들어 1바이트의 비트값이 XXXXXXX0(X = 0 or 1)이라면 마지막 digit 변수값이 0이므로 짝수가 된다. 이때 XOR(^) 연산을 먼저 하면 'XXXXXXX0 ^ 00000001 = XXXXXXX1'이 되고 이어서 AND(&) 연산을 하면 'XXXXXXX1 & 00000001 = 00000001'이 되어 짝수라면 if문은 true 값을 가지게 된다.

❺에서는 나누는 값인 denominator 변수값을 3으로 초기화한다.

❻에서 max 변수값은 A 변수에서 denominator 변수를 나눈 결과 값을 가진다.

❼에서는 max 변수가 denominator 변수보다 크거나 같다면 while문을 실행한다. 이때 max 변수와 denominator 변수는 함수를 실행하는 동안 값이 항상 변한다는 것을 기억하자.

❽에서는 A 변수를 denominator 변수로 나누는데, 나머지가 0이면 A 변수는 소수가 아니다. MODULAR() 함수를 실행한 결과를 저장하는 result 변수는 나머지 값을 가지며, if문에서는 result 변수값이 0인지를 판별한다. if문 안의 내용은 false를 반환하기 전 임시로 사용한 메모리 공간을 삭제하는 구문이다.

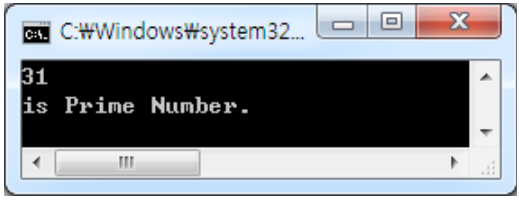
❾에서는 나누는 값인 denominator 변수값을 2만큼 증가시킨다. 앞서 ❺에서 denominator 변수값을 3으로 초기화했으므로 항상 홀수값을 가진다.

❿의 max 변수는 denominator 변수에 종속되어 항상 새로운 값을 가진다.

코드 4-2는 IsPrimeNumber() 함수를 테스트한다.

코드 4-2 소수 알고리즘 테스트

```
int main()
{
    BIG_DECIMAL A = CreateDecimal((unsigned char *)"31", 2);

    printDecimal(A);

    if(IsPrimeNumber(&A))
        printf("is Prime Number.\n");
    else
        printf("is Not Prime Number.\n");
}
```



소수인지 판별하기 위한 수가 상당히 크다면 연산 시간도 오래 걸린다. 예를 들면 어떠한 수를 판별하는 데 1개월이 걸릴 수도 있고, 심하면 1년 동안 연산을 실행할 수도 있다. 그나마 다행인 것은 컴퓨터가 그 일을 대신하고 우리는 전기세만 내면 된다는 사실이다.

그럼 소수를 판별하는 시간을 줄일 방법은 없을까? 기본적으로는 연산 속도를 빠르게 하면 되는데, 필자가 구현한 나누기와 나머지 연산은 빼기 연산을 바탕으로 하기에 아직은 불가능하다고 본다. 누군가가 나누기 연산을 빠르게 할 수 있는 알고리즘을 만든다면 역사에 길이 남을 일이 아닐까 싶을 정도다.

4.2 가장 큰 소수 구하기

소수 집합이 무한이라는 것은 수학적으로 증명되었지만, 많은 사람이 가장 큰 소수를 찾으려고 지금도 노력하고 있다.

지금부터는 큰 소수를 구하는 방법을 알아보도록 하자.

가장 큰 소수를 찾기 위해 사용하는 방법으로는 '메르센 수Mersenne Number'를 이용하는 것이 있다. 메르센 수는 2의 거듭제곱에서 1이 모자란 수로, 메르센 수 중에서 소수인 수를 '메르센 소수Mersenne Prime'라고 하며, 그림 4-3과 같은 식으로 구한다.

그림 4-3 메르센 소수의 수식

$$M_n = 2^n - 1 \quad (n\text{은 소수})$$

앞의 식에서 n이 소수라고 하더라도 항상 메르센 수가 소수는 아니다. 예를 들어 그림 4-4와 같은 결과가 나온다.

그림 4-4 메르센 소수의 예외

$M_n = 2^2-1 = 3(n=2)$

$M_n = 2^3-1 = 7(n=3)$

$M_n = 2^5-1 = 31(n=5)$

$M_n = 2^7-1 = 127(n=7)$

$M_n = 2^{11}-1 = 2047(n=11)$ //소수가 아니다. $2047 = 23 \times 89$

즉, n이 소수라고 해서 항상 M_n이 소수가 아니란 사실을 알 수 있는데, n이 11일 때 M_n은 2047이고 이 값은 인수분해할 수 있으므로 소수가 아니다.

이런 이유로 생각보다 많은 메르센 소수가 밝혀지지는 않았는데, 2011년까지 밝혀진 메르센 소수는 47개로 표 4-1과 같다.

표 4-1 메르센 소수 목록[01]

#	n	M_n	M_n의 자릿수	발견일(년)	발견자
1	2	3	1	고대	고대
2	3	7	1	고대	고대
3	5	31	2	고대	고대
4	7	127	3	고대	고대
5	13	8191	4	1456	아무개
6	17	131071	6	1588	피에트로 카탈디
7	19	524287	6	1588	피에트로 카탈디
8	31	2147483647	10	1772	레온하르트 오일러
9	61	2305843009213693951	19	1883	이반 미흐비치 페르부쉰
10	89	618970019…449562111	27	1911	R. E. Powers

01 출처: http://ko.wikipedia.org/wiki/메르센_소수

#	n	M_n	M_n의 자릿수	발견일(년)	발견자
11	107	162259276…010288127	33	1914	R. E. Powers
12	127	170141183…884105727	39	1876	에두아르 뤼카
13	521	686479766…115057151	157	1952	라파헬 로빈슨
14	607	531137992…031728127	183	1952	라파헬 로빈슨
15	1,279	104079321…168729087	386	1952	라파헬 로빈슨
16	2,203	147597991…697771007	664	1952	라파헬 로빈슨
17	2,281	446087557…132836351	687	1952	라파헬 로빈슨
18	3,217	259117086…909315071	969	1957	한스 리젤
19	4,253	190797007…350484991	1,281	1961	알렉산더 허비츠
20	4,423	285542542…608580607	1,332	1961	알렉산더 허비츠
21	9,689	478220278…225754111	2,917	1963	도널드 길리스
22	9,941	346088282…789463551	2,993	1963	도널드 길리스
23	11,213	281411201…696392191	3,376	1963	도널드 길리스
24	19,937	431542479…968041471	6,002	1971	브리언트 터커맨
25	21,701	448679166…511882751	6,533	1978	랜돈 커트 놀과 로라 니켈
26	23,209	402874115…779264511	6,987	1979	랜돈 커트 놀
27	44,497	854509824…011228671	13,395	1979	해리 넬슨과 데이빗 슬로빈스키
28	86,243	536927995…433438207	25,962	1982	데이빗 슬로빈스
29	110,503	521928313…465515007	33,265	1988	월크 콜킷과 루크 웰시
30	132,049	512740276…730061311	39,751	1983	데이빗 슬로빈스키
31	216,091	746093103…815528447	65,050	1985	데이빗 슬로빈스키
32	756,839	174135906…544677887	227,832	1992	데이빗 슬로빈스키와 폴 게이지
33	859,433	129498125…500142591	258,716	1994	데이빗 슬로빈스키와 폴 게이지
34	1,257,787	412245773…089366527	378,632	1996	데이빗 슬로빈스키와 폴 게이지
35	1,398,269	814717564…451315711	420,921	1996	GIMPS / 조엘 아르멩고
36	2,976,221	623340076…729201151	895,932	1997	GIMPS / 고든 스펜스
37	3,021,377	127411683…024694271	909,526	1998	GIMPS / 롤랜드 클락슨
38	6,972,593	437075744…924193791	2,098,960	1999	GIMPS / 난야 하이라트왈라
39	13,466,917	924947738…256259071	4,053,946	2001	GIMPS / 마이클 카메론
40	20,996,011	125976895…855682047	6,320,430	2003	GIMPS / 마이클 셰이퍼

#	n	M_n	M_n의 자릿수	발견일(년)	발견자
41*	24,036,583	299410429…733969407	7,235,733	2004	GIMPS / 조지 핀들리
42*	25,964,951	122164630…577077247	7,816,230	2005	GIMPS / 마르틴 노바크
43*	30,402,457	315416475…652943871	9,152,052	2005	GIMPS / 커티스 쿠퍼와 스티븐 분
44*	32,582,657	124575026…053967871	9,808,358	2006	GIMPS / 커티스 쿠퍼와 스티븐 분
45*	37,156,667	202254406…308220927	11,185,272	2008	GIMPS / 한스 미하엘 에브니히
46*	42,643,801	169873516…562314751	12,837,064	2009	GIMPS / 오드 마그너 스트린모
47*	43,112,609	316470269…697152511	12,978,189	2008	GIMPS / 에드슨 스미스
48*	57,885,161	581887266…724285951	17,425,170	2013	GIMPS / 커티스 쿠퍼

밝혀진 수 중에서 가장 큰 수는 $2^{57885161} - 1$인데, 소수인지 판별을 하지 않고 Big Number 연산으로 코드 4-3처럼 수를 계산하는 데도 일주일 이상 소요된다.[02] 필자는 Big Number 연산으로 이 수가 소수임을 판별하는 데, 적어도 1년 이상의 긴 연산 시간이 필요하다고 추측한다. 따라서 직접 연산을 실행하지는 않겠다.

코드 4-3 현재까지 알려진 가장 큰 메르센 소수를 출력하는 main() 함수

```
int main()
{
    // --- ❶
    BIG_DECIMAL A = CreateDecimal((unsigned char *)"2", 1);
    BIG_DECIMAL N = CreateDecimal((unsigned char *)"57885161", 8);
    BIG_DECIMAL one = CreateDecimal((unsigned char *)"1", 1);

    // --- ❷
    time_t now;
    time(&now);
```

02 참고로 필자 시스템의 CPU는 인텔 Core i5였으며 필자는 사정상 일주일 이상 연산을 실행할 수 없었다.

```
    printf("Starting...\n%s", ctime(&now));

    // --- ❸
    BIG_DECIMAL result = MINUS(&(MULTIPLY_EXPONENT(&A, &N)), &one);

    // --- ❹
    time(&now);
    printf("Calculation is Finished.\n%s", ctime(&now));

    // --- ❺
    FILE *fp;
    if((fp=fopen("result.txt", "wt")) == NULL)
        printf("file open error. \n");

    fprintDecimal(fp, result);

    fclose(fp);
}
```

❶에서는 세 개의 수 2, 57885161, 1을 각각 BIG_DECIMAL 구조체 변수 A, N, one에 저장한다.

❷에서는 연산 속도를 계산하기 위해 현재 시간을 출력한다.

❸에서는 지금까지 알려진 최대 메르센 소수인 '$2^{57885161}$ - 1'을 계산한다. MULTIPLY_EXPONENT()는 지수를 계산하는 함수로서 자세한 설명은 '6.2 지수 곱'에서 다룬다.

❹에서는 연산이 끝난 후의 현재 시간을 출력한다. 이처럼 현재 시간을 알아보는 이유는 연산 시간이 얼마나 걸리는지를 판단하기 위해서다.

❺에서는 결과를 파일로 출력한다. 결과 값이 상당히 큰 수이므로 콘솔에 출력하면 결과 값을 알아낼 수가 없기 때문이다.

위 메르센 수를 소수인지 판별하는 데는 상당히 오랜 시간이 걸린다. 이 책에 나와 있는 방법을 기준으로 생각한다면 몇 년 이상의 시간이 걸릴 수도 있다. 그러면 이 연산이 끝날 때까지 기다려야 할까? 그건 아니다. 코드 4-4와 같이 간단한 메르센 소수를 대상으로 판별하는 방법을 살펴보자.

코드 4-4 메르센 소수 판별

```
int main()
{
    BIG_DECIMAL A = CreateDecimal((unsigned char *)"2", 1);
    BIG_DECIMAL N = CreateDecimal((unsigned char *)"31", 2);
    BIG_DECIMAL one = CreateDecimal((unsigned char *)"1",  1);

    // 메르센 수를 계산한다
    BIG_DECIMAL result = MINUS(&(MULTIPLY_EXPONENT(&A, &N)), &one);

    if(IsPrimeNumber(&result)) // 메르센 수가 소수인지 판별한다
        printf("2^31 - 1 is Prime Number.\n");
    else
        printf("Not Prime Number.\n");
}
```

NOTE 어떤 수를 연산하고 판별하는 일은 상당한 인내를 요구하는 일로, 간단히 연산할 수 있으면 다행이지만 언제 연산을 완료하는지 모르는 상황에서 막연하게 기다려야 할 때가 더 많으니 지루한 것도 사실이다.

사실 큰 소수라는 주제는 필자도 찾을 엄두가 나지 않는다. 컴퓨터 한 대를 몇 년씩 켜놓아야 하니 상당한 부담이 되기 때문이다.이 책의 프로그램은 프로그래밍 이해라는 주제에 중점을 두었기 때문에 전문적으로 소수를 찾아내는 프로그램은 아니며 간단히 방법만을 논했다. 바람이라면 이 책을 읽고 소수에 관심이 생긴 독자들이 더 좋은 프로그램을 만들 수 있는 계기가 되었으면 한다.

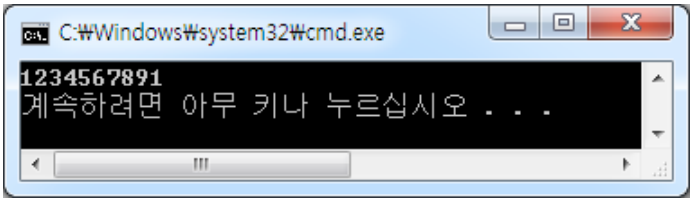
4.3 소수 알고리즘 테스트

이제 소수 알고리즘의 테스트를 진행해보자. 테스트할 때는 어떤 주어진 수보다 큰 소수를 찾는 방법을 사용하는데, 6장에서 RSA 공개 키 암호 구현 시 필요한 소수를 찾을 때 이 방법을 사용한다.

코드 4-5에서는 1234567890보다 큰 첫 번째 소수 하나를 찾는다.

코드 4-5 주어진 수보다 큰 첫 번째 소수 찾기

```
int main()
{
    // 134567890을 BIG_DECIMAL 구조체 변수 A에 저장한다
    BIG_DECIMAL A = CreateDecimal((unsigned char *)"1234567890", 10);

    for( ; ; )
    {
        // 소수인지를 판별한 후 소수이면 for문 실행을 종료한다
        if(IsPrimeNumber(&A))
            break;

        A = PlusDigit(&A, 1); // 수를 1 증가시킨다
    }

    printDecimal(A); // 소수를 출력한다
}
```

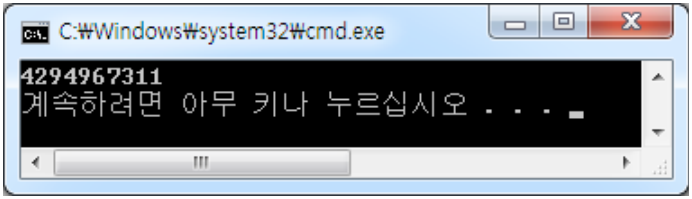


코드 4-6은 32비트로 표현할 수 있는 수보다 큰 첫 번째 소수를 찾는 방법으로 for 문을 사용한다. 예를 들면 다섯 번 실행하는 for문이기에 {4(=2×2), 16(=4×4), 256(=16×16), 65536(=256×256), 4294967296(=65536×65536)}이라는 연산을 실행한다.

즉, 코드 4-6은 2^{32}(=4294967296)보다 큰 소수를 찾는 방법을 구현한 예다(64비트로 표현할 수 있는 수를 찾으려면 5를 6으로 바꾸면 된다).

코드 4-6 32비트로 표현할 수 있는 수보다 큰 첫 번째 소수 찾기

```
int main()
{
    int i;
    BIG_DECIMAL A = CreateDecimal((unsigned char *)"2", 1);

    // 지수 곱을 계산하는 방법이다. 즉, A 변수가 2이므로
    // 2^32을 계산한 후 A 변수에 저장한다
    for(i = 0; i < 5; i++)
        A = MULTIPLY(&A, &A);

    // A 변수를 1씩 증가시키면서 첫 번째 소수를 찾는다
    for( ; ; )
    {
        if(IsPrimeNumber(&A))
            break;

        A = PlusDigit(&A, 1);
    }

    printDecimal(A);
}
```

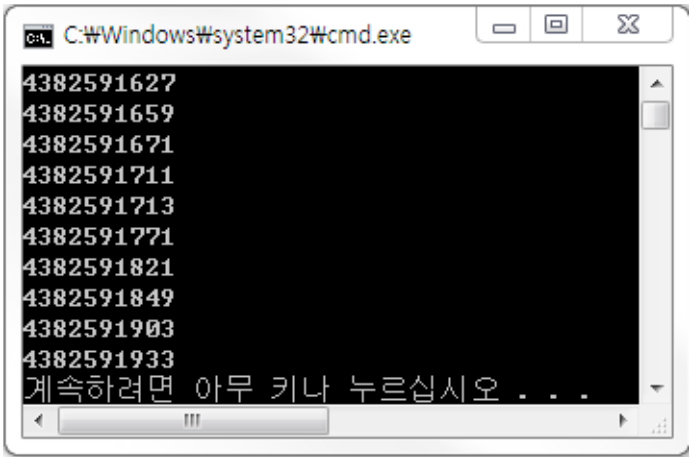


코드 4-7은 어떤 수(4382591627)보다 큰 소수 10개를 찾아 파일에 출력하는 방법을 보여주는 예로, 많은 소수를 한 번에 구할 수 있다. 참고로, 필자는 100만 개의 소수를 찾는 데 3시간 이상을 소요했다. 연산하는 시간 못지않게 파일에 출력하는 시간도 많이 소요되었기 때문이다.

코드 4-7 다수의 소수를 순차적으로 찾아서 파일에 출력하기

```
int main()
{
    // 4382591627을 BIG_DECIMAL 구조체 변수 A에 저장한다
    BIG_DECIMAL A = CreateDecimal((unsigned char *)"4382591627", 10);

    int i;
    unsigned char *ptrForFree;

    //  출력할 파일을 생성하고 연다
    FILE *fp;

    if((fp=fopen("prime.txt", "wt")) == NULL)
        printf("file open error. \n");

    // 소수 10개를 출력하려고 for문을 사용한다
    // 소수를 찾았을 때만 i를 증가시킨다
    for(i = 0; i < 10; )
    {
        // 소수인지를 판별해 소수면 파일에 출력하고 i를 증가시킨다
        if(IsPrimeNumber(&A))
```

```
            {
                fprintDecimal(fp, A);
                i++;
            }

            // PlusDigit() 함수에서 A 변수를 1 증가시키며, ptrForFree 포인터
            // 변수를 사용해 필요 없는 이전 A 변수값을 힙 영역에서 삭제한다
            ptrForFree = A.digit;
            A = PlusDigit(&A, 1);
            free(ptrForFree);
        }

        fclose(fp); // 출력한 파일을 닫는다

        return 0;
    }
```



5 | 고급 연산

5장에서는 지수 연산Exponential Calculation과 지수를 가진 나머지 연산을 다룬다. 또한 지수 계산을 할 때 사용하는 2진수의 값을 처리하는, BIG_DECIMAL과 BIG_BINARY 구조체 사이의 진수 변환을 하는 코드를 구현할 것이다. 마지막에는 인수분해를 어떻게 하는지 알아볼 것이다.

이러한 고급 연산은 주로 통계에 기반을 둔 다양한 데이터 분석에 응용해서 사용할 수 있으므로 데이터 분석에 관심이 있는 독자라면 꼭 살펴보기 바란다. 또한 5장에서 다루는 연산을 이해하고 나면 필요한 연산을 직접 만들어 사용할 수 있는 힘을 갖게 될 것이다.

5.1 진수 변환

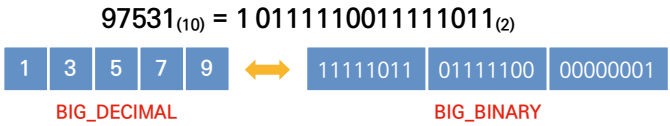
연산할 때 중요한 점은 정확성이다. 항상 정확한 결과 값을 얻어야 하기 때문이다. 다음으로 중요한 점은 연산 속도다. 그리고 속도는 연산을 어떻게 구현하느냐에 따라 달라지므로 알고리즘이 상당히 중요하다.

예를 들어 지수Exponent를 가진 수의 연산을 생각해보자. 일반적으로는 12345^{97}을 계산할 때 12345를 97번 곱해 결과를 얻을 수 있지만 97은 소수이므로 2진수를 이용해 좀 더 빠르게 연산을 실행하는 알고리즘이 존재한다. 예를 들어 97을 2진수로 변환하면 $1100001_{(2)}$이 되는데, 2진수라면 곱하기 연산을 아홉 번만 실행해 결과를 얻을 수 있다(이 알고리즘은 '5.2 지수 곱'에서 자세히 다룬다).

즉, 지수를 다루는 연산 속도를 향상시키려고 2진수를 사용하며, 2진수를 사용하기 위해 BIG_DECIMAL과 BIG_BINARY 구조체 사이의 진수 변환을 구현해야 한다.

그림 5-1은 10진수와 2진수 사이의 진수 변환Antilogarithm을 나타낸 그림이다.

그림 5-1 진수 변환



먼저 BIG_BINARY 구조체를 BIG_DECIMAL 구조체로 변환해보자. 쉽게 이해하기 위해서 $1101001_{(2)}$을 105로 변환한다고 가정하면 그림 5-2와 같다.

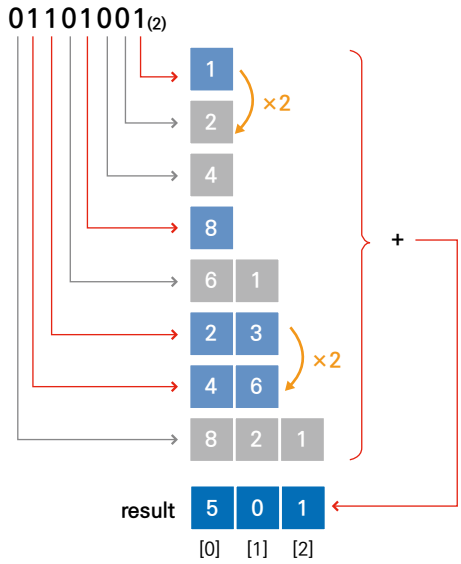
그림 5-2 $1101001_{(2)}$을 105로 변환

$$1101001_{(2)} = 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times 2^0$$
$$= 64 + 32 + 8 + 1$$
$$= 105$$

그럼 위 식을 코드로 구현하면 어떻게 될까? 가장 하위 비트인 2^0부터 계산하면서 한 비트씩 자리를 이동할 때마다 2를 곱해 값을 가진 뒤, 해당 비트 값이 0이면 무시하고 1이면 결과 값(result 변수값)에 2를 곱한 값을 더하면 된다.

그림 5-3을 보면 BIG_DECIMAL 구조체는 1부터 시작해 아래 방향으로 차례로 2를 곱한다. 그리고 2진수에서 해당 비트가 1인 모든 값을 더해 결과 값을 얻는다.

그림 5-3 2진수에서 10진수로 변환할 때의 메모리 구조



코드 5-1은 위 그림의 알고리즘을 구현한 것이다.

코드 5-1 BIG_BINARY 구조체를 BIG_DECIMAL 구조체로 변환하는 GetDecimal() 함수

```
BIG_DECIMAL GetDecimal(BIG_BINARY *binary) // --- ❶
{
    int i, j;
    unsigned char flag, *ptrForFree; // --- ❷

    // --- ❸
    BIG_DECIMAL decimal, temp;

    decimal = CreateDecimal((unsigned char *)"0", 1);
    temp    = CreateDecimal((unsigned char *)"1", 1);
```

```
    for(i = 0; i < binary->size; i++) // --- ❹
    {
        flag = 0x01; // --- ❺

        for(j = 0; j < 8; j++) // --- ❻
        {
            // --- ❼
            if(binary->byte[i] & flag)
            {
                ptrForFree = decimal.digit;
                decimal = PLUS(&decimal, &temp);
                free(ptrForFree);
            }
            flag <<= 1; // --- ❽

            // --- ❾
            ptrForFree = temp.digit;
            temp = MultiplyDigit(&temp, 0x02);
            free(ptrForFree);
        }
    }

    return decimal;
}
```

❶은 BIG_BINARY 구조체 변수 binary를 매개변수로 입력받아 BIG_DECIMAL 구조체 변수로 변환하는 GetDecimal() 함수를 선언한다.

❷에서 flag 변수는 특정 비트값을 알아내려고 사용하며, ptrForFree 포인터 변수는 힙 영역에 할당한 공간을 지우려고 사용한다.

❸에서 decimal 변수는 결과 값을 저장하며 0으로 초기화한다. 그리고 temp 변수는 1로 초기화하는데, 초기화한 후에는 2를 순차적으로 곱해 증가하게 된다. 즉, temp 변수값은 '1, 2, 4, 8, 16, 32, 64, 128, 256, ……'이라는 값을 순차적으로 가진다.

❹는 매개변수로 입력받은 binary 변수(2진수)의 최하위 size 변수값부터 시작해 모든 size 변수값을 검사해 계산한다.

❺에서 비트값을 확인하려고 사용하는 flag 변수의 값을 00000001로 초기화한다. 이후에 ❽에서 한 비트씩 왼쪽으로 이동하면서 8개 비트를 모두 검사한다.

❻에서는 1바이트가 8비트이므로 for문이 8번을 실행해서 검사하도록 설정한다.

❼에서는 flag 변수의 bytes 변수값이 1인 위치에 있는 값을 검사한다. Binary->byte[i]의 해당 변수값이 0이면 값을 더하지 않고, 1이면 if문 안 연산을 실행한다. 이때 PLUS() 함수에서는 결과 값 decimal 변수값에 두 배씩 증가하는 temp 변수값을 더한다. 또한 메모리를 낭비하지 않기 위해 free() 함수로 사용하지 않는 공간을 힙 영역에서 지운다.

❽에서는 flag 변수값을 한 비트 이동시켜 상위 비트를 순차로 검사한다.

❾에서는 MultiplyDigit() 함수로 temp 변수값을 두 배 증가시킨다. 그리고 ❼에서와 마찬가지로 free() 함수로 곱하기 연산을 실행한 뒤 필요 없는 메모리 공간을 삭제한다.

이번에는 BIG_DECIMAL 구조체를 BIG_BINARY 구조체로 변환하는 방법을 알아보기로 하자. 10진수를 2진수로 만드는 법은 10진수 값을 2로 나눈 후 나머지를 왼쪽에서부터 순서대로 쓰면 2진수가 된다.

그림 5-4는 10진수를 2진수로 바꾸는 방법을 보여준다.

그림 5-4 10진수를 2진수로 바꾸는 연산 구조

(10진수) (나머지)

12345 / 2 1
6172 / 2 0
3086 / 2 0
1543 / 2 1
771 / 2 1
358 / 2 1
192 / 2 0
96 / 2 0
48 / 2 0
24 / 2 0
12 / 2 0
6 / 2 0
3 / 2 1
1 / 2 .....1 **결과 값: $11000000111001_{(10)}$**

위 계산 방법을 살펴보면 진수 변환을 쉽게 이해할 수 있다. 하지만 Big Number에서는 바이트 단위로 2진수를 표현하므로 8비트에서 표현할 수 있는 수인 2^8(256)으로 나눈 나머지를 이용해 결과 값을 얻는다. 즉, 위 연산은 그림 5-5와 같은 방법으로 계산되며 결과는 같다.

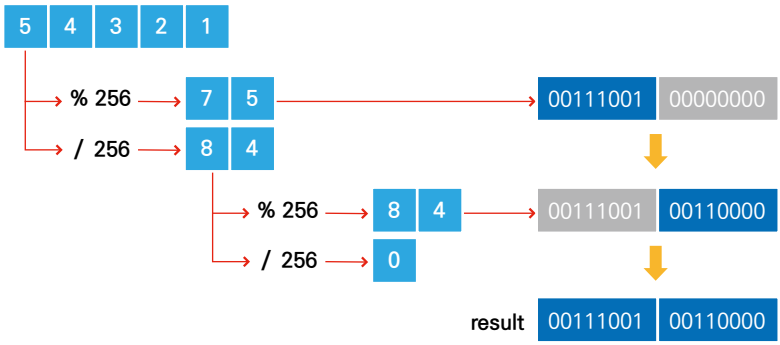
그림 5-5 2^8으로 나눈 나머지로 2진수를 구하는 연산 구조

12345 / 256 .....57($00111001_{(2)}$)
48 / 256 .....48($00110000_{(2)}$) **결과 값: $11000000111001_{(2)}$**

위 그림의 연산은 10진수를 256진수로 바꾼 다음 256진수를 2진수로 다시 한번 바꾼 것이라고도 볼 수 있다. 그런데 256진수의 값은 자연스럽게 2진수의 값으로 저장되기에 추가적인 연산이 필요하지 않다는 장점이 있다. 즉, 10진수를 256진수로 바꾸기만 하면 BIG_DECIMAL 구조체에서 BIG_BINARY 구조체로의 진수 변환이 이루어진다고 보면 된다.

그림 5-6은 Big Number의 구조체 안에서 어떻게 연산이 실행되는지를 보여준다. 참고로 이 책에서 설명하는 Big Number 연산에서는 10진수를 2진수로 변환할 때 256으로 나눈 나머지를 이용해 결과 값을 얻는다.

그림 5-6 Big Number 연산에서 10진수 → 2진수로 변환했을 때의 메모리 구조



나누기와 나머지 연산만을 이용해 진수 변환이 이루어짐을 알 수 있다.

그럼 코드 5-2를 살펴보자.

코드 5-2 BIG_DECIMAL 구조체를 BIG_BINARY 구조체로 변환하는 GetBinary() 함수

```
BIG_BINARY GetBinary(BIG_DECIMAL *decimal) // --- ❶
{
    int i, j, position; // --- ❷
```

```
    BIG_BINARY binary; // --- ❸
    BIG_DECIMAL numerator, denominator, remainder, zero; // --- ❹

    binary.byte = (unsigned char *)malloc((int)(decimal->size / 2) + 1); //
--- ❺

    // --- ❻
    numerator.digit = (unsigned char *)malloc(decimal->size);
    numerator.size = decimal->size;
    for(i = 0; i < numerator.size; i++)
        numerator.digit[i] = decimal->digit[i];

    zero = CreateDecimal((unsigned char *)"0", 1); // --- ❼

    denominator = CreateDecimal((unsigned char *)"256", 3); // --- ❽

    for(position = 0; ; position++) // --- ❾
    {
        remainder = MODULAR(&numerator, &denominator); // --- ❿

        // --- ⓫
        void *ptrDigit = (void *)numerator.digit;
        numerator = DIVIDE(&numerator, &denominator);
        free(ptrDigit);

        binary.byte[position] = 0x00; // --- ⓬

        // --- ⓭
        for(i = 0; i < remainder.size; i++)
        {
            unsigned char tempMultiply = 1;

            for(j = 0; j < i; j++)
```

```
            tempMultiply *= 10;

            binary.byte[position] += remainder.digit[i] * tempMultiply;
        }

        // --- ⓮
        if(IsEqual(&numerator, &zero))
            break;
    }

    binary.size = position + 1; // --- ⓯

    // --- ⓰
    free(numerator.digit);
    free(denominator.digit);
    free(remainder.digit);
    free(zero.digit);

    return binary;
}
```

❶에서는 BIG_DECIMAL 구조체 변수 decimal을 매개변수로 입력받아 BIG_BINARY 구조체로 변환한 결과 값을 BIG_BINARY 구조체 변수 binary에 저장하는 GetBinary() 함수를 선언한다.

❷에서 position 변수는 BIG_BINARY 구조체 변수 binary의 byte 변수를 위한 인덱스로 사용한다. ❾에서 position 변수값을 0으로 설정해서 for문 실행을 시작하는데, 이는 결과 값을 가지는 binary.byte[0]부터 결과 값을 저장하기 위함이다.

❸의 binary는 결과 값을 저장하는 변수다.

❹는 연산에서 사용하는 변수들이다. numerator 변수는 나누는 값을 저장하고,

denominator 변수는 나누는 값으로 256을 가지며, remainder 변수는 나머지를 저장하고, zero 변수는 0 값을 저장하려고 각각 사용한다.

❺에서는 결과 값을 저장할 binary 변수의 메모리 공간을 할당한다.

❻에서는 함수 실행을 끝낸 후 매개변수 decimal 값에 변화를 주지 않으려고 나누는 변수 numerator에 decimal 변수값을 복사한다.

❼에서는 zero 변수값으로 0을 할당하는데, ⓮에서 나누는 값이 0인지를 확인하려고 사용한다.

❽에서는 나누는 변수 denominator에 256을 할당한다.

❾에서는 결과 값을 가지는 binary 변수의 최하위 byte 변수부터 차례대로 값을 저장하므로, 인덱스인 position 변수는 0부터 순차적으로 증가한다. 결과 값의 크기를 알 수 없으므로 for문의 조건식을 생략했으며, for문의 분기는 ⓮에서 이루어진다.

❿에서는 나머지를 계산해 remainder 변수에 저장한다.

⓫에서는 나누기 연산을 실행하는데, 이때 numerator 변수값은 변한다. 나누기가 끝나면 free() 함수로 numerator 변수의 메모리 공간을 지운다.

⓬에서는 계산한 값을 저장할 byte 변수를 0으로 초기화한다.

⓭에서는 10진수를 256으로 나눈 나머지를 byte 변수에 저장한다. 예를 들어 나머지가 48이라면 byte 변수에는 비트값인 00110000을 저장한다. 구문이 다소 복잡한데, 그 이유는 48이라는 값을 이루는 4와 8을 BIG_DECIMAL 구조체 변수 remainder의 digit 변수 안에 따로 저장했기 때문이다. tempMultiply 변수는 1, 10, 100이라는 값을 저장하려고 사용하는데, BIG_DECIMAL 구조

체 변수 remainder의 digit 변수에 따라 tempMultiply 변수값이 결정된다. 즉, tempMultiply 변수값을 remainder 변수의 digit 변수값에 곱하여 remainder 변수값을 실제 계산할 수 있는 값으로 바꾼다. 예를 들면 '4 × 10 + 8 × 1'이라는 연산을 실행한다.

⓮에서는 나누는 값인 numerator 변수값이 0이면 for문 실행을 종료한다.

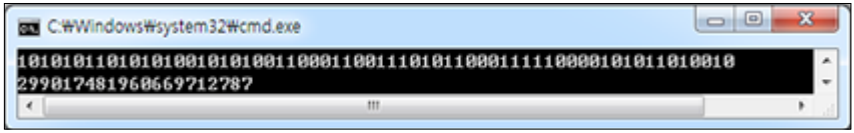
⓯에서는 결과 값 binary 변수의 size 변수값을 정한다.

⓰에서는 GetBinary() 함수 연산을 위해 만든 변수들을 힙 영역에서 지운다.

코드 5-3은 진수 변환을 위해 구현한 GetDecimal() 함수와 GetBinary() 함수를 테스트한다.

코드 5-3 진수 변환 테스트

```
int main()
{
    BIG_DECIMAL A;
    BIG_BINARY B;

    A = CreateDecimal((unsigned char *)"12345678901234567890", 20);
    B = GetBinary(&A);
    printBinary(B);

    unsigned char data[9] =
        {0x10, 0x35, 0xb3, 0x7f, 0x94, 0xa9, 0x7e, 0x11, 0x93};

    B = CreateBinary(data, sizeof(data));
    A = GetDecimal(&B);
    printDecimal(A);
}
```



지금까지 설명한 2진수와 10진수의 변환을 이해했다면 어떠한 진수든 변환할 수 있을 것이다.

> **NOTE** 컴퓨터를 만든 이유 중 하나는 수학 연산을 빨리 처리하기 위함이다. 다르게 표현하자면 수학 연산의 많은 부분을 컴퓨터로 계산할 수 있다는 뜻이다. 진수 변환도 그 중 하나일 뿐이다.

암호학에서 사용되는 키Key값은 보통 큰 2진수 값으로 주어진다. 따라서 진수 변환은 6장에서 요긴하게 사용할 수 있다. 즉, 어떤 2진수를 10진수로 변환해 계산하고 결과 값을 다시 2진수로 변환하는 과정으로, 공개 키 암호 방식에서 Big Number 연산을 유용하게 사용할 수 있다.

5.2 지수 곱

지수 곱Multiply Exponent은 보통 지수 부분을 인수분해한 후에 계산한다. 그런데 결과 값이 기하급수적으로 커지기 때문에 손으로 계산하는 데는 한계가 있다.

그림 5-7은 지수인 30을 인수분해한 후 2^{30}을 계산하는 일반적인 방법이다.

그림 5-7 지수의 인수분해를 이용한 방법

$$2^{30} = ((2^2)^3)^5$$
$$= (4^3)^5$$
$$= 64^5$$
$$= 1073741824$$

그럼 컴퓨터를 이용해 지수의 인수분해를 이용한 지수 곱을 구현하려면 어떻게 해야 할까? 인수분해해야 하는 함수를 먼저 구현해야 하는데, 속도면에서 좋은 방법은 아니다. 더욱이 2^{97}처럼 지수가 소수라면 인수분해가 불가능하므로 어쩔 수 없이 97번의 곱하기 연산을 실행해야만 한다.

그렇다면 효과적인 방법은 무엇일까? 진수 변환 때와 마찬가지로 2진수를 이용하는 방법이 효과적이다. 2진수를 이용하면 인수분해라는 불필요한 작업을 안 해도 되기 때문이다.

그림 5-8은 2^{30}을 2진수를 이용해 계산하는 방법으로 지수 부분을 2진수로 바꾸었다. 즉, 30을 $11110_{(2)}$로 바꾼 후 계산한다.

그림 5-8 2진수를 이용해 지수를 인수분해하는 방법

$$2^{30} = 2^{11110_{(2)}}$$
$$= 2^{10000_{(2)}} \times 2^{1000_{(2)}} \times 2^{100_{(2)}} \times 2^{0}$$
$$= 2^{16} \times 2^{8} \times 2^{4} \times 2^{2} \times 2^{0}$$
$$= 1073741824$$

위 식만을 보면 어차피 지수 연산을 해야 한다. 하지만 지수 부분이 2의 배수인 것에 착안하면 지수 연산을 빨리하는 방법을 찾을 수 있다. 즉, 지수가 '2, 4, 8, 16, ……'로 두 배씩 증가하는 집합이면 지수 계산을 낮은 지수의 결과 값에 종속하는 형태로 계산할 수 있음을 알 수 있다.

그림 5-9는 이를 설명한다.

그림 5-9 2진수로 이루어진 지수를 계산하는 방법

$2^1 = 2$

$2^2 = 2^1 \times 2^1 = 4$

$2^4 = 2^2 \times 2^2 = 4 \times 4 = 16$

$2^8 = 2^4 \times 2^4 = 16 \times 16 = 256$

$2^{16} = 2^8 \times 2^8 = 256 \times 256 = 65536$

$2^{32} = 2^{16} \times 2^{16} = ...$

.....

그림을 살펴보면 지수가 2의 배수로 증가하면 어떤 값을 계산할 때 바로 이전 값을 서로 곱하면 된다는 사실을 알 수 있다.

그림 5-10은 이를 일반화한 형태로 임의의 수 N에 아래와 같은 연산을 할 수 있다.

그림 5-10 2진수 지수 계산의 일반화

$N^1 = N$

$N^2 = N^1 \times N^1$

$N^4 = N^2 \times N^2$

$N^8 = N^4 \times N^4$

$N^{16} = N^8 \times N^8$

$N^{32} = N^{16} \times N^{16}$

.....

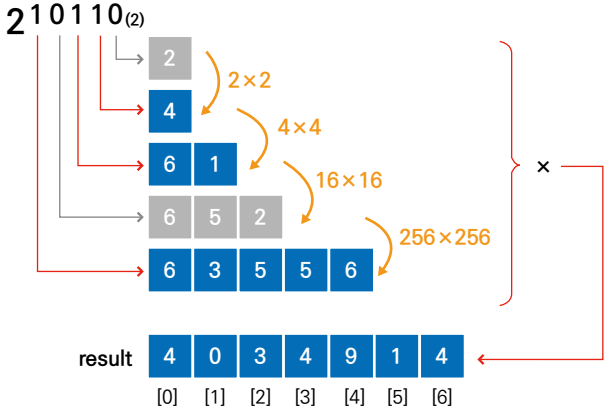
$N^{2K} = N^K \times N^K \quad (K = 1, 2, 4, 8, 16, 32, 64, ...)$

이처럼 지수를 2진수로 만든 후에 지수 연산을 하면 빠른 연산이 가능해진다. 그럼 이를 프로그래밍 알고리즘으로 바꿔보면 어떨까? 어떠한 수를 계속해서 서로 곱하면서 지수 부분의 해당 비트가 1이라면 결과 값에 그 수를 곱해주고, 0이라면 무시하는 방식으로 구현할 수 있다.

예를 들어 2^{22}을 계산할 때, 22를 2진수로 변환하면 $10110_{(2)}$이 된다. 그럼 결과 값 result 변수값을 1로 초기화한 후 2진수 지수의 최하위값을 확인하면 $0(10110_{(2)})$이므로 2를 곱해 줄 필요가 없다. 다음 값은 $1(10110_{(2)})$이므로 4(=2×2)를 result 변수에 곱해준다. 이런 방식으로 살펴보면 다음 $1(10110_{(2)})$은 16(=4×4)을 result 변수에 곱해야 하고, $0(10110_{(2)})$은 256(=16×16)을 곱하지 않는다.

여기서 꼭 기억해야 할 점은 어떠한 수를 결과 값 result 변수에 곱하지 않더라도 하나의 수를 서로 곱하는 과정은 다음 연산을 위해 계속 진행해야 한다는 것이다. 즉, 256(=16×16)을 결과 값 result 변수에 곱하지 않더라도 다음 과정인 $1(10110_{(2)})$에서 곱하는 65536(=256×256)을 위해서 꼭 연산해야 한다. 그림 5-11은 Big Number 구조체에서 어떻게 연산이 이뤄지는지를 보여준다.

그림 5-11 2진수 지수를 이용한 Big Number 지수 곱의 메모리 구조



그럼 그림 5-11의 Big Number 지수 연산을 어떻게 구현하면 될까? BIG_BINARY 구조체를 BIG_DECIMAL 구조체로 변환하는 진수 변환과 비슷하므로 진수 변환 알고리즘과 상당 부분 일치할 것이다. 코드 5-4를 살펴보도록 하자.

코드 5-4 BIG_DECIMAL 구조체끼리의 지수 곱을 위한 MULTIPLY_EXPONENT() 함수

```
BIG_DECIMAL MULTIPLY_EXPONENT(BIG_DECIMAL *A, BIG_DECIMAL *E) // --- ❶
{
    int i, j;
    unsigned char flag, *ptrForFree; // --- ❷
    BIG_DECIMAL result, temp; // --- ❸

    BIG_BINARY binaryE = GetBinary(E); // --- ❹

    // --- ❺
    result = CreateDecimal((unsigned char *)"1", 1);
    temp = MultiplyDigit(A, 1);

    for(i = 0; i < binaryE.size; i++) // --- ❻
    {
        flag = 0x01; // --- ❼

        for(j = 0; j < 8; j++) // --- ❽
        {
            // --- ❾
            if(binaryE.byte[i] & flag)
            {
                ptrForFree = result.digit;
                result = MULTIPLY(&result, &temp);
                free(ptrForFree);
            }

            // --- ❿
            ptrForFree = temp.digit;
            temp = MULTIPLY(&temp, &temp);
            free(ptrForFree);
```

```
            flag <<= 1; // --- ⓫
        }
    }

    return result;
}
```

❶에서는 BIG_DECIMAL 구조체 변수 A와 E를 매개변수로 입력받아 E 변수를 지수로 A^E 연산을 실행하는 MULTIPLY_EXPONENT() 함수를 선언한다.

❷에서 flag 변수는 비트값을 확인하기 위해 사용하며, ptrForFree 포인터 변수는 힙 영역에서 필요 없는 공간을 지우기 위해 사용한다.

❸에서 BIG_DECIMAL 구조체 변수 두 개를 정의하는데, result는 결과 값을 저장하고 temp 변수는 연산 중에 사용하는 임시 변수로 두 수를 두 번 곱한 값을 저장한다.

❹에서는 지수로 사용하는 E 변수를 2진수로 변환해 BIG_BINARY 구조체 변수 binaryE에 저장한다.

❺에서는 result 변수를 1로 초기화하고, temp 변수는 A 변수값으로 초기화한다. temp 변수를 초기화하기 위해 메모리 공간을 할당한 후 A 변수값을 복사할 수도 있지만 코드를 간결하게 하기 위해 A 변수에 1을 곱한 값을 갖게 했다.

❻에서는 지수로 사용하는 binaryE 변수의 bytes 변수 개수만큼 for문을 실행한다.

❼에서는 1바이트 크기의 비트값인 flag 변수값을 가장 오른쪽의 한 비트만 1을 가진 00000001로 초기화한다. 이후 1이 왼쪽으로 한 비트씩 이동하면서 binaryE 변수의 비트값을 검사하는 데 사용한다.

❽에서는 1바이트에 8개 비트가 있으므로 for문을 8번 반복하게 된다.

❾에서는 binaryE 변수의 비트값 각각을 확인해서 1이면 결과 값 result 변수에 temp 변수값을 곱하고, 0이면 아무런 연산도 하지 않는다. 곱하기 연산 후에는 free() 함수로 필요하지 않은 메모리 공간을 지운다.

❿에서는 이전 값을 두 번 곱해 temp 변수에 저장한다. ❾의 MULTIPLY() 함수에서 곱하기 연산을 실행할 때 사용할 temp 변수값을 만들기 위해서다.

⓫에서는 지수 비트를 확인하기 위해 사용하는 flag 변수값의 비트를 왼쪽으로 하나 이동시킨다. 즉, flag 변수값이 00000001이라면 00000010으로 1의 위치가 변한다.

그런데 코드 5-4는 속도 면에서 한 가지 문제가 있다. 바이트의 모든 비트에 1이 있는지를 확인하기 때문이다. 예를 들어 지수 부분의 최상위 바이트 비트값이 00000001이라고 가정하면 비트값이 1인 비트에서 지수 연산을 끝내야 함에도 위에서 구현한 알고리즘은 나머지 7개의 0인 비트를 확인한 후에야 연산을 끝내게 된다. 일반 자료형 연산이면 큰 문제가 없으나 Big Number 연산에서 위 알고리즘은 문제가 된다. 이를 해결하려면 지수 부분 바이트의 비트값에서 마지막으로 1 값을 가진 위치를 알아내 해당 비트까지 연산한 후에는 연산이 끝나도록 해야 한다.

따라서 코드 5-5는 지수 부분 최상위 바이트의 비트값에서 마지막으로 1 값을 가진 위치를 알아내는 코드를 추가했다.

코드 5-5 처리 속도를 향상시킨 MULTIPLY_EXPONENT() 함수

```
BIG_DECIMAL MULTIPLY_EXPONENT(BIG_DECIMAL* A, BIG_DECIMAL* E)
{
    int i, j, position;
    unsigned char flag, *ptrForFree;
```

```
BIG_DECIMAL result, temp;

BIG_BINARY binaryE = GetBinary(E);

result = CreateDecimal((unsigned char*)"1",1);
temp = MultiplyDigit(A, 1);

// --- ❶
position = 8 * (binaryE.size - 1);

j = 8;
flag = 0x80;

for(i = 0; i < 8; i++)
{
    if(binaryE.byte[binaryE.size - 1] & flag)
    {
        position += j;
        break;
    }

    j--;
    flag >>= 1;
}

for(i = 0; i < binaryE.size; i++)
{
    flag = 0x01;

    for(j = 0; j < 8; j++)
    {
        if(binaryE.byte[i] & flag)
```

```
            {
                ptrForFree = result.digit;
                result = MULTIPLY(&result, &temp);
                free(ptrForFree);
            }

            // --- ❷
            position--;
            if(position == 0)
                break;

            ptrForFree = temp.digit;
            temp = MULTIPLY(&temp, &temp);
            free(ptrForFree);

            flag <<= 1;
        }
    }

    return result;
}
```

알고리즘은 코드 5-4와 같으므로 추가 항목만 설명한다.

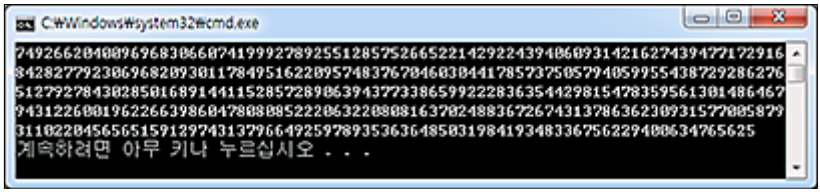
❶에서 position 변수는 BIG_BINARY 구조체 변수 binaryE의 최상위 바이트에서 마지막으로 1 값을 가진 비트 위치를 저장하는 변수다. 예를 들어 2진수 값이 $1000101010111110000_{(2)}$처럼 3바이트(11110000 01010101 00000100)라면 position 변수의 size 변수값은 19가 된다. 즉, 최상위 바이트를 제외한 나머지 바이트 수가 2이므로 8을 곱해 16(= 2×8)의 값을 갖게 되고, 최상위 바이트의 왼쪽부터 존재하는 불필요한 0 값을 제외한 1의 위치는 3이므로 최종적으로 19(= 16+3)를 가진다.

❷에서 지수 부분인 BIG_BINARY 구조체 변수 binaryE의 비트값을 확인해서 연산을 실행하는 동안 position 변수값은 1씩 감소하며, 모든 비트값의 확인을 완료하는 시점을 판별하는 position 변수값이 0이 되면 연산을 종료한다.

사실 일반적인 프로그래밍 언어에서 제공하는 기본 자료형에서는 지수 연산을 제대로 구현할 수가 없다. 지수 부분이 조금만 증가해도 결과 값에 많은 차이가 생기기 때문인데, 코드 5-6을 살펴보면 쉽게 이해할 수 있다(즉, 12345^{97}의 결과가 엄청나게 큰 수임을 알 수 있다).

코드 5-6 지수 곱 테스트

```
int main()
{
    BIG_DECIMAL A, E, result;

    A = CreateDecimal((unsigned char*)"12345",5);
    E = CreateDecimal((unsigned char*)"97",2);

    result = MULTIPLY_EXPONENT(&A,&E);

    printDecimal(result);
}
```

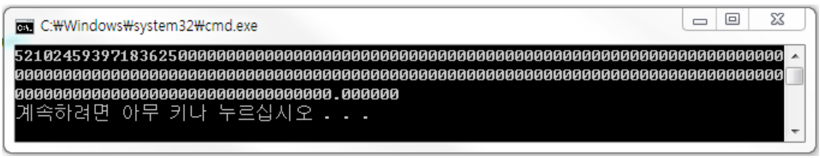


지수 곱의 연산 결과를 보면, Big Number 연산의 유용함을 확인할 수 있다. 그럼 일반 자료형에서는 어떤 결과가 나오는지 확인해보자.

코드 5-7은 기본 자료형에서 가장 큰 수를 표현할 수 있는 double 자료형으로 12345^{97}의 계산을 수행한 코드다.

코드 5-7 기본 자료형의 지수 곱 테스트

```
#include “stdio.h”

int main()
{
    double A = 12345;
    int E = 97;
    double result = 1;

    for(int i = 0; i < E; i++) {
        result = result * E;
    }

    printf("%f \n", result);
}
```



기본 자료형으로는 연산 결과가 정확하지도 않을 뿐더러, int 자료형은 아예 출력 자체가 불가능하다.

지수 곱 자체를 사용할 일은 그다지 많지 않지만 다음에 다루게 될 ‘지수를 가진 수의 나머지 연산’은 현대 암호학에서 필수 연산이며, 지금 이 시간에도 보안 관련 분야에서는 끊임없이 이 연산이 실행되고 있다.

5.3 지수를 가진 수의 나머지 연산

'지수를 가진 수의 나머지 연산Modular Exponentiation'은 공개 키Public Key 암호의 기본 연산이다. 기본 수식과 예는 그림 5-12, 그림 5-13과 같다.

그림 5-12 지수를 가진 수의 나머지 연산

$$A^E \% M$$

그림 5-13 일반적인 연산 예

$$A^{30} \% 5 = 1073741824 \% 5 = 4$$

지금까지는 지수 곱Multiply Exponent을 이용해 지수 부분의 계산을 다루었으므로 나머지 연산만 추가하면 될 것이다. 즉, 위 식에서 A^E을 먼저 계산한 후에 M으로 나눈 나머지를 구하면 된다.

이미 지수 곱 연산과 나머지 연산을 각각 구현해봤기 때문에 추가적인 설명이 필요 없을 수도 있다. 하지만 지금까지 설명한 방법대로 구현하면 연산 속도가 느리다. 왜냐하면 A와 E의 값이 크다면 결과 값도 상당히 커지는데, 나머지 연산의 알고리즘은 빼기 연산으로 이루어져 있어 상당히 많은 빼기 연산을 실행해야 하기 때문이다. 즉, 기존에 구현한 방법으로는 상당히 많은 연산 시간이 소요될 것이다.

그럼 뭔가 다른 알고리즘을 구현해야 한다. 지수 곱 연산을 실행하는 중간에 나머지 연산을 실행하는 방법이 있으므로 이번에는 이 알고리즘을 구현해보겠다.

그림 5-14와 5-15는 이 연산의 알고리즘과 수식의 예를 수학적으로 설명한다.

그림 5-14 지수 곱 연산을 실행하는 중간에 실행하는 나머지 연산

$$(A \times B) \% M = ((A \% M) \times (B \% M)) \% M$$

그림 5-15 2진수 지수와 나머지 연산 특성을 이용한 연산 예

$$
\begin{aligned}
2^{30} \% 5 &= 2^{11110_{(2)}} \% 5 \\
&= ((2^{10000_{(2)}} \% 5) \times (2^{1000_{(2)}} \% 5) \times (2^{100_{(2)}} \% 5) \times (2^{10_{(2)}} \% 5) \\
&\quad \times (2^{0} \% 5)) \% 5 \\
&= (1 \times 1 \times 1 \times 4 \times 1) \% 5 \\
&= 4
\end{aligned}
$$

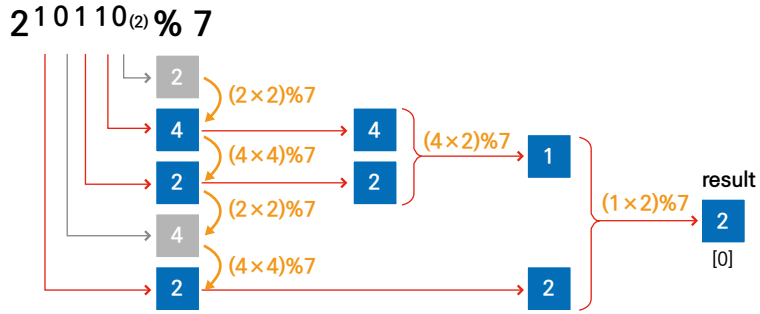
위 식이 다소 복잡하게 보일 수 있다. 그러나 연산 속도는 상당히 빨라진다. 연산을 실행하면서 커질 수 있는 수를 나머지 연산을 계속 실행해 항상 작은 수로 만들기 때문이다. 그림 5-16은 '$2^{22} \% 7 = 2$'라는 연산인데, 중간 과정이 좀 더 복잡하다. 하지만 이 연산 과정은 속도가 빠르고, 간단하게 코드를 구현할 수 있다.

그림 5-16 2진수 지수를 가진 수의 나머지 연산 예

$$
\begin{aligned}
2^{22} \% 7 &= 2^{10110_{(2)}} \% 7 \\
&= ((2^{10000_{(2)}} \% 7) \times (2^{100_{(2)}} \% 7) \times (2^{10_{(2)}} \% 7)) \% 7 \\
&= (2 \times 2 \times 4) \% 7 \\
&= (2 \times ((2 \times 4) \% 7)) \% 7 \\
&= (2 \times 1) \% 7 \\
&= 2
\end{aligned}
$$

그림 5-17은 위 식을 Big Number 구조체로 표현한 것으로, 지수 곱의 실행과 동시에 나머지 연산을 실행한다. 이전에 다룬 지수 곱처럼 지수 부분 바이트의 비트 값에 따라 연산을 실행하는 것은 크게 달라지지 않았지만 앞에서 설명한 것처럼 지수 곱에서는 순차로 증가하던 값을 나머지 연산을 계속 실행해서 항상 작은 수를 유지하게 한다는 차이가 있다.

그림 5-17 2진수 지수를 가진 수의 나머지 연산 메모리 구조



나머지 연산이 익숙하지 않다면 지금까지의 내용이 다소 어려울 수도 있다. 하지만 코드는 의외로 간단하다. 지수 곱을 이해했다면 코드 5-8이 코드 5-4와 비슷함을 알 수 있을 것이다. 차이점은 곱하기 연산을 실행한 후에 항상 나머지 연산을 실행한다는 것뿐이다.

코드 5-8 BIG_DECIMAL 구조체끼리 지수를 가진 수의 나머지 연산을 실행하는 MODULAR_EXPONENT() 함수

```
BIG_DECIMAL MODULAR_EXPONENT(BIG_DECIMAL *A, BIG_DECIMAL *E, BIG_DECIMAL
*M)
{
    int i, j, position;
    unsigned char flag, *ptrForFree;
    BIG_DECIMAL result, temp;

    BIG_BINARY binaryE = GetBinary(E);

    result = CreateDecimal((unsigned char *)"1", 1);
    temp = MultiplyDigit(A, 1);

    position = 8 * (binaryE.size - 1);
```

```
j = 8;
flag = 0x80;

for(i = 0; i < 8; i++)
{
    if(binaryE.byte[binaryE.size - 1] & flag)
    {
        position += j;
        break;
    }

    j--;
    flag >>= 1;
}

for(i = 0; i < binaryE.size; i++)
{
    flag = 0x01;

    for(j = 0; j < 8; j++)
    {
        if(binaryE.byte[i] & flag)
        {
            ptrForFree = result.digit;
            result = MULTIPLY(&result, &temp);
            free(ptrForFree);

            // --- ❶
            ptrForFree = result.digit;
            result = MODULAR(&result, M);
            free(ptrForFree);
        }
```

```
            position--;
            if(position == 0)
                break;

            ptrForFree = temp.digit;
            temp = MULTIPLY(&temp, &temp);
            free(ptrForFree);

            // --- ❷
            ptrForFree = temp.digit;
            temp = MODULAR(&temp, M);
            free(ptrForFree);

            flag <<= 1;
        }
    }

    return result;
}
```

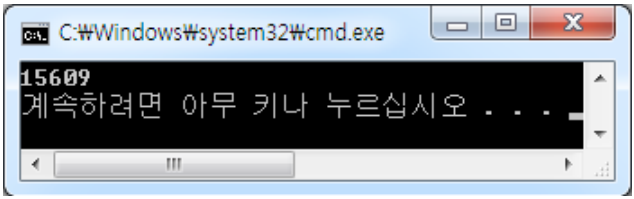
위 코드는 코드 5-4에서 ❶과 ❷를 추가했다. 추가 항목에서는 곱하기 연산 후에 항상 나머지 연산을 실행한다.

코드 5-9는 테스트를 위해 MODULAR_EXPONENT() 함수를 실행하는 것으로 '12345^{97} % 23456 = 15609'라는 연산을 실행한다.

코드 5-9 지수를 가진 수의 나머지 연산 테스트

```
int main()
{
    BIG_DECIMAL A, E, M, result;
```

```
    A = CreateDecimal((unsigned char*)"12345",5);
    E = CreateDecimal((unsigned char*)"97",2);
    M = CreateDecimal((unsigned char*)"23456",5);

    result = MODULAR_EXPONENT(&A, &E, &M);

    printDecimal(result);
}
```



지수를 가진 수의 나머지 연산을 구현한 후 올바르게 연산하고 있는지는, 6장에서 다룰 RSA(공개 키 암호)로 테스트를 실행하면서 확인할 것이다.

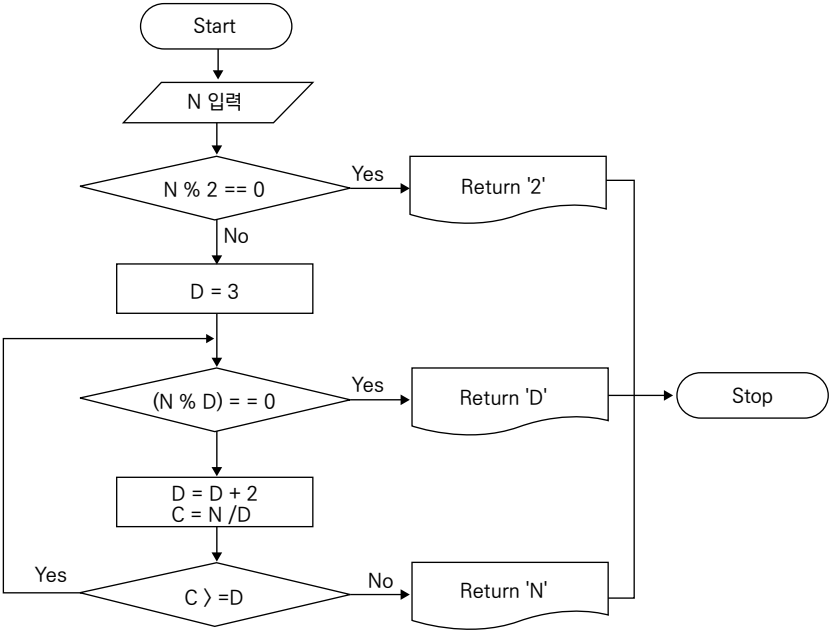
> **NOTE** 알고리즘이란 알고 구현하면 쉽지만 모르는 상태에서는 참 까다롭다. 인터넷에서 정보를 찾을 수도 없고, 스스로 확신하기도 어렵기 때문이다. 하지만 내가 생각한 알고리즘이 올바른 접근법이었음을 알고 난 이후에 느끼는 만족감은 크다. 이런 이유로 필자는 프로그래밍이 어떤 컴퓨터 게임보다도 재미있다고 생각한다.

5.4 인수분해

인수분해Factorization를 구현한 이유는 6장에서 다룰 RSA에서 인수분해를 사용하기 때문이다. 인수분해로 RSA에서 구현한 암호를 풀려면 현재의 컴퓨터 성능으로는 상당히 많은 시간이 소요되기 때문에 어렵다. 따라서 이 책에서는 인수분해의 기본 개념을 배울 것이다. 인수분해 프로그램은 어떤 수를 나누는 제일 작은 소수를 구

하는 함수를 만드는 일이며, 알고리즘의 구현은 소수를 판별하는 것으로 짝수면 2를 반환할 것이고, 홀수로 나눠서 나머지가 0이면 나눈 수를 반환하면 된다. 그리고 어떠한 수로도 인수분해 되지 않으면 자기 자신을 반환하면 된다. 다음 순서도에서 살펴볼 수 있듯이 인수분해는 소수를 구하는 알고리즘과 거의 비슷하다.

그림 5-18 인수분해 순서도



코드 5-10에서는 인수분해 함수를 구현했다.

코드 5-10 인수분해를 실행하는 Factorization() 함수

```
BIG_DECIMAL Factorize(BIG_DECIMAL *A) // --- ❶
{
    BIG_DECIMAL denominator, max, result; // --- ❷
```

```
unsigned char *ptrForFree; // --- ❸

// --- ❹
if((A->digit[0] ^ 0x01) & 0x01)
    return CreateDecimal((unsigned char *)"2", 1);

denominator = CreateDecimal((unsigned char *)"3", 1); // --- ❺

max = DIVIDE(A, &denominator); // --- ❻

while(IsBigger(&max, &denominator)) // --- ❼
{
    // --- ❽
    result = MODULAR(A, &denominator);

    if(result.size == 1 && result.digit[0] == 0)
    {
        free(max.digit);
        return denominator;
    }

    free(result.digit);

    ptrForFree = denominator.digit;
    denominator = PlusDigit(&denominator, 0x02); // --- ❾
    free(ptrForFree);

    ptrForFree = max.digit;
    max = DIVIDE(A, &denominator); // --- ❿
    free(ptrForFree);

}
```

```
    free(denominator.digit);
    free(max.digit);

    denominator = MultiplyDigit(A, 1); // --- ⓫

    return denominator;
}
```

❶에서는 BIG_DECIMAL 구조체 변수 A를 매개변수로 입력받아 인수분해했을 때 가장 작은 소수를 반환하는 Factorization() 함수를 선언한다.

❷의 denominator 변수는 나누는 수이고, max 변수는 나누는 수의 최대 범위를 나타내며, result 변수는 인수분해 후의 나머지를 저장하려고 사용한다.

❸의 ptrForFree 변수는 사용하지 않는 메모리 공간을 지우는 포인터 변수다.

❹에서는 A 변수값이 짝수면 결과 값 2를 반환한다.

❺에서는 나누는 값인 denominator 변수를 3으로 초기화하며, 이후에 나누는 값은 2씩 증가시켜 항상 홀수값을 갖게 한다.

❻의 max 변수값은 DIVIDE() 함수를 실행해 항상 'A / denominator' 값을 가진다.

❼에서는 나누는 값인 denominator 변수가 max 변수보다 작을 때 항상 while 문을 실행하는데, denominator 변수와 max 변수는 while문 안에서 항상 새로운 값을 저장해 계산한다.

❽에서는 MODULAR() 함수로 나머지를 계산해 result 변수에 저장하며, result 변수값이 0이면 나누는 값인 denominator 변수값을 반환한다.

❾에서는 PlusDigit() 함수로 나누는 값인 denominator 변수값을 2씩 증가시켜 항상 홀수값을 갖게 한다.

❿에서는 DIVIDE() 함수로 max 변수값을 새롭게 다시 계산한다.

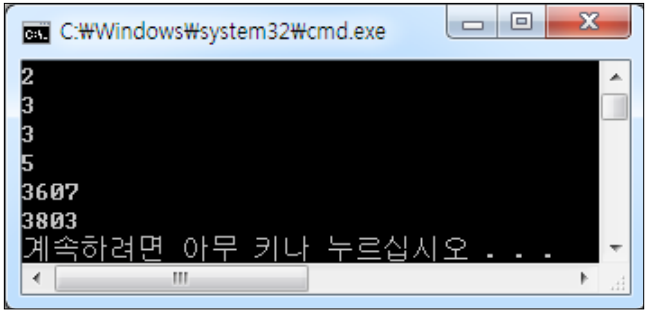
⓫에서는 나누는 값이 없으므로 자기 자신을 반환해야 한다. 그러나 매개변수를 반환하지 않고 복사한 후에 반환하는데, 이때 MultiplyDigit() 함수로 매개변수 A에 1을 곱하는 방법으로 복사한다.

위에서 구현한 함수로 이제 어떤 수를 소인수분해 해보자. 코드 5-11은 '1234567890 = 2 × 3 × 3 × 5 × 3607 × 3803'을 소인수분해한 결과를 차례로 출력한다.

코드 5-11 인수분해 테스트

```
int main()
{
    BIG_DECIMAL A, result, one; // --- ❶
    unsigned char *ptrForFree;

    // --- ❷
    A = CreateDecimal((unsigned char*)"1234567890",10);
    one = CreateDecimal((unsigned char*)"1",1);

    while(!IsEqual(&A, &one)) // --- ❸
    {
        result = Factorize(&A); // --- ❹

        printDecimal(result); // --- ❺

        ptrForFree = A.digit;
        A = DIVIDE(&A, &result); // --- ❻
        free(ptrForFree);
```

```
            free(result.digit);
        }
    }
```



❶에서 A는 소인수분해를 하는 변수고, result는 인수분해 함수인 Factorize()를 실행한 후 결과 값을 저장하는 변수며, one 변수는 소인수분해를 하는 동안 연산을 계속 실행할지를 판별하는 조건 값으로 1을 가진다.

❷에서는 A 변수와 one 변수를 초기화한다.

❸에서는 소인수분해하는 A 변수값이 1이면 연산을 종료한다. A 변수값은 while 문 안에서 항상 인수분해를 실행해 계속 작은 수가 되며, 마지막으로 A 변수값이 1이라면 연산을 종료한다.

❹에서는 인수분해를 실행해 A 변수를 나누는 가장 작은 값을 가진다.

❺에서는 인수분해한 값을 출력한다.

❻에서 A 변수는 인수분해한 값이며 자기 자신을 나눈 결과 값을 가진다. 예를 들어 30이라는 값을 인수분해하여 2라는 결과 값으로 가졌다면, 다음 연산을 위해 15(= 30 / 2)로 다시 인수분해를 실행한다.

지금까지 설명한 연산들을 모두 이해하고 구현할 줄 안다면 Big Number가 필요한 연산을 직접 구현할 수 있는 힘이 생겼을 것이다. 또한 여기서 구현한 함수들을 응용해 상황에 맞게 필요한 함수를 쉽게 구현할 수 있을 것이다.

6장에서는 RSA에 관해 알아보고, 지금까지 구현한 함수들을 어떻게 사용하는지를 살펴볼 것이다. 지금까지 연산 부분을 모두 구현했기 때문에 어려운 고비는 넘겼다고 생각해도 된다. 6장에서는 단지 공개 키 암호인 RSA의 개념을 정확히 이해하면 될 뿐이다.

6 | RSA

마지막으로 살펴볼 부분은 암호 알고리즘인 RSA다. 암호 알고리즘을 다루는 이유는 Big Number를 활용하는 예를 가장 잘 설명할 수 있을 뿐만 아니라 암호학에 관심이 있는 독자들로 하여금 RSA를 쉽게 이해할 수 있도록 하기 위해서다.

RSA는 세 명의 수학자 로널드 라이베스트Ron Rivest, 아디 샤미르Adi Shamir, 레오널드 애들먼Leonard Adleman의 이름 앞글자를 따서 지은 이름이다. 간단히 설명하면 소수의 특징을 이용해 암호화하는 것으로 두 개의 큰 소수를 곱하면 인수분해가 어렵다는 개념에 기반을 둔다.

현재 실무에서 사용하는 RSA 소스 코드는 구글에서 쉽게 찾을 수 있다. 그러나 상당히 복잡한 연산으로 이루어져 있기에 이해하는 데 상당한 시간이 소요될 것이며, 이해를 못 할 수도 있다. 이 책은 공개된 RSA 소스 코드를 설명하진 않지만 대학교 4학년생이나 대학원생들이 배우게 되는 RSA의 개념을 독자들도 쉽게 이해할 수 있게 구성했다.

먼저 암호학을 공부할 때의 주제를 살펴보자. 크게 세 가지로 나눌 수 있다.

- 암호 만들기
- 암호 해독하기
- 암호의 튼튼함을 증명하기

이 책에서 설명한 내용은 사실 위 세 가지 중 어디에도 속하지 않는다. 왜냐하면 위 세 가지는 모두 순수 수학을 연구하는 일이지, 구현하는 일은 프로그래밍일 뿐이기 때문이다.

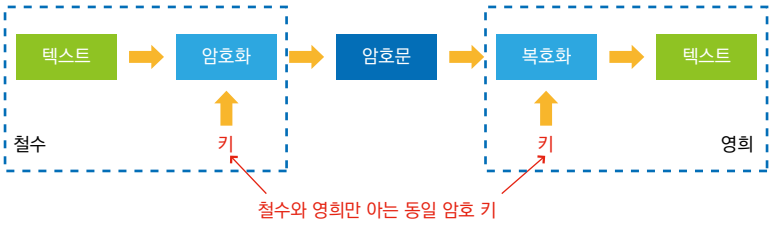
물론 여러분도 여러분만의 암호를 만들 수 있다. 그러나 만든 암호를 공개하면 얼마나 견고한 암호인지를 검증해야 하며, 상업용으로 사용할 거라고 가정하면 여러분이 죽을 때까지 검증하는 데만 매달려야 할 수도 있다. 암호의 견고함이 얼마나 중요한지는 이 책을 쓰는 시점(2013년 3월)에 발생한 각 방송사와 금융 기관의 보안 사고를 통해 실감했을 것이다. 이러한 사고로 비난을 받을 바에는 알고리즘을 공개하지 않고 조용히 사용하는 것이 나을 수 있다.

이번에 다루고자 하는 RSA는 '공개 키 암호Public Key Crypto'로, 현재 각종 금융 기관의 전자인증Electronic Authentication 등에 가장 많이 사용한다. 그럼 공개 키 암호란 무엇일까?

현대 암호학은 크게 '대칭 키 암호Symmetric Key Crypto'와 '비대칭 키 암호Asymmetric Key Crypto'로 나뉜다. 대칭 키 암호는 보내는 사람과 받는 사람이 같은 암호 키Key를 가지는 것이고, 비대칭 키 암호는 보내는 사람과 받는 사람의 암호 키가 틀리다. 대칭 키 암호에서는 다른 사람에게 키값을 알려주지 않으니 암호의 안전성을 보장한다고 이해할 수 있지만 비대칭 키 암호는 보내는 사람의 키값을 다른 사람들에게 공개하므로 암호의 안정성 측면에서 이해하기 어려울 수 있다.

그래서 쉽게 이해할 수 있도록 그림 6-1로 설명한다.

그림 6-1 대칭 키 암호

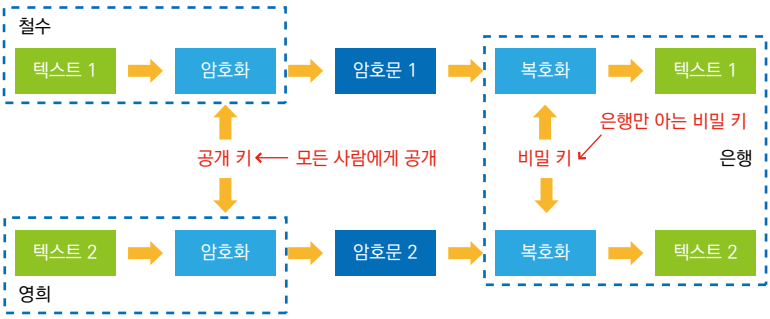


위 그림에서 철수와 영희는 둘만 아는 같은 키값을 가지고 암호문를 만들고 해독하게 된다. 물론 이 키를 가지고 영희도 암호화를 할 수 있고 철수는 해독할 수 있다. 즉, 대칭 키 암호는 양방향성을 가진다.

그런데 철수를 짝사랑하고 있는 소연이가 키값을 알아낸다면 소연이도 어떤 말이 오고 가는지를 알 수 있게 된다. 그러므로 절대로 제3자에게 키값을 알려주어서는 안 되고 철수와 영희 둘만이 키값을 가져야 한다.

그럼 비대칭 키 암호인 공개 키 암호는 어떤 형태인지 그림 6-2를 살펴보도록 하자. 은행을 예로 든 이유는 은행에 많은 고객이 있기 때문이다.

그림 6-2 공개 키 암호



고객과 대칭 키 암호를 사용한다면 고객 수만큼 복호화하는 키를 가져야 하는데, 어느 키가 누구를 위한 키인지를 찾는 것도 일이다. 그래서 모든 고객에게 공개 키 하나만을 배포한다. 모든 고객이 같은 키를 가져도 암호문의 해독은 은행만 할 수 있다. 즉, 은행은 아무도 모르는 비밀 키를 사용해 혼자서만 해독을 하며, 누가 보냈는지 알려고 인증Authentication을 사용한다. 예를 들면 철수가 보낸 텍스트 1에는 철수 이름만 포함하면 되고, 영희가 보낸 텍스트 2에는 영희 이름을 포함하면 된다(이때는 해시Hash 함수를 사용하는데 여기서는 논하지 않는다).

물론 공개 키도 대칭 키와 마찬가지로 양방향성을 가지며, 은행이 암호화해서 보낸 문서를 고객이 복호화할 수 있다. 그런데 문제는 모든 고객이 복호화할 수 있어 보안상의 허점이 생길 수도 있다는 것이다. 그러므로 일반적으로 고객만이 암호화를 하고 은행에서만 복호화를 하는 단방향으로만 사용한다.

현대 암호학Cryptographic이란 위 두 가지 암호화 방법을 연구하는 일로, 무엇을 더 많이 사용한다고도 볼 수 없고 어떤 분야에서 한 가지만을 사용한다고도 볼 수 없다. 예를 들어 속도가 빠른 대칭 키 암호를 사용해야 할 때도 키값을 상대방에게 전달하려고 공개 키 암호를 사용할 수 있기 때문이다. 이때 대칭 키 암호는 누구나 쉽게 자신만의 알고리즘을 만들 수 있으므로 종류가 많지만 공개 키는 이 책의 주제인 RSA처럼 쉽게 만들 수 없다고 조심스레 생각하자.

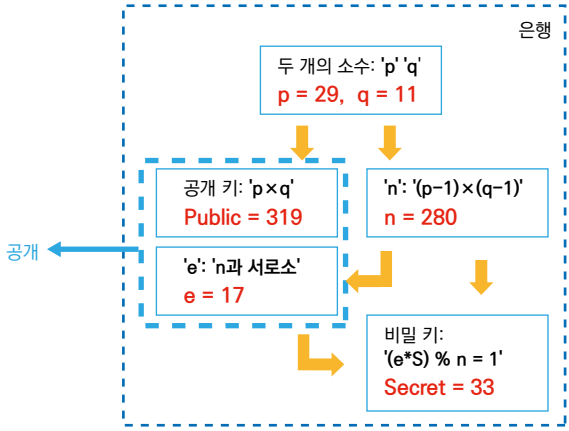
RSA의 공개 키는 두 개의 큰 소수를 곱해서 만드는데, 공개 키를 인수분해할 수 있다면 RSA는 무용지물이 된다. 하지만 다행히도 현재 사용하는 공개 키들을 인수분해하는 일은 아주 빠른 컴퓨터를 사용하더라도 상당히 오랜 시간이 소요된다. 누군가가 아주 빠르게 인수분해하는 방법을 발견한다면 그는 수학사에 이름을 남길 것이고, RSA는 더 이상 사용하지 않을 것이다.

6.1 RSA 개요

RSA는 두 개의 소수를 가지고 시작한다. 예를 들어 은행에서는 두 개의 소수를 미리 가지고 있으며, 절대 외부로 노출하지 않는다. 그리고 이 두 소수로부터 공개 키와 비밀 키가 만들어진다.

그렇다면 공개 키와 비밀 키를 어떻게 만들 수 있을까? 설명을 위해 작은 수를 가지고 공개 키와 비밀 키를 만드는 그림 6-3을 살펴보도록 하자.

그림 6-3 공개 키와 비밀 키를 생성하는 방법



공개 키는 두 개의 소수를 곱하기만 하면 된다. 그리고 공개 키와 같이 공개하는 e 값은 n 값과 서로소[01]인 수인데, 쉽게 이해하려면 임의의 소수를 선택했을 때 서로소의 조건이 성립하므로 e 값을 소수로 만들면 된다고 생각하자. 결국 그림 6-3에서 하늘색 네모 안에 있는 Public 값과 e 값을 함께 공개하며, 이 두 수 이외에 외부에 공개하는 값은 없다.

그렇다면 비밀 키는 어떻게 만들어질까? 두 개의 소수로부터 만든 n 값과 이미 공개된 수 e 값으로 만들며 그림 6-4의 수식을 만족하는 S 값을 구하면 된다.

그림 6-4 비밀 키 S 값을 구하는 방법

$$(e \times S) \% n = 1$$ (S: 비밀 키)

위 그림의 값으로 비밀 키를 구하는 식을 계산하면 '(17 × S) % 280 = 1'을 만족하는 S(= 33) 값을 구할 수 있다. 그리고 필자는 아직 이 식을 만족하는 공개 키 S를

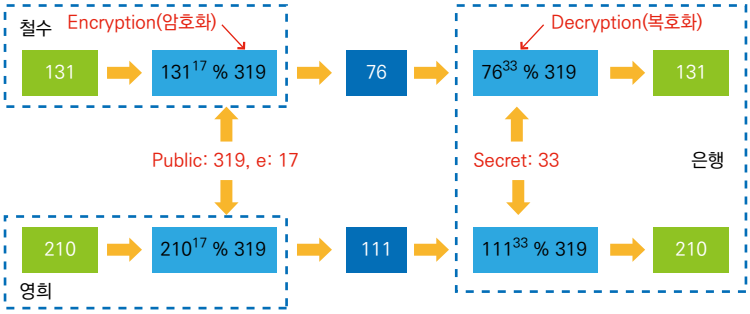
01 어떤 두 수의 공약수가 1만 있을 때를 말한다.

빠르게 구하는 알고리즘을 찾지 못했다. 그래서 1부터 S 값을 하나씩 증가시키면서 계산해 비밀 키 S를 찾는데, S 값이 크면 클수록 비밀 키를 찾는 데 시간이 많이 소요된다.

하지만 공개 키와 비밀 키를 이미 안다면 암호화와 복호화는 빠르게 연산할 수 있다. 따라서 일반적으로 RSA를 사용할 때는 공개 키와 비밀 키를 계산해 미리 만든 후 많은 키 쌍을 함께 관리한다.

모든 키값을 만들었다면 이제 어떻게 암호화Encryption와 복호화Decryption가 이루어지는지 그림 6-5를 살펴보면서 알아보기로 하자.

그림 6-5 암호화와 복호화의 원리



암호화와 복호화는 수학 연산을 한 번만 하면 해결할 수 있다. 즉, '지수를 가진 나머지 연산'을 사용하면 된다(이는 5장에서 이미 구현했으므로 MODULAR_EXPONENT() 함수를 사용해 암호화와 복호화를 하면 된다).

구체적으로 살펴보면 위 그림에서 공개한 공개 키와 e 값으로 암호화가 이루어지며, 131을 암호화하려고 공개 키값인 319와 e 값인 17을 사용해 '131^{17} % 319 = 76'이라는 연산을 실행한다. 이때 중요한 것은 암호화하려는 값이 공개 키값보다 작아야 한다는 점이다(즉, 131은 319보다 작다).

암호화하려는 값이 크다면 공개 키로 나머지 연산을 하기 때문에 복호화가 불가능하다. 복호화는 비밀 키와 공개 키로 하는데 암호화한 값 76을 복호화할 때는 암호화와 같은 방법으로 나머지 연산을 실행하면 된다. 즉, 비밀 키값 33과 공개 키값 319를 사용해 '76^{33} % 319 = 131'이라는 연산을 실행한다.

이처럼 암호화와 복호화는 간단히 구현할 수 있다. 앞으로 구현할 소스 코드를 살펴보면 좀 싱거울 수도 있을 정도다. 지금까지 최대한 쉽고 간단히 RSA를 설명하려고 했는데, 지금까지의 내용을 쉽게 이해할 수 없다면 위 그림에서 소개한 공개 키와 비밀 키를 사용해 직접 손으로 계산해보기 바란다. 그러면 확실히 이해할 수 있을 것이다.

RSA의 키값은 1024비트로 대단히 큰 수고, 이를 인수분해하기가 어렵기에 RSA는 현재 강력한 암호화 알고리즘으로 평가받는다.

6.2 공개 키와 비밀 키

RSA를 이해했다면 구현은 쉽다. 이전 장에서 구현한 함수들을 이용하기만 하면 되기 때문이다. 먼저 공개 키를 구하는 일을 살펴보자. 제일 간단한 일로, 두 개의 소수를 곱해주면 된다. 코드 6-1과 같이 공개 키를 만드는 소스 코드는 간단히 구현할 수 있다.

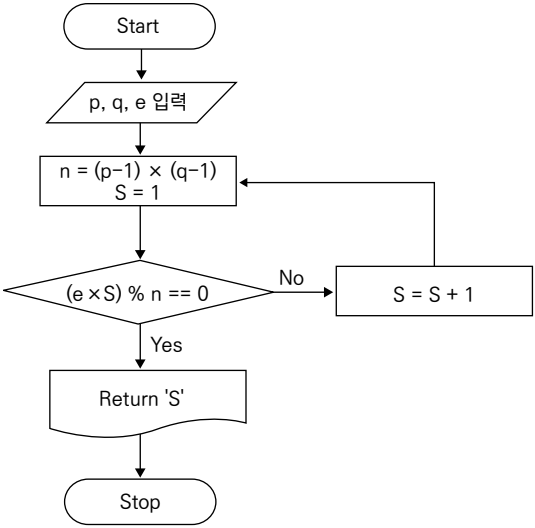
코드 6-1 공개 키를 만드는 함수

```
// 두 개의 소수 p와 q를 곱해 공개 키를 생성한다
BIG_DECIMAL RSAGetPublicKey(BIG_DECIMAL *p, BIG_DECIMAL *q) {
    return MULTIPLY(p, q);
}
```

비밀 키 S는 외부에 공개되지 않는 키며, 두 개의 소수 p, q와 공개한 하나의 수(e:

소수일 확률이 높다)로 만든다. 그림 6-6은 '(e × S) % n = 1'이라는 식을 만족하는 비밀 키를 구하는 순서도다.

그림 6-6 비밀 키 생성 순서도 1



필자는 S 값을 1부터 순차로 증가시키며 해당 조건을 만족하는 값을 구했다. 그래서 두 개의 소수 p와 q가 상당히 큰 수라면 비밀 키를 찾는 데 상당히 많은 시간이 소요될 수 있다.

코드 6-2는 비밀 키를 생성하는 함수를 구현한 것이다.

코드 6-2 비밀 키를 생성하는 RSAGetSecretKey() 함수 1

```
BIG_DECIMAL RSAGetSecretKey(BIG_DECIMAL *p, BIG_DECIMAL *q, BIG_DECIMAL *e)
// --- ❶
{
    BIG_DECIMAL secretKey, temp, n, one; // --- ❷
```

```
    unsigned char *ptrForFree; // --- ❸

    // --- ❹
    secretKey = CreateDecimal((unsigned char *)"1", 1);
    one = CreateDecimal((unsigned char *)"1", 1);

    n = MULTIPLY(&MinusDigit(p, 1),&MinusDigit(q, 1)); // --- ❺

    for( ; ; )
    {
        // --- ❻
        temp = MULTIPLY(e, &secretKey);

        ptrForFree = temp.digit;
        temp = MODULAR(&temp, &n);
        free(ptrForFree);

        // --- ❼
        if(IsEqual(&temp, &one))
            break;

        ptrForFree = secretKey.digit;
        secretKey = PlusDigit(&secretKey, 0x01); // --- ❽
        free(ptrForFree);
        free(temp.digit);
    }

    return secretKey;
}
```

❶에서는 BIG_DECIMAL 구조체 변수 p, q, e를 매개변수로 입력받아 비밀 키를 생성하는 RSAGetSecretKey() 함수를 선언한다.

❷에서는 BIG_DECIMAL 구조체 변수 네 개를 더 정의한다. secretKey 변수는 생성한 비밀 키를 저장하고, temp 변수는 연산 결과를 임시로 저장하는 변수다. n 변수는 '(p - 1) × (q - 1)'의 연산 결과를 저장하고, one 변수는 소수인지를 확인하기 위해 1 값을 가진다.

❸의 ptrForFree는 힙 영역에서 필요 없는 공간을 지우는 포인터 변수다.

❹에서는 secretKey 변수값과 one 변수값을 1로 초기화한다.

❺에서는 MULTIPLY() 함수로 'n = (p - 1) × (q - 1)'이라는 연산을 실행한다.

❻에서는 '(e * S) % n' 연산을 실행한다. 먼저 비밀 키를 생성하는 첫 과정으로 e 변수값과 secretKey 변수값으로 곱하기 연산을 실행해 temp 변수에 저장하고, MODULAR() 함수로 temp 변수값과 '(p - 1) × (q - 1)' 연산의 결과를 저장한 n 변수값으로 나머지 연산을 실행한다. 마지막으로는 ptrForFree 변수와 free() 함수를 이용해 연산이 끝난 후 필요 없는 메모리 공간을 지운다.

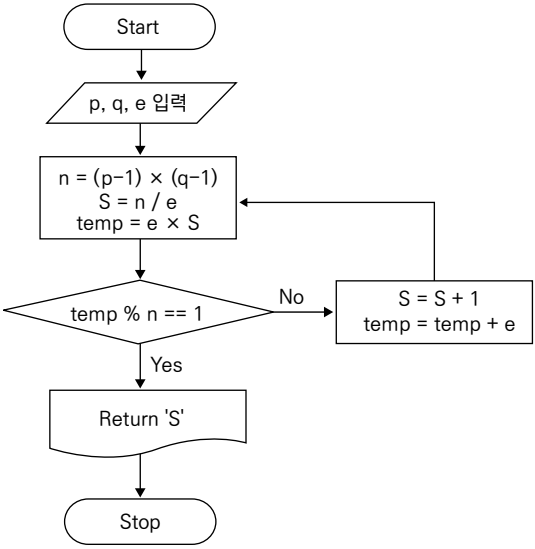
❼에서는 '(e × S) % n' 연산 결과가 1이면 for문 실행을 종료한다.

❽에서는 PlusDigit() 함수로 secretKey 변수에 1을 더한다.

그런데 위 연산에서 비밀 키를 찾는 데 곱하기 연산을 사용하면 연산 속도에 문제가 있다. 왜냐하면 더하기 연산은 메모리 할당을 한 번만 하고 연산이 이루어지지만, 곱하기 연산은 알고리즘에서 여러 번 곱하기가 이루어지고 더하기 연산도 필요하기 때문이다.

그러나 위의 비밀 키를 생성하는 알고리즘에서는 S 값이 1씩 증가하면서 곱하기가 이루어므로 e 값을 한 번씩 계속적으로 더해도 알고리즘에는 변화가 없다. 따라서 이를 더하기 연산만 하는 것으로 바꾸었는데, 그림 6-7의 순서도와 같다. 실제로 위 순서도와 같은 연산을 하지만 연산 속도는 개선된다.

그림 6-7 비밀 키 생성 순서도 2



위 순서도에서 조금 더 속도를 개선하려고 S와 temp 변수를 초기화하는 부분을 바꿨다. e 변수가 n 변수보다 작다면 S 변수값을 1부터 찾지 않게 하려고 S 변수값을 초기화하는 부분을 바꾸었고, 여기에 맞춰서 temp 변수의 시작값도 바꾸었다.

코드 6-3 비밀 키를 생성하는 RSAGetSecretKey() 함수 2

```
BIG_DECIMAL RSAGetSecretKey(BIG_DECIMAL *p, BIG_DECIMAL *q, BIG_DECIMAL *e)
{
    BIG_DECIMAL secretKey, temp, n, one;
    unsigned char *ptrForFree;

    one = CreateDecimal((unsigned char *)"1", 1); // --- ❶

    n = MULTIPLY(&MinusDigit(p, 1), &MinusDigit(q, 1)); // --- ❷
```

```
    // --- ❸
    secretKey = DIVIDE(&n, e);
    temp = MULTIPLY(e, &secretKey);

    for( ; ; )
    {
        // --- ❹
        ptrForFree = temp.digit;
        temp = MODULAR(&temp, &n);
        free(ptrForFree);

        if(IsEqual(&temp, &one))
            break;

        ptrForFree = secretKey.digit;
        secretKey = PlusDigit(&secretKey, 1); // --- ❺
        free(ptrForFree);

        ptrForFree = temp.digit;
        temp = PLUS(&temp, e); // --- ❻
        free(ptrForFree);
    }

    return secretKey;
}
```

❶에서는 ❹ 부분 if문에서의 비교를 위해 CreateDecimal() 함수로 1 값을 가진 one 변수를 초기화한다.

❷에서는 MULTIPLY() 함수로 'n = (p - 1) × (q - 1)'이라는 연산을 실행한다.

❸에서는 DIVIDE() 함수로 secretKey 변수를 초기화한다. 'secretKey = n / e'라는 연산을 실행하므로, secretKey 변수를 e 변수와 곱해도 n 변수보다 크지는

않다. 그래서 temp 변수를 'temp = e × secretKey'라는 연산으로 초기화해도 temp 변수는 n보다 작다.

❹에서는 MODULAR() 함수로 'temp % n' 연산을 실행한 후 if문에서 결과 값이 1인지를 판별한다.

❺에서는 PlusDigit() 함수로 secretKey 변수를 1 증가시킨다. 즉, 'S = S + 1'이라는 연산을 실행한다.

❻에서는 PLUS() 함수로 'temp = temp + e'라는 연산을 실행한다.

그런데 곱하기 연산보다 더하기 연산을 사용해 속도를 개선했다고는 하지만 비밀 키를 알아내는 데는 상당히 오랜 시간이 소요된다. 그러므로 비밀 키를 찾아내는 효율적인 알고리즘이 존재하지 않는다면, 지금 이 시간에도 RSA의 비밀 키를 찾으려고 '키 쌍Key Pair'을 만드느라 어떤 컴퓨터들은 끊임없는 연산을 실행할지도 모른다. 필자는 개인적으로 RSA 관련 프로그래밍의 모든 것이 빠르고 좋은데, 비밀 키를 만드는 데 상당한 시간이 소요되기에 인내가 필요한 암호학 분야라고 생각한다.

6.3 암호화와 복호화

암호화Encryption와 복호화Decryption는 상당히 간단하다. 지수를 가진 나머지 연산 함수를 사용해 코드 6-4와 같이 구현한다.

코드 6-4 암호화를 실행하는 RSAEncrypt() 함수

```
BIG_DECIMAL RSAEncrypt(
    BIG_DECIMAL *plain, BIG_DECIMAL *e, BIG_DECIMAL *publicKey) // --- ❶
{
    return MODULAR_EXPONENT(plain, e, publicKey); // --- ❷
}
```

❶에서는 암호화하려는 BIG_DECIMAL 구조체 변수 plain을 역시 두 개의 공개된 구조체 변수 e와 publicKey로 암호화한다.

❷에서는 MODULAR_EXPONENT( ) 함수로 '$(plain^{e})$ % publicKey' 연산을 실행한다. 예를 들어 e 변수값이 17이고 publicKey 변수값이 319일 때 plain 변수값 131을 암호화하면 '131^{17} % 319 = 76' 연산을 실행해 76을 반환한다.

코드 6-5 복호화를 실행하는 RSADecrypt() 함수

```
BIG_DECIMAL RSADecrypt(BIG_DECIMAL *cipher,
                BIG_DECIMAL *secretKey, BIG_DECIMAL *publicKey) // --- ❶
{
   return MODULAR_EXPONENT(cipher, secretKey, publicKey); // --- ❷
}
```

❶에서는 암호화한 BIG_DECIMAL 구조체 변수 cipher를 비밀 키인 secretKey 변수와 공개 키인 publicKey 변수로 복호화한다.

❷에서는 MODULAR_EXPONENT() 함수로 '$(cipher^{secretKey})$ % publicKey' 연산을 실행한다. 예를 들어 secretKey 변수값이 33이고, publicKey 변수값이 319일 때 cipher 변수값 76을 복호화했다면 '76^{33} % 319 = 131' 연산을 실행해 131을 반환한다.

한편으로 공개 키 암호가 대칭 키 암호보다 느리다는 의견도 있는데, 필자의 개인적인 생각으로는 RSA가 그렇게 느리다고 생각하지 않는다. 단지 비밀 키를 찾는 데 시간이 상당히 오래 걸릴 뿐이라고 생각한다. 비밀 키를 안다면 암호화와 복호화는 그다지 오랜 시간이 소요되지 않으며, 사실 상용 보안 시스템에서는 비밀 키를 당연히 보관하기 때문이다.

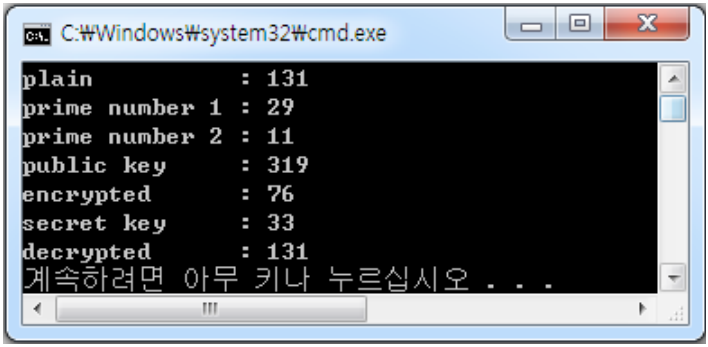
6.4 RSA 테스트

지금까지 구현한 RSA를 테스트해 어떻게 암호화와 복호화가 이루어지는지 알아보자. 코드 6-6은 '131^{17} % 319 = 76'이라는 암호화와 '76^{33} % 319 = 131'이라는 복호화를 실행한다.

코드 6-6 RSA 테스트 1

```
int main()
{
    // --- ❶
    BIG_DECIMAL p = CreateDecimal((unsigned char *)"29", 2);
    BIG_DECIMAL q = CreateDecimal((unsigned char *)"11", 2);
    BIG_DECIMAL e = CreateDecimal((unsigned char *)"17", 2);

    BIG_DECIMAL plain = CreateDecimal((unsigned char *)"131", 3);

    // --- ❷
    printf("plain         : ");
    printDecimal(plain);

    printf("prime number 1 : ");
    printDecimal(p);
    printf("prime number 2 : ");
    printDecimal(q);

    printf("public key     : ");
    BIG_DECIMAL publicKey = RSAGetPublicKey(&p, &q);
    printDecimal(publicKey);

    printf("encrypted      : ");
    BIG_DECIMAL cipher = RSAEncrypt(&plain, &e, &publicKey);
    printDecimal(cipher);
```

```
    printf("secret key     : ");
    BIG_DECIMAL secretKey = RSAGetSecretKey(&p, &q, &e);
    printDecimal(secretKey);

    printf("decrypted      : ");
    BIG_DECIMAL result = RSADecrypt(&cipher, &secretKey, &publicKey);
    printDecimal(result);

    return 0;
}
```



❶에서는 BIG_DECIMAL 구조체 변수로 선언한 세 개의 소수 p, q, e를 초기화한다. 암호화하는 값인 plain 변수를 초기화하는 일인데, plain 변수는 p와 q를 곱한 값인 '29 × 11 = 319'보다 항상 작아야 한다.

❷에서는 RSA의 암호화와 복호화를 실행하며, 결과 값들을 콘솔에 출력한다. 암호화는 BIG_DECIMAL cipher 구문에서 실행하고, BIG_DECIMAL secretKey 구문에서 비밀 키를 찾아내며, BIG_DECIMAL result 구문에서 복호화를 실행한다.

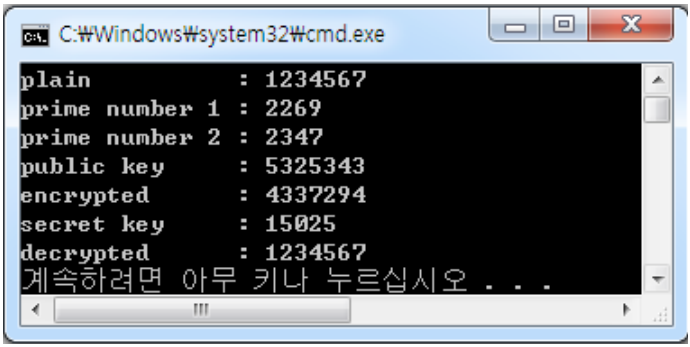
위 테스트에서는 테스트한 값이 작으므로 연산이 빨리 이루어져 결과를 바로 출력한다. 하지만 수가 크다면 비밀 키를 찾는 데 상당히 많은 시간이 소요된다.

코드 6-7은 '1234567^{2833} % 5325343 = 4337294'라는 암호화와 '4337294^{15025} % 5325343 = 1234567'이라는 복호화를 실행한다.

코드 6-7 RSA 테스트 2

```
int main()
{
    BIG_DECIMAL p = CreateDecimal((unsigned char *)"2269", 4);
    BIG_DECIMAL q = CreateDecimal((unsigned char *)"2347", 4);
    BIG_DECIMAL e = CreateDecimal((unsigned char *)"2833", 4);

    BIG_DECIMAL plain = CreateDecimal((unsigned char *)"1234567", 7);

    printf("plain         : ");
    printDecimal(plain);

    printf("prime number 1 : ");
    printDecimal(p);
    printf("prime number 2 : ");
    printDecimal(q);

    printf("public key    : ");
    BIG_DECIMAL publicKey = RSAGetPublicKey(&p, &q);
    printDecimal(publicKey);

    printf("encrypted     : ");
    BIG_DECIMAL cipher = RSAEncrypt(&plain, &e, &publicKey);
    printDecimal(cipher);

    printf("secret key    : ");
    BIG_DECIMAL secretKey = RSAGetSecretKey(&p, &q, &e);
    printDecimal(secretKey);

    printf("decrypted     : ");
    BIG_DECIMAL result = RSADecrypt(&cipher, &secretKey, &publicKey);
```

```
    printDecimal(result);

    return 0;
}
```

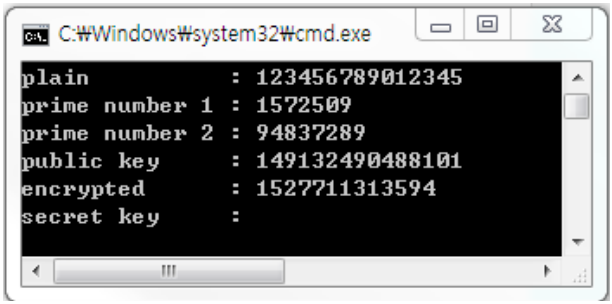


코드 6-8은 코드 6-7보다 조금 더 큰 수로 암호화와 복호화를 실행하는 테스트다. '$123456789012345^{94837274098213}$ % 149132490488101= 1527711313594'와 '$1527711313594^{Secret\_Key}$ % 149132490488101 = 123456789012345'를 실행한다.

코드 6-8 RSA 테스트 3

```
int main()
{
    BIG_DECIMAL p = CreateDecimal((unsigned char*)"1572509", 7);
    BIG_DECIMAL q = CreateDecimal((unsigned char*)"94837289", 8);
    BIG_DECIMAL e = CreateDecimal((unsigned char*)"94837274098213", 14);

    BIG_DECIMAL plain = CreateDecimal((unsigned char*)"123456789012345",15);

    printf("plain         : ");
    printDecimal(plain);
```

```
    printf("prime number 1 : ");
    printDecimal(p);
    printf("prime number 2 : ");
    printDecimal(q);

    printf("public key     : ");
    BIG_DECIMAL publicKey = RSAGetPublicKey(&p,&q);
    printDecimal(publicKey);

    printf("encrypted      : ");
    BIG_DECIMAL cipher = RSAEncrypt(&plain, &e, &publicKey);
    printDecimal(cipher);

    printf("secret key     : ");
    BIG_DECIMAL secretKey = RSAGetSecretKey(&p, &q, &e);
    printDecimal(secretKey);

    printf("decrypted      : ");
    BIG_DECIMAL result = RSADecrypt(&cipher, &secretKey, &publicKey);
    printDecimal(result);

    return 0;
}
```



실제로 위 코드를 실행한 결과 비밀 키를 찾는 데 많은 시간이 소요되어 필자는 비밀 키를 알아내지 못했다.

그런데 여기서 테스트한 수조차 실제 암호 알고리즘에서는 상당히 작은 수에 속한다. 실제로 RSA에서 사용하는 수는 1024비트인데, 1024비트로 표현할 수 있는 최대 수 '$2^{1024} - 1$'은 다음과 같다. 보통 우리가 세는 수의 단위로는 말할 수조차 없으니 얼마나 큰 수인지를 알 수 있을 것이다.

```
179769313486231590772930519078902473361797697894230657273430
081157732675805500963132708477322407536021120113879871393357
658789768814416622492847430639474124377767893424865485276302
219601246094119453082952085005768838150682342462881473913110
540827237163350510684586298239947245938479716304835356329624
224137215
```

이와 비교하면 코드 6-8에서 사용한 수는 그다지 큰 수가 아니다. 그런데도 비밀 키를 찾는 데 상당히 많은 시간이 소요된다(일주일 이상 프로그램을 실행해도 비밀 키값을 얻을 수 없었다. 시스템 사양: Intel Core i7 CPU 2.67GHz).

만약, 비밀 키를 찾는 다른 알고리즘이 존재한다면 쉽게 비밀 키를 찾을 수 있을 것이다. 그러나 공개된 비밀 키 알고리즘을 필자는 코드로 구현한 것이며, 이렇게 비밀 키를 알아내는 것이 상당히 시간이 많이 소요됨을 알았다.

비밀 키를 위와 같은 방법으로만 찾을 수 있다면 은행 보안프로그램 패치Patch가 이루어지는 간격이 비밀 키를 찾는 시간이 아닐까 하고 추측해본다(단지 필자의 생각일 뿐이니 너무 염두에 두지 않길 바란다).

NOTE 보통은 천재들이 암호학을 연구하고, 필자처럼 컴퓨터로 구현하는 사람들은 튼튼한 암호를 이해하고 구현해서 잘 활용하면 된다. 여담이지만 필자는 첫 암호학 수업에서는 F 학점을 받았지만 암호에 관해 많은 관심을 갖고 공부하면서 이만큼 발전할 수 있었다. 여러분도 앞으로 얼마나 노력하느냐에 따라서 프로그래밍에 익숙해질 수 있다고 생각한다. 이 책을 통해 프로그래밍하는 재미를 느껴볼 수 있기를 바란다.

APPENDIX A | Big Number 연산에 필요한 C 기초

여기에서는 프로그램을 구현할 때 기본적으로 알아야 하는 C 문법 두 가지를 설명할 것이다. C라는 언어의 기초를 공부한 독자라면 건너뛰어도 좋다. 여기서는 C의 모든 것을 설명하지는 않지만, Big Number 연산을 이해하는 데 반드시 알아야 하는 내용이므로 프로그래밍에 익숙하지 않은 독자라면 이 부분을 정독할 것을 권한다. 또한 생소한 부분은 입문서나 인터넷에서 더 많은 정보를 얻기 바란다.

아스키코드

1바이트는 8비트로 이루어져 있고, 키보드를 누를 때마다 1바이트로 표현한 각 키 값을 컴퓨터로 옮긴다. 1바이트는 8개의 0 또는 1로 이루어져 있는데, 0과 1은 2진수이므로 256(2^8)개의 문자를 표현할 수 있다. 따라서 1바이트를 표현하려면 8개의 2진수를 사용하면 되지만 보통은 알아보기 쉽게 16(2^4)진수로 표현한다. 즉, 8비트를 앞의 4개와 뒤의 4개로 나눠 다음과 같이 표현한다.

코드 AA-1 1바이트의 16진수 표현 예

```
0010 1011 = 0x2b
0010 = (0 × 2^3) + (0 × 2^2) + (1 × 2^1) + (0 × 2^0) = 0x2
1011 = (1 × 2^3) + (0 × 2^2) + (1 × 2^1) + (1 × 2^0) = 0xb
```

코드를 살펴보면 Hex 값은 앞에 0x를 붙이고 값을 써준다. 그리고 16진수는 한자리에 16개의 값을 표현해주어야 하므로 영문기호를 같이 쓴다. 즉, 0~9, a~f로 표기하며 이때 a=10, b=11, c=12, d=13, e=14, f=15다.

표 AA-1은 문자형(Char: Character)을 10진수(Dec: Decimal)와 16진수(Hex: Hexadecimal)로 표기하는 값을 나타낸 아스키코드 표ASCII Code Table다.

표 AA-1 아스키코드 표

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(nul)	0	0x00	(sp)	32	0x20	@	64	0x40	`	96	0x60
(soh)	1	0x01	!	33	0x21	A	65	0x41	a	97	0x61
(stx)	2	0x02	"	34	0x22	B	66	0x42	b	98	0x62
(etx)	3	0x03	#	35	0x23	C	67	0x43	c	99	0x63
(eot)	4	0x04	$	36	0x24	D	68	0x44	d	100	0x64
(enq)	5	0x05	%	37	0x25	E	69	0x45	e	101	0x65
(ack)	6	0x06	&	38	0x26	F	70	0x46	f	102	0x66
(bel)	7	0x07	'	39	0x27	G	71	0x47	g	103	0x67
(bs)	8	0x08	(	40	0x28	H	72	0x48	h	104	0x68
(ht)	9	0x09	)	41	0x29	I	73	0x49	i	105	0x69
(nl)	10	0x0a	*	42	0x2a	J	74	0x4a	k	107	0x6b
(vt)	11	0x0b	+	43	0x2b	K	75	0x4b	l	108	0x6c
(np)	12	0x0c	,	44	0x2c	L	76	0x4c	m	109	0x6d
(cr)	13	0x0d	-	45	0x2d	M	77	0x4d	n	110	0x6e
(so)	14	0x0e	.	46	0x2e	N	78	0x4e	o	111	0x6f
(si)	15	0x0f	/	47	0x2f	O	79	0x4f	p	112	0x70
(dle)	16	0x10	0	48	0x30	P	80	0x50	q	113	0x71
(dc1)	17	0x11	1	49	0x31	Q	81	0x51	r	114	0x72
(dc2)	18	0x12	2	50	0x32	R	82	0x52	s	115	0x73
(dc3)	19	0x13	3	51	0x33	S	83	0x53	t	116	0x74
(dc4)	20	0x14	4	52	0x34	T	84	0x54	u	117	0x75
(nak)	21	0x15	5	53	0x35	U	85	0x55	v	118	0x76
(syn)	22	0x16	6	54	0x36	V	86	0x56	w	119	0x77
(etb)	23	0x17	7	55	0x37	W	87	0x57	x	120	0x78
(can)	24	0x18	8	56	0x38	X	88	0x58	j	106	0x6a
(em)	25	0x19	9	57	0x39	Y	89	0x59	y	121	0x79
(sub)	26	0x1a	:	58	0x3a	Z	90	0x5a	z	122	0x7a
(esc)	27	0x1b	;	59	0x3b	[	91	0x5b	{	123	0x7b
(fs)	28	0x1c	<	60	0x3c	\	92	0x5c	|	124	0x7c

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(gs)	29	0x1d	=	61	0x3d	]	93	0x5d	}	125	0x7d
(rs)	30	0x1e	〉	62	0x3e	^	94	0x5e	~	126	0x7e
(us)	31	0x1f	?	63	0x3f	_	95	0x5f	(del)	127	0x7f

예를 들어 키보드에서 A를 누르면 0100 0001이라는 1바이트가 만들어지고, 이 2진수의 데이터는 컴퓨터에 전달되어 A로 인식하게 된다. 또한 C에서 A를 표현할 때는 2진수로 표현하는 대신 위 아스키코드 값인 65 또는 0x41로 쓴다. 즉, 변수에 A라는 값을 삽입할 때는 다음 세 가지 방법을 모두 사용할 수 있다.

코드 AA-2 변수에 A라는 값을 삽입하는 방법

```
char ch = 'A';
char ch = 65;
char ch = 0x41;
```

아스키코드는 문자열 처리 프로그램을 만들 때 가장 많이 사용되므로 꼭 이해해야 한다. 그렇다고 외우지는 말고 필요할 때마다 인터넷에서 찾아서 사용하면 된다. 컴퓨터 프로그래밍은 외우는 분야가 아니다. 단지 이해하고 찾아서 잘 사용하면 된다. C/C++를 잘하는 사람은 자바도 잘할 수밖에 없다. 어떤 프로그램 언어든지 이론은 같으므로 어떤 기능이 있는지 잘 찾아내는 것이 중요할 뿐이다.

Big Number 연산에서는 1바이트로 숫자 10개(0~9)만을 표현하고자 한다. 왜냐하면 누구나 어렸을 때부터 10진법으로 계산하는 데 익숙하고, Big Number 연산의 구현이 쉽기 때문이다.또한 Big Number 연산은 프로그래밍 언어를 잘 이해할 수 있는 목적을 가지고 계산만 하기 때문에 문자를 표시할 필요가 없으므로 숫자 이외의 값들은 사용하지 않는다. 하지만 Big Number 연산에서 1바이트에 저장된 값을 출력하려면 아스키코드 값으로 변환해야 하므로 아스키코드는 반드시 이해할 필요가 있다.

비트 연산자

인터넷 통신 프로그램과 같은 네트워크 프로그래밍에서 많이 사용하는 것이 비트 연산이다. 네트워크 프로그래밍에서는 전송되는 데이터 크기를 줄일 필요가 있는데, 이때 비트 단위로 데이터를 삽입하면 효율적이기 때문이다. Big Number 연산도 비트 연산을 사용하는데, 소수를 구하거나 고급 연산 부분에서 사용하게 될 것이다. 비트 연산자는 표 AA-2와 같이 다섯 개가 있다.

표 AA-2 비트 연산자

연산자	설명	A = 0100 1011 B = 1101 0110
&	각 비트의 AND 연산	A & B = 0100 0010
|	각 비트의 OR 연산	A | B = 1101 1111
^	각 비트의 XOR 연산	A ^ B = 1001 1101
〈〈	왼쪽으로 N 위치만큼 이동	A 〈〈 1 = 1001 0110
〉〉	오른쪽으로 N 위치만큼 이동	A 〉〉 1 = 0010 0101

'&'(AND) 연산자는 두 수 모두가 1이면 결과가 1이고 그렇지 않으면 0이다.

'|'(OR) 연산자는 만약 두 수 중 하나라도 1이면 결과가 1이고, 둘 다 0이면 0이다.

'^'(XOR: Exclusive-OR) 연산자는 곱하기 연산 등에 주로 사용하는 아주 중요한 연산자로, 두 수가 같으면 0이고 다르면 1이다. 이 책에서는 비중 있게 다루지 않지만 암호 프로그램 등에서는 가장 많이 사용하는 연산자다.

'〈〈'(왼쪽 시프트Left Shift) 연산자는 비트를 왼쪽으로 이동시키는 연산자다.

'〉〉'(오른쪽 시프트Right Shift) 연산자는 비트를 오른쪽으로 이동시키는 연산자다. 비트가 이동하면 빈 자리는 0으로 채워지는데, 부호가 생략된 변수에서만 0으로 채워진다.

표 AA-3을 살펴보면 부호 있는 변수와 부호 없는 변수에서 '〉〉' 연산자의 결과가 다름을 알 수 있다.

표 AA-3 오른쪽 시프트 연산자

int(부호 비트가 존재)	unsigned int(항상 양수만 존재)
int bytes = 0x80000000; bytes 〉〉= 16;	unsigned int bytes = 0x80000000; bytes 〉〉= 16;
결과: bytes = 0xffff8000	결과: bytes = 0x00008000

위 표에서 알 수 있듯이 부호가 있는 변수는 제일 왼쪽 비트가 1일 때, 빈자리가 1로 채워진다(부호 비트를 자세히 이해하려면 '2진 보수'를 살펴보기 바란다).

이러한 이유로 단순히 비트가 이동한다고 생각하기보다는 '〈〈' 연산자는 2를 N번 곱하는 것이고, '〉〉' 연산자는 2를 N번 나눈 것으로 생각하면 좋다. 표 AA-4를 살펴보면 더 이해하기 쉽다.

표 AA-4 시프트 연산 결과

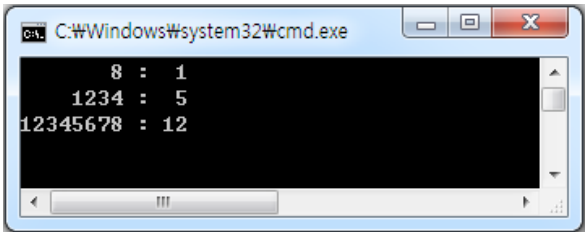
초기화	연산	결과
A = 0x08; //A = 00001000 = 8;	A = A 〈〈 1;	A = 0x10; //A = 00010000 = 16 = $8x2^1$
	A = A 〈〈 3;	A = 0x40; //A = 01000000 = 64 = $8x2^3$
	A = A 〉〉 1;	A = 0x04; //A = 00000100 = 4 = $8/2^1$
	A = A 〉〉 3;	A = 0x01; //A = 00000001 = 1 = $8/2^3$

다음은 비트 연산자를 어떻게 사용하는지 살펴보자. 입력되는 정수값을 2진수로 표현했을 때 1이 몇 개가 있는지 알아내는 프로그램이다.

코드 AA-3 2진수에 포함된 1의 개수 알아내기

```
#include "stdio.h"
//-------------------------------------------------
int countOne(unsigned int input)
{
```

```
    unsigned int i, count, flag;

    count = 0;

    // 가장 오른쪽에 있는 비트값 하나만 1로 만들고 나머지는 0으로 초기화한다
    flag  = 0x00000001;

    for(i = 0; i < 32; i++)
    {
        // 입력된 값의 비트 위치와 flag의 비트 위치를 비교해서
        // 비트값이 모두 1이면 count를 1 증가시킨다
        if(input & flag)
            count++;

        // flag의 하나뿐인 1을 왼쪽으로 한 비트 이동시킨다
        flag <<= 1;    //flag = flag << 1;
    }

    return count;
}
//-----------------------------------------------
int main()
{
    printf("%8d : %2d\n", 8, countOne(8));
    printf("%8d : %2d\n", 1234, countOne(1234));
    printf("%8d : %2d\n", 12345678, countOne(12345678));
}
```



APPENDIX B | 코드를 작성하는 방법

충분히 생각한 내용을 문서로 만든 후 코딩을 시작하자

프로그래밍한다는 것은 코딩하는 것만을 의미하지 않는다. 구현해야 할 것이 정해지면 어떻게 구현해야 하는지 정보를 찾고 준비하는 과정이 매우 중요하다. 준비 작업 없이 코딩 먼저 시작하면 프로젝트를 완료하는 데 더 많은 시간이 소요된다.

정확하지 않은 알고리즘은 항상 수정 작업을 발생시키며 심한 경우엔 처음부터 다시 코딩하는 상황을 만들기도 한다. 따라서 코딩하기 전에 충분히 생각해야 한다. 또한 자료를 찾아 어떻게 구현할 것인지를 머릿속에 그려야 한다.

알고리즘이나 자료구조를 충분히 생각했다면 코딩 전에 문서로 만들어 정리하면 가장 좋다. 프로그래밍은 코드를 작성하는 일만 의미하는 것이 아니라 문서 정리도 포함한다. 필자는 어떻게 구현해야 하는지 머릿속에 떠오르지 않으면 한 줄도 코딩하지 않는다. 그렇지만 다른 사람과 비교하더라도 충분히 프로젝트를 빨리 끝내는 편이다.

함수명과 변수명을 생각하고 만들자

코딩할 때 프로그램을 어떻게 만들지만 고민하지 말고, 좋은 함수명과 변수명을 만들려고 노력해야 한다. 이는 좋은 코드를 작성하는 기본이다.

좋은 함수명과 변수명은 이름만 봐도 함수를 무엇 때문에 만들었는지 알 수 있게 한다. 또한 짧은 이름으로 함수 특성을 나타내면 좋지만, 짧은 이름에 얽매이는 것보단 길게 이름을 쓰더라도 특성을 나타내는 이름이 더 좋은 이름일 수 있다. 글자 수를 줄이려고 짧은 이름을 고집하는 일은 피하기 바란다.

함수명과 변수명을 지을 때는 반드시 영어 때문에 고민한다. 왜냐하면 한글로 코드를 작성할 수는 없기 때문이다. 그렇다고 오랜 시간을 투자해도 쉽게 능률이 오르지 않는 영어를 또다시 공부하면서 스트레스 받을 필요는 없다. 요즘에는 인터넷에서 제공하는 사전이 많으므로 한글로 생각한 멋진 단어를 인터넷 사전에서 검색해 멋진 영어 이름으로 바꿔 코드에 반영하면 된다.

좋은 이름을 가진 코드는 이해하기도 쉽고, 무엇보다 아름답다는 사실을 다시 한번 강조한다.

밑그림 먼저 그리자

코딩을 시작하지 못해 멍하니 있는 초급 프로그래머의 모습을 본 적이 있다. 이런 프로그래머라면 큰 밑그림을 먼저 그려본 후 넓은 시각에서 프로그램을 바라보라고 말해주고 싶다.

프로그래밍도 그림을 그리는 것처럼 밑그림(큰 골격)을 만들고 각각의 부분을 세세하게 만들어 나가는 일이다. 따라서 코딩 처음에는 주요 기능을 하는 함수들을 선언만 해두고 컴파일해보면 좋다. 구체적으로 설명하면 클래스를 만들어야 하면 클래스 골격을 만들어 컴파일을 실행하고, C처럼 함수가 필요하면 함수명만 만들어 놓고 컴파일하면 된다. 이 작업이 프로그래밍에서 밑그림을 그리는 것이다.

클래스나 함수 안을 어떻게 구현할지 처음부터 고민할 필요는 없다. 프로그램은 글자가 모여 줄이 되고, 줄이 모여 완성된다. 조금씩 만들어 나가는 것이 중요하다.

컴파일을 많이 할수록 좋다

컴파일은 수시로 하면 좋다. 가능하다면 한 라인을 구현하고 컴파일해도 좋다. 에러가 있다면 방금 작성한 곳에서 에러가 발생한 것이니 얼마나 찾기 쉬운가? 물론 컴파일 시간이 길게 소요된다면 최소화해야겠지만 단 몇 초라면 자주 하는 것이 좋다.

프로그래밍을 많이 한 사람일수록 컴파일 횟수는 줄어든다. 그렇다고 컴파일 횟수가 현격히 줄어드는 것은 아니다. 디버깅하면서 무수히 많은 컴파일을 실행하기 때문이다.

테스트 구문 작성의 중요성

필자는 윈도우 프로그래밍을 좋아하지는 않는다. 완성했을 때는 멋진 프로그램이지만 만드는 과정에서는 실행해야 하는 테스트 과정이 불편해서 텍스트 기반의 프로그램보다 신경 써야 할 부분이 많기 때문이다(특히 윈도우 프로그래밍은 테스트 때문에 버튼을 누를 때가 많다).

사실 테스트는 컴파일할 때 같이 이루어지므로 무수히 많이 실행한다. 따라서 입력이 필요한 프로그램이라도 테스트를 실행하면서 직접 입력하게 해서는 안 된다. 즉, 입력을 받는 C의 scanf() 함수는 프로그램 작성 중에는 사용하지 말고 프로그램을 다 만든 후에 사용하는 것이 좋다.

또한 프로그래밍할 때 main() 함수(시작 함수)에 테스트 구문을 작성해서 테스트용 함수로 많이 활용하자. 이때 테스트를 위한 구문 작성을 귀찮아하면 안 된다. 타이핑 시간을 줄이는 것보다 수시로 이루어지는 테스트 시간을 줄이는 것이 훨씬 효율적이다.

마지막으로 테스트 구문은 키보드를 한 번만 누르면 모든 테스트를 실행할 수 있게 만들면 좋다. 그래야 컴파일을 수시로 하더라도 문제를 확인하면서 프로그래밍할 수가 있다.

단위 테스트 없이 코딩하면 하수다

코드를 작성하면서 컴파일 에러가 없으면 코드 작성이 끝났다고 생각하는 사람들이 있다. 의외로 대단히 많다. 이럴 때 자칫 잘못하면 문법 에러Syntax Error는 없어도

고치기 어려운 논리 에러Logical Error가 많은 코드가 된다. 그리고 컴파일만 되는 코드를 작성하는 사람은 프로그램 완성 시간이 상당히 길고, 에러를 찾는 데도 상당히 힘들어한다.

프로그래밍은 코딩하는 중간마다 단위 테스트Unit Test를 계속 실행해야 한다. 이전 코드가 잘 작동하는지 확인한 후 다음 코드를 구현하는 것이 에러가 적으면서도 프로그램을 빨리 만드는 비결이다. 자신이 생각한 알고리즘을 아무리 확신해도 코드에 에러가 없는 것은 아니다(필자의 경험상 에러 없는 코드를 구현해본 적이 단 한 번도 없다). 함수를 하나 만들어도 10줄 이상만 넘어가면 에러를 발견할 수도 있다. 즉, 중간에 단위 테스트를 하지 않으면 나중엔 찾을 수 없는 에러에 스트레스를 받으며 밤잠을 설칠 수도 있다.

예를 들어 if문에서 '==' 대신에 '='을 사용하거나 '&&' 대신에 '&'을 사용해도 무난하게 컴파일할 수 있다. 그러나 값이 같은지 확인하는 연산자와 값을 할당하는 연산자가 같을 리는 없다. 이런 에러처럼 나중에 찾기 어렵고 디버깅하기 어려운 것은 없다.

모듈 단위로 구현하라

프로그램이 커질 때를 대비하려면 모듈module 단위로 구현해야 한다. 필자는 새로운 프로그램을 만들 때 간단한 동작을 하는 함수를 만들고 정상적으로 동작하면 이 함수를 큰 프로그램에 삽입하는 방식으로 프로그래밍한다. 혼자서 진행하는 프로젝트라도 프로그램이 커진다면 이처럼 모듈 단위로 진행하는 것이 효과적이다.

좋은 주석은 나를 위한 것이다

프로그래머들이 귀찮아하는 일 중 하나가 주석을 적는 것이다. 물론 자신이 만드는 간단한 프로그램이라면 주석을 적는 일은 귀찮은 일이다. 그런데 크기가 크거나 알고리즘이 복잡하다면 주석을 적는 것이 좋다.

처음 프로그램을 시작하는 사람에게 주석은 굉장히 중요하다. 특히나 회사에서 업무를 처리하는 것이라면 주석을 적는 일만큼 중요한 일이 없다. 누군가 회사를 떠나더라도 떠난 사람이 작성한 코드를 빨리 이해해야만 여러 가지 업무를 효율적으로 진행할 수 있기 때문이다. 입장을 바꿔 남이 만든 주석 없는 코드를 이해하기 어려운 상황이라고 생각해보자. 내가 만든 프로그램도 주석 없이 이해하기가 어려운 마당에 하물며 남이 만든 코드라면 짜증 날 것이다.

APPENDIX C | 디버깅 방법

Debugging은 나의 친구다

에러 없는 코드를 작성할 수 있다면 얼마나 좋을까? 그러나 그건 희망 사항일 뿐이다(코드를 작성하면서 에러가 없다면 아마 'Hello World' 같은 쉬운 프로그래밍일 것이다).

에러는 프로그래밍의 동반자다. 다행히도 필자는 어느 정도 에러에 내성이 생겨서 에러를 미워하지 않는다. 오히려 에러는 무능력함이 만든 친구라고 생각하면서 에러를 당연한 듯 받아들인다. 에러를 받아들일 마음이 없다면 프로그래머가 천직이 아니라고 생각해도 좋다.

바라건대 여러분도 디버깅을 친구로서 받아들인다면 프로그래머로 사는 삶이 외롭지 않을 것이다. 그리고 에러를 해결해 나가는 즐거움을 느껴라.

출력 함수를 가장 먼저 확인하자

지금까지 프로그래밍해 본 적 없는 Objective-C로 프로그램을 만들어야 할 일이 있었다. 물론 가장 먼저 책을 샀지만 코드를 작성하기 전 가장 먼저 살펴본 것은 디버깅 방법이었다.

우리가 'Hello World'를 배우면서 가장 먼저 접하는 것이 출력 함수다. 그런데 출력 함수는 사실 실무 프로그래밍 코드에 많이 사용한다기보다는 코딩 중간에 에러가 발생했을 때 직접 출력값을 적거나 메시지가 정상적으로 출력되는지를 확인하는 등 전체 소스 코드에서 어디까지 정상적으로 동작하는지 확인하는 용도로 많이 사용한다.

프로그래밍 언어의 입문서에서 가장 먼저 출력 함수를 설명하는 것은 그만큼 디버깅의 중요성을 무의식적으로 설명하는 것인지도 모른다.

main() 함수부터 문제 발생 원인을 찾자

에러 원인을 찾는 방법의 기본은 main() 함수를 확인하는 것이다. main() 함수 안의 모든 코드를 주석처리 한 후 main() 함수의 첫 부분부터 하나씩 주석 처리를 지우면서 실행해보면, 에러가 발생하는 지점을 찾을 수 있다.

main() 함수에서 에러가 발생한 부분이 함수라면 main() 함수와 같은 방법으로 해당 함수의 에러를 찾으면 된다. 그러면 그때는 정말 문제가 발생한 원인이 되는 에러 부분을 발견할 수 있다. 다소 무식한 방법이지만 의외로 효과가 있다. 컴파일러가 알려주는 에러 메시지가 명확하지 않다면 이렇게 해서라도 찾아야 한다.

다른 한 가지 방법은 루틴 테스트routine test라는 출력 함수를 구문 중간에 작성해서 출력되는지를 알아보는 것이다. for문이나 while문 안에서 무한 루프가 발생할 때는 이 방법을 사용하면 좋다.

의심스러운 변수값은 반드시 출력하자

코딩할 때 변수값이 의심스러우면 바로 값을 확인해봐야 한다. 즉, 막연히 변수값이 맞겠지 하는 생각을 하지 말고 테스트 구문을 바로 작성해 출력해야 한다.

디버깅은 꾸준한 인내가 필요하다. 예를 들어 수학을 잘하는 사람들이 프로그램을 잘하는 편인데, 머리가 좋아서 프로그래밍을 잘하는 것이 아니라 직면한 문제를 꾸준히 해결해 가는 과정을 잘 참고 견디는 습관이 있기 때문이다. 디버깅은 테스트 구문 한 줄을 기꺼이 작성하는 사람이 잘할 수밖에 없다. 의심스러운 것은 무엇이든 확인하라.

인터넷 검색을 활용하자

프로그래머에게 인터넷은 필수다. 특히 디버깅할 때는 책보다 더 많은 정보를 알려주는 것이 인터넷이다. 컴파일할 때 에러가 발생하면 에러 구문을 구글에서 검색하자. 다양한 해답을 검색할 수 있다. 보통 한국 사람들은 Daum이나 네이버에서 검색하는 경향이 있는데, 영어에 어려움이 있어도 구글을 이용하는 것이 좋다. 프로그래밍 분야에서만큼은 한국 사이트보다 더 많은 정보를 얻을 수 있다.

APPENDIX D | 코드를 분석하는 방법

코드의 골격을 먼저 확인하자

다른 사람의 코드를 살펴보는 일은 어렵다. 왜냐하면 알고리즘을 알고 보는 것이 아니기 때문이다. 그래서 단순히 코드를 복사하고 붙여넣기 하는 사람이 많이 생기는지도 모르겠다.

코드를 잘 분석하는 사람은 구현부(소스 코드 내용)를 처음부터 살펴보지 않는다. 주석과 함수명을 살펴보면서 함수가 하는 일을 알아내려고 한다. 다른 사람의 코드를 살펴볼 때는 함수가 어떤 일을 하는지만 파악한 후 main() 함수에서 무엇부터 실행하는지를 살펴보면 좋다. 즉, 코드의 골격을 먼저 살펴보고 모든 것을 이해해야 할 때만 함수 내용을 살펴보자.

함수가 하는 일을 먼저 파악하자

일반적으로 올바르게 코드를 작성하는 사람이라면 함수 구현부에 함수 특징을 주석으로 적어둔다. 물론 함수명을 통해 함수의 특징을 알 수도 있지만 좀 더 자세한 특징을 알려면 주석과 함수의 매개변수를 잘 살펴볼 필요가 있다. 함수의 내용이 짧다면 문제 되지 않겠지만 길다면 자세히 보는 일은 나중으로 미뤄라.

구조체 역할을 확실히 파악하라

프로그래밍 언어의 입문서에서는 구조체를 간단히 설명한다. 그러나 프로그램에서 가장 많이 사용하는 것 중 하나가 구조체다. 구조체는 함수 사이에 주고받는 데이터를 통합해서 사용할 수 있게 하므로 구조체를 위해서 함수를 만든다고도 볼 수 있으며, 구조체 역할을 파악하면 해당 프로그램의 성격을 알 수 있다. 사실 객체 지향 프로그래밍의 클래스 개념은 구조체 개념을 확장해서 발전시킨 것이기도 하다.

함수 구현부의 알고리즘은 마지막에 파악하자

함수의 세세한 부분을 보면서 알고리즘을 파악하는 일은 프로그램의 골격을 파악한 후 맨 나중에 해야 한다. 사실 함수의 알고리즘을 파악하는 일은 가장 어려우며 많은 시간을 할애해야 한다(어쩌면 코드를 분석하다 지쳐 프로그래밍에 흥미를 잃을 수도 있을 정도다). 또한 코드를 알아보기 쉬운 함수명과 변수명으로 작성했다면 분석하는 일이 쉬울 수 있으나 보통은 어떤 특정한 규칙을 가지지 않고 작성하는 경우가 많으므로 알고리즘 분석은 상당히 어려운 일이다. 따라서 앞에서 설명한 분석에 익숙해지고 나서 가장 마지막에 파악하면 좋다.

집필을 마치며

집필하면서 책을 쓴다는 일이 쉽지 않음을 알았고, 내가 알고 있는 내용을 다른 사람에게 전달한다는 일이 참 어렵다는 것을 다시 한번 느끼게 되었다. 처음에는 어찌 보면 간단하고 쉬운 주제임에도 누구나 쉽게 이해할 수 있게 책을 쓰겠다고 생각했는데, 책을 쓰는 마무리 단계에서는 '나도 참 어렵게 썼구나'라고 생각하게 되었다. 설명이란 초등학생도 알 수 있을 정도로 쉽게 하는 것이 좋다는 것을 알지만 C를 충분히 이해해야만 쉽게 볼 수 있는 책이 된 것 같아 아쉽다.

책의 마지막 글을 쓰는 지금, 여러분에게 말하고 싶은 것은 이 책은 완성이 아니라는 것이다. 그렇다고 내용이 부족하다는 말은 아니다. 여러분이 이 책의 내용을 바탕으로 코드를 자유자재로 응용해서 만들었을 때가 되어야 이 책이 진정하게 완성되었다고 생각하기 때문이다. 이 책에서 소개한 소스 코드는 앞으로 좀 더 유용하게 사용할 수 있는 프로그램을 만드는 기본 토대일 뿐이다. 예를 들어 비교 함수에서 더 많은 연산자를 만들 수도 있고, 더 나아가 정수가 아닌 실수 연산을 구현한다면 훨씬 유용한 프로그램이 될 것이다. 그리고 항상 필자가 쓴 내용이 정답이 아닐 수 있다는 생각을 하기 바란다. 프로그래밍에 정답이란 없다. 필자가 소개한 이 책도 하나의 프로그래밍 방법일 뿐이며 Big Number 연산을 꼭 필자처럼 구현할 필요는 없다.

필자는 특히 RSA 연산을 직접 만들어 구현한 후 올바르다는 사실을 알게 되었을 때의 희열이 상당히 컸고 오래갔다. 평소 프로그래밍을 하면서 생각했던 알고리즘이 중요하다는 사실을 새삼 다시 느낀 것이다. 보통 알고리즘에서 막히고 알고리즘 때문에 괴로워한다. 알고리즘이 존재한다면 코드 구현은 단순 작업일 정도다. 필자는 현재 'How-to Programming Series'라는 큰 주제의 첫 책으로 Big Number

연산을 선택했다. 이는 프로그래밍의 기본기를 단단하게 다지는 데는 몇 개의 알고리즘을 직접 만들어보고 희열을 많이 느끼는 일이 가장 좋다고 생각하기 때문이다. 그리고 Big Number 연산이 그러한 기본기를 다지는 알고리즘을 생각하는 능력을 키울 수 있다고 판단하기 때문이기도 하다.

사실 간단한 주제로 책을 쓰는 것이 아니냐는 생각을 하곤 한다. 잘하는 사람이 본다면 뭐 이런 주제로 책까지 쓰냐고 할 수도 있기 때문인데, 쉽더라도 많은 자료를 정리해서 남긴다면 모두의 자산이 될 것이다. 바람이라면 필자보다 실력 있는 사람들이 간단한 주제로라도 책으로 많이 알려주었으면 하는 것이다.

첫 책을 출판하면서 책이 완성되기까지 여러 사람의 노력과 시간이 합쳐져야 함을 알게 되었다. 초라한 원고 초안을 독자분들이 읽을 수 있는 책으로 편집해준 한빛미디어 이중민 대리께 진심으로 감사드린다.

마지막으로 끝까지 읽어주신 독자분께 감사드린다.

01
유지보수하기 어렵게 코딩하는 방법

평생 개발자로 먹고 살 수 있다

–

로에디 그린 지음
우정은 옮김

이렇게 하면
개발자로 평생
먹고 살수 있다

02
대용량 서버 구축을 위한

Memcached와 Redis

–

강대명 지음

대용량 서버 구축을 위한
분산 캐시의 이해!
간단한 Short URL 실습
으로 이해하는
분산 캐시 기술

03
스마트폰과 태블릿 호환을 위한

안드로이드 앱 프로그래밍

–

고강태 지음

스마트폰 앱을 태블릿 앱
으로 쉽고, 빠르게 전환
하고 싶으신가요?
다양한 기기와 호환하는
안드로이드 앱 개발의
모든 것!

04
프로젝트 성공을 위한 갑과 을의 상생협력

21명의 현직 전문가가 전하는 생생한 성공 노하우

–

이재용 지음

갑과 협력하기 어려우신가요? 을과 협력하기
어려우신가요?
21명의 현직 전문가가
알려주는 갑과 을의
상생협력 전략

05
자바 개발자를 위한

함수형 프로그래밍

–

딘 왐플러 지음 / 임백준 옮김

자바 개발자을 위한
함수형 프로그래밍 기법
함수형 프로그래밍 기법을
익힌 프로그래머와
그렇지 않은 프로그래머가
작성하는 코드의 품질은
완전히 다르다

06
일관성 있는 웹 서비스
인터페이스 설계를 위한

REST API 디자인 규칙

–

마크 마세 지음
김관래, 권원상 옮김

REST API를 사용하기
어려우신가요?
사전처럼 찾아 쓰는
REST API 규칙과 팁

07
웹 프로그래머를 위한

서블릿 컨테이너의 이해

–

최희탁 지음

웹 프로그래밍에 깊이를
더 하자!
서블릿 컨테이너를
제대로 알면
웹 프로그래밍이 쉬워진다

08
자바스크립트와 SVG로 쉽게 만드는

웹 기반 데이터 비주얼라이제이션 D3

–

마이크 드워 지음
양성일 옮김

쉽게 배우는
웹 기반 데이터
비주얼라이제이션 D3

09
멀티스레드를 위한

자바스크립트 프로그래밍 웹 워커

–

아이두 그린 지음
김보경 옮김

웹 애플리케이션에서
멀티스레드를 구현하는
가장 쉬운 방법.
웹 애플리케이션에
날개를 달자!

10
소프트웨어
생명 연장을 위한 원칙

Code Simplicity

–

맥스 카넷-알렉산더 지음
신정안 옮김

어떻게 해야
소프트웨어 설계를
잘할 수 있을까?
좋은 소프트웨어 설계는
무엇일까?

11

Thinking About
C++ STL 프로그래밍
–
최흥배 지음

C++ 프로그래밍에서는 STL은 필수다 STL을 아는 만큼 C++ 프로그래밍 스킬을 키울 수 있다

12

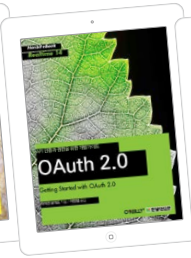
빅데이터 처리를 위한 웹 개발 노하우
MongoDB와 PHP
–
스티브 프랜시아 지음
최병현 옮김

MongoDB와 PHP의 만남, 빅데이터 처리 실전 노하우

13

파이썬과 Boto로 클라우드 관리하기
파이썬을 이용한 AWS 가이드
–
미치 가나트 지음
강권학 옮김

파이썬으로 AWS(아마존 웹 서비스)를 할 수 있다고요? 파이썬과 Boto로 AWS를 사용해보자



14

API 인증과 권한을 위한 개발가이드
OAuth 2.0
–
라이언 보이드 지음
이정림 옮김

웹 API 인증을 위한 만능 키, OAuth 2.0을 만나다

15

윤리적인 빅데이터 사용을 위한 정책 가이드
빅데이터 윤리
–
코드 데이비스 지음
강석주 옮김

빅데이터 시대! 윤리적인 책임은 없는가? 빅데이터 시대에서 지켜야 할 윤리 책임의 기준을 제시한다

16

윈도우 런타임을 이용한 실전 앱 개발
Windows 8 앱 개발 가이드
–
벤 듀이 지음 / 이원영 옮김

윈도우 런타임(Windows Runtime, WinRT)으로 시작하는 새로운 개발 세계

17

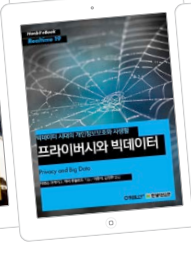
Xen으로 배우는 가상화 기술의 이해
CPU 가상화
–
박은병, 김태훈, 이상철, 문대혁 지음

반가상화와 전가상화 기술을 다루는 x86 아키텍처 기반의 CPU 가상화!

18

JSP 바이블
STEP 01
JSP 시작과 개발환경 구축
–
조효은 지음

JSP를 마스터하기 위한 길잡이 JSP를 위한 최고의 바이블 시리즈 첫 번째 책



19

빅데이터 시대의 개인정보보호와 사생활
프라이버시와 빅데이터
–
테렌스 크레이그, 메리 루틀로프 지음
이춘식, 김정환 옮김

빅데이터 시대! 개인정보와 프라이버시는 과연 보호받을 수 있을까?

20

프로그래머를 위한 가이드
드루팔
–
제니퍼 하지던 지음
김지원 옮김

드루팔 개발을 위한 원칙과 팁 이렇게 하면 실수를 피할 수 있다