Number Theoretic Algorithms in Competitive Programming

rkm0959

Version 0.1

§0.1 Introduction

이 책은 Competitive Programming 또는 Algorithmic Problem Solving을 즐기는 분들을 위한 정수론 입문서입니다. 범위는 기초부터 Project Euler 문제를 본격적으로 해결할 수 있는 수준까지 넓게 다루고자 합니다. 퀄리티를 신경쓰는 대신 완성 자체에 목적을 둔 초안이니, 오탈자가 많을 수 있습니다. 차근차근 수정해나가면 좋을 것 같습니다.

사람마다 생각이 다 다르겠지만, 저는 기본적으로 CP나 PS를 하는 사람이 알고리즘을 공부하는 접근 방법에는 크게 두 가지가 있다고 생각합니다.

- 알고리즘의 원리 및 증명을 완전히 이해하고 이를 문제 풀이에 적용한다
- 알고리즘 자체를 하나의 Black Box로 두고, 이를 활용하여 문제를 푸는 법을 익힌다

예를 들어, Network Flow 분야의 문제는 후자의 접근을 택한 사람이 많을 것입니다. 팀노트가 있는 대회에서는 후자의 방법이 나름 강력한 힘을 발휘할 수 있으나, 팀노트가 없는 경우, 또는 문제가 매우 어려운 경우에는 알고리즘을 Black Box로 취급하는 공부 방법으로는 한계가 보이기도 합니다. 하지만 이러한 Black Box 접근 방식은 알고리즘을 공부하는 진입장벽이나 접근 난이도를 매우 낮출 수 있다는 점에서 긍정적인 측면도 있습니다. Flow 비유를 계속해보면, 제가 Flow에 대한 증명 기법을 몰라도, 문제를 풀 수 있는 모델링을 잘 할 수 있으면 문제는 풀리니까요. 이렇게 두 접근에는 분명히 장단점이 있습니다.

PS/CP의 수학에서도 비슷한 생각을 할 수 있습니다. 모든 알고리즘의 원리를 정확하게 공부하는 접근을 택할 수도 있으며, 각 알고리즘을 Black Box로 보고 이를 활용하여 문제를 푸는 방법을 익힐 수도 있습니다. 이 책은 **두 가지 접근을 모두 존중하는 것을 목표**로 하고 있습니다. 이를 위해서, 책의 구성을 다음과 같이 하기로 계획하고 있습니다.

- 맨 처음에, 소개할 알고리즘으로 **할 수 있는 것이 무엇인지** 소개합니다.
- 알고리즘이 이루는 바가 무엇인지 **기본적인 이해를 갖추기 위한** 지식을 소개합니다.
- 알고리즘의 **작동 과정과 원리 및 증명**을 설명합니다.
- 수학적 엄밀성과 깊이를 더욱 원하는 분들을 위하여, **수학 문제를** 제공합니다.
- 문제풀이 연습을 하기 위하여, **알고리즘 문제 예시를** 제공합니다.
- 다른 사람의 설명을 읽고 싶은 분을 위하여, **여러 좋은 링크들을** 제공합니다.

Contents

I

Fundamentals of Number Theory

1 Basic Facts & Algorithms

이번 Chapter에서 다루는 내용은

- 약수, 배수, 합동, 소수, 소인수분해의 개념
- 약수를 계산하는 $\mathcal{O}(\sqrt{n})$ 알고리즘
- 소인수분해를 계산하는 $\mathcal{O}(\sqrt{n})$ 알고리즘
- 1 이상 n 이하의 소수를 $\mathcal{O}(n \log \log n)$에 구하는 에라토스테네스의 체

§1.1 Basic Number Theory

이번 section에서는 기초적인 정수론을 다룬다. 모든 논의를 이해하기 위해 필수적이다.

Definition 1.1.1 (약수와 배수)**.** 정수 a, b가 있을 때, a가 b의 **약수**라는 것은 정수 n이 있어 $b = an$이라는 것이다. 이때 b를 a의 **배수**라고 한다. 이를 표기하는 방법은 $a|b$.

Definition 1.1.2 (modulo의 개념)**.** 정수 a, b와 정수 $n \neq 0$이 있을 때, $a \equiv b \pmod{n}$이라는 것은 $n|(a-b)$가 성립한다는 것이다. n으로 나눈 나머지가 같다고 생각하면 쉽다. 흔히, 이를 a와 b가 $\pmod{n}$에서 **합동**이라고 말한다. 영어로는 "congruent".

Example 1.1.3
$7|(27-13) = 14$이므로, $27 \equiv 13 \pmod{7}$.

Theorem 1.1.4 (Division Algorithm)
정수 a와 $b \neq 0$이 있을 때, **유일한** 정수 q, r이 존재하여 $a = bq + r$이고 $0 \leq r < |b|$이다. 이때 흔히 q를 **몫**이라고 하고, r을 **나머지**라고 부른다.

Example 1.1.5
$47 = 5 \cdot 9 + 2$이고 $0 \leq 2 < 5$이므로 47을 5로 나눈 몫은 9, 나머지는 2.

modulo의 중요한 성질 중 하나는 이를 equivalence relation으로 볼 수 있다는 것이다.

Theorem 1.1.6 (Congruence as an Equivalence Relation)
(Reflexivity) : $a \equiv a \pmod{n}$이 성립한다.
(Symmetry) : $a \equiv b \pmod{n}$이면 $b \equiv a \pmod{n}$이 성립한다.
(Transitivity) : $a \equiv b \pmod{n}$, $b \equiv c \pmod{n}$이면 $a \equiv c \pmod{n}$이다.
즉, $a \equiv b \pmod{n}$은 equivalence relation이다.

equivalence relation의 중요한 성질은 집합을 partition 한다는 것이다. 이는 modulo와 합동의 context에서는, 결국 각 $n \neq 0$에 대하여 정수의 집합이 n으로 나눈 나머지가 0인 정수로 이루어진 집합, 1인 정수로 이루어진 집합, $\cdots$, $|n|-1$인 정수로 이루어진 집합으로 partition 된다는 것이다. 이렇게 집합이 나뉘어지면, 두 정수가 $\pmod n$에서 합동인 것은 두 정수가 같은 집합에 속하는 것과 완전히 동치이다. 말은 복잡하게 했으나, 결국

- a는 a를 n으로 나눈 나머지와 $\pmod n$에서 합동이며
- 두 정수가 n으로 나눈 나머지가 같은 것과 $\pmod n$에서 합동인 것은 동치

라는 말이다. 이래서 a를 n으로 나눈 나머지를 $a \pmod n$이라고 표기하기도 한다.

Definition 1.1.7 (소수)**.** 자연수 $p \geq 2$의 약수가 1과 p 뿐이면, p를 **소수**라고 부른다.

소수는 영어로 "prime"이다. 또한, p가 소수일 때 p^k를 prime power라고 부른다.

Example 1.1.8
$17, 31, 47, 10^9+7$은 소수이다. $36, 420$은 소수가 아니다.

소수에 대한 중요한 성질 중 하나로 다음 정리가 있다.

Theorem 1.1.9
정수 a, b와 소수 p가 있을 때, $p|ab$인 것은 $p|a$ 또는 $p|b$가 성립하는 것과 동치이다.
일반적으로, $p|a_1a_2\cdots a_k$임은 $p|a_i$인 $1 \leq i \leq k$가 존재하는 것과 동치다.

다음은 이미 모두가 알고 있는 사실일 것이다.

Theorem 1.1.10 (Fundamental Theorem of Arithmetic)
각 자연수 $n \geq 2$는 소수들의 곱으로 유일하게 표현할 수 있다. 이때 소수들의 순서만 다른 경우는 같은 표현으로 본다. 즉, $p_1 < p_2 < \cdots < p_k$와 $e_1, e_2, \cdots, e_k \geq 1$을 만족하는 소수 $\{p_i\}$와 자연수 $\{e_i\}$가 유일하게 존재하여

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

이와 같은 n의 표현을 구하는 것을 n을 **소인수분해**한다고 한다.
이때, p_i들을 n의 **소인수**라고 부르고, $e_i = \nu_{p_i}(n)$이라고 쓰기도 한다.
또한 e_i를 "n이 갖는 p_i의 개수 (또는 지수)"라고 부르도록 하겠다.

Example 1.1.11
420의 소인수분해는 $2^2 \cdot 3 \cdot 5 \cdot 7$.

소인수분해는 정수론에서 다루는 가장 중요하고 어려운 문제 중 하나로, n의 소인수분해 결과를 알면 여러 정보를 쉽게 계산할 수 있다. 이미 익숙할 것으로 예상하는 결과는

Theorem 1.1.12

$n = p_1^{e_1} \cdots p_k^{e_k}$라 하면, n의 양의 약수의 집합은

$$d = p_1^{f_1} \cdots p_k^{f_k}, \quad 0 \leq f_i \leq e_i \quad \forall 1 \leq i \leq k$$

형태로 나타나는 자연수들의 집합과 같다. 특히, n의 약수의 개수를 $\tau(n)$이라 하면

$$\tau(n) = \prod_{i=1}^{k} (e_i + 1)$$

일단은 이 정도만 하고, 다른 이론은 연관된 알고리즘과 동시에 배우도록 하자.

여기서 설명을 하지는 않았지만, Theorem 1.1.9, 1.1.10, 1.1.12는 직접 증명하기에 생각보다 쉽지 않은 내용이다. 이들을 증명하려면 Chapter 2에서 Euclidean Algorithm과 Bezout's Identity에 대한 공부가 필요하다. 하지만 위 정리들은 이미 교과과정에서 증명없이 학습한 내용일 것이므로, Chapter 1에서는 언급만 하고 넘어가도록 한다. 이들에 대한 엄밀한 증명은 정수론 교과서를 참고하거나, Chapter 2의 여러 연습문제를 참고하자.

§1.2 Basic Sqrt-Time Algorithms

이번 section에서는 약수, 소인수분해를 계산하는 $\mathcal{O}(\sqrt{n})$ 알고리즘을 각각 소개한다.

약수 : 알고리즘의 핵심 아이디어는

Proposition 1.2.1
(1) : 자연수 d가 n의 약수라면, n/d 역시 n의 약수이다.
(2) : 이 때, d 또는 n/d는 $\sqrt{n}$ 이하의 값이다.

Proof. (1) : $n = d \cdot (n/d)$이고 $d, n/d$가 각각 자연수이므로 증명 끝.
(2) : d, n/d가 모두 $\sqrt{n}$보다 크다면 그 곱인 n이 $(\sqrt{n})^2 = n$보다 크게되어 모순. □

그러므로, $\sqrt{n}$ 이하의 약수만 모두 찾으면 n의 약수를 모두 찾을 수 있다.

Algorithm 1.2.2 (Finding Divisors)
다음과 같은 과정으로 n 이하의 약수를 모두 찾을 수 있다.

- $1 \le d \le \sqrt{n}$인 자연수 d 중 n의 약수인 것을 모두 찾는다.
- 각 약수 $d|n$에 대하여, n/d 역시 n의 약수이다.

특히, $n = m^2$이 제곱수인 경우 $\sqrt{n} = m$이 약수로 두 번 세지는 경우를 조심해야 한다.

Example 1.2.3
45의 약수를 모두 구해보자. $6 < \sqrt{45} < 7$이므로 6 이하의 약수를 먼저 모두 구한다.
순서대로 나눠보면 $1, 3, 5$가 약수임을 쉽게 확인할 수 있다.
이에 따라 6 초과의 약수는 $45/1, 45/3, 45/5$, 즉 $45, 15, 9$가 된다.

Remark : 이 방법으로 약수를 모두 구할 수 있으니, 소수 판별 역시 똑같이 할 수 있다.

소인수분해 : 먼저 확인할 사실은

Proposition 1.2.4

자연수 n은 $\sqrt{n}$보다 큰 소인수를 중복을 포함하여 최대 하나 가질 수 있다.

Proof. $\sqrt{n}$보다 큰 소인수가 두 개 이상이었다면, 그 두 개만 곱해도 n을 넘는다. □

그러므로 n의 소인수 중 $\sqrt{n}$ 이하인 것을 전부 나누어준다면, 그 결과는 1이거나 $\sqrt{n}$보다 큰 n의 소인수가 된다. 어느 경우던 상관없이 우리는 n의 소인수분해를 마칠 수 있다.
이를 종합하면 다음과 같은 알고리즘을 설계할 수 있다.

Algorithm 1.2.5 (Finding Prime Factorization)

$A = \lfloor \sqrt{n} \rfloor$을 먼저 계산하자. 그 후,

- 각 $i \in [2, 3, \cdots, A]$를 순서대로 보면서, i가 n의 **소인수**인지 판별한다.
- 만약 그렇다면, n이 i의 배수가 아닐 때까지 $n \leftarrow n/i$를 반복한다.
- 이때 i는 기존 n의 소인수이며, 대응되는 지수 e는 i를 나누어준 횟수다.
- i에 대한 iteration이 끝난 뒤, $n \neq 1$이라면 이 역시 기존 n의 소인수다.

위 알고리즘의 문제는 i에 대한 iteration 과정에서 i가 n의 **소인수**인지 판별한다는 것이다. 이는 단순히 i가 n의 약수인 것이 아니라, i가 소수임까지 확인하라는 의미를 담고 있다. 그런데 실제로는 위 알고리즘에서 i가 소수인지 확인할 필요가 없다.

Algorithm 1.2.6 (Finding Prime Factorization)

$A = \lfloor \sqrt{n} \rfloor$을 먼저 계산하자. 그 후,

- 각 $i \in [2, 3, \cdots, A]$를 순서대로 보면서, i가 n의 **약수**인지 판별한다.
- 만약 그렇다면, n이 i의 배수가 아닐 때까지 $n \leftarrow n/i$를 반복한다.
- 이때 i는 기존 n의 소인수이며, 대응되는 지수 e는 i를 나누어준 횟수다.
- i에 대한 iteration이 끝난 뒤, $n \neq 1$이라면 이 역시 기존 n의 소인수다.

Example 1.2.7

153을 소인수분해한다. $A = \lfloor \sqrt{153} \rfloor = 12$이고,

- 2는 약수가 아니지만 3은 약수가 맞고, $153/3 = 51$ 역시 3의 배수이다.
- 그러니 3을 한 번 더 나누어서 $51/3 = 17$을 얻는다.
- 17은 $[4, 5, \cdots, 12]$에 속하는 약수를 갖지 않는다.
- 그러므로 17 역시 153의 소인수이다. 결론은 $153 = 3^2 \cdot 17$.

이제 이 알고리즘은 온전하게 $\mathcal{O}(\sqrt{n})$이다. 하지만 이 알고리즘은 간단한 최적화가 더 가능하고, 아직 정당성을 증명하지도 않았다. 이에 대한 부분은 연습문제로 넘긴다.

§1.3 Sieve of Eratosthenes

이어서 1부터 n까지의 모든 소수를 $\mathcal{O}(n \log \log n)$에 찾는 에라토스테네스의 체를 공부하자.

기본 아이디어는 2부터 n까지의 자연수를 모두 깔아놓고, 다음 과정을 거치는 것이다.

- 2를 제외한 2의 배수를 모두 제거한다
- 3을 제외한 3의 배수를 모두 제거한다
- 4를 제외한 4의 배수를 모두 제거한다
- 이를 반복하여, n을 제외한 n의 배수를 모두 제거한다.

먼저 이 알고리즘이 실제로 소수를 구할 수 있음을 증명해보자.

Proposition 1.3.1

각 $2 \le i \le n$에 대하여, i가 소수인 것은 i가 제거되지 않은 것과 동치이다.

Proof. i가 소수라면 i가 제거될 수 없음은 소수의 정의에서 확인할 수 있다.
만약 i가 소수가 아니라면 $i = ab$, $2 \le a, b < i$인 자연수 a, b가 존재한다.
이제 우리가 a를 제외한 a의 배수를 제거할 때 i가 제거됨을 확인할 수 있다. □

이제 이 알고리즘이 효율적인 알고리즘임을 증명해보자. 이를 위해서 필요한

Lemma 1.3.2 (Harmonic Series)

$$\sum_{i=1}^{n} \frac{1}{i} = \log n + \mathcal{O}(1)$$

는 잘 알려져 있고, 그 증명은 연습문제로 넘긴다. 이를 사용하면 다음을 얻는다.

Proposition 1.3.3

위 알고리즘의 시간복잡도는 $\mathcal{O}(n \log n)$이다.

Proof. n 이하인 i의 배수의 개수는 $\lfloor n/i \rfloor$이므로, i를 제외한 i의 배수를 제거하는 데 필요한 시간은 $\lfloor n/i \rfloor$에 비례한다. 그러므로, 위 알고리즘에 필요한 연산량은 대략 $\sum_{i=1}^{n} \lfloor n/i \rfloor$에 비례한다. 이 값이 $\mathcal{O}(n \log n)$임은 바로 위 lemma에서 나온다. □

하지만 아직 최적화가 가능하다. 자연수 $2 \le i \le n$이 소수가 아니라고 하자. 그러면

- i를 제외한 i의 배수를 모두 제거한다

는 과정은 의미가 없는데, p를 i의 소인수라고 하면 p를 제외한 p의 배수를 이전에 미리 지웠을 것이고, 이 과정에서 i를 포함한 i의 배수를 전부 제거했을 것이기 때문이다.

이를 종합하면 다음 알고리즘을 얻는다.

Algorithm 1.3.4 (Sieve of Eratosthenes)
2부터 n까지의 자연수를 모두 깔아놓고, 다음 과정을 거친다.

- 각 $i \in [2, \cdots, n]$에 대하여, i가 지워졌는지 확인한다.
- 만약 지워졌다면, 다음 자연수인 $i+1$로 넘어간다.
- 지워지지 않았다면, i를 제외한 i의 배수를 모두 제거한다.

위 논리를 잘 따라갔다면 각 $2 \leq i \leq n$에 대하여 우리가 i를 확인할 때 i가 지워지지 않은 것과 i가 소수임은 동치임을 알 수 있다. 이 알고리즘의 시간복잡도를 분석하기 위해,

Theorem 1.3.5 (Merten's 2nd Theorem)

$$\sum_{\substack{p \leq n \\ p \text{ is prime}}} \frac{1}{p} = \log\log n + \mathcal{O}(1)$$

라는 결과를 증명하지 않고 사용하자. 이제 다음 결과를 얻을 수 있다.

Proposition 1.3.6
위 알고리즘의 시간복잡도는 $\mathcal{O}(n \log\log n)$이다.

Proof. 기존 $\mathcal{O}(n \log n)$을 위한 증명과 다르지 않다. 다만, 계산해야 할 값은 이제

$$\sum_{\substack{p \leq n \\ p \text{ is prime}}} \lfloor n/p \rfloor$$

이고 이는 바로 앞의 정리에서 $\mathcal{O}(n \log\log n)$임을 알 수 있다. □

Proposition 1.2.1.을 생각하면, 사실 i의 범위를 $\sqrt{n}$ 이하까지만 봐도 소수를 찾기 위해서는 충분함을 알 수 있다. 이는 시간 최적화가 필요한 경우 약간의 도움을 줄 수 있다.

§1.4 Problems

Problem 1A. solved.ac 기준 Silver 이하 난이도의 정수론 문제를 필요한만큼 해결하라.

Problem 1B. Section 1.1에 있는 모든 Theorem의 증명을 공부하라.

Problem 1C. $a \equiv b \pmod{n}$이면, 다음이 성립함을 증명하여라.
(1) : 각 $c \in \mathbb{Z}$에 대하여 $a + c \equiv b + c \pmod{n}$
(2) : 각 $c \in \mathbb{Z}$에 대하여 $ac \equiv bc \pmod{n}$
(3) : 각 $c \in \mathbb{N}$에 대하여 $a^c \equiv b^c \pmod{n}$
(4) : 각 정수 계수 다항식 p에 대하여 $p(a) \equiv p(b) \pmod{n}$

Problem 1D. 정수 a, b, n에 대하여 다음을 증명하라.
(1) : $b|a$이고 $a \neq 0$이면 $|b| \leq |a|$이다.
(1) : $n|a$, $n|b$이면 $n|(a \pm b)$, $n|ab$.
(2) : $b|a$이면 $nb|na$이다.

Problem 1E. Algorithm 1.2.5에서 Algorithm 1.2.6으로 넘어가는 과정을 정당화하라.

Problem 1F. Algorithm 1.2.6에서 $A = \lfloor \sqrt{n} \rfloor$을 계산하는 것은 n의 값이 계산 과정에서 달라지기 때문이다. 하지만 실제로는 이러한 과정이 필요하지 않음을 보여라. 즉,

Algorithm 1.4.1
다음 과정을 거친다.

- 각 $i \in [2, 3, \cdots]$를 순서대로 본다. $i^2 > n$이면 iteration을 중단한다.
- i가 n의 **약수**인지 판별한다.
- 만약 그렇다면, n이 i의 배수가 아닐 때까지 $n \leftarrow n/i$를 반복한다.
- 이때 i는 기존 n의 소인수이며, 대응되는 지수 e는 i를 나누어준 횟수다.
- i에 대한 iteration이 끝난 뒤, $n \neq 1$이라면 이 역시 기존 n의 소인수다.

역시 올바른 소인수분해 알고리즘임을 증명하라. 이 알고리즘이 더 효율적임을 설명하라.

Problem 1G. OPTIONAL : Lemma 1.3.2를 증명하라.

Problem 1H. OPTIONAL : Theorem 1.3.5의 증명을 공부하여라.

Problem 1I. 이 Chapter에서 등장한 알고리즘을 모두 구현하라.

2 Euclidean Algorithms

이번 chapter에서 배우는 내용은 다음과 같다.

- 최대공약수의 여러 중요한 성질들
- 최대공약수를 $\log(\max(a,b))$ 시간에 구하는 유클리드 알고리즘
- 잉여역수를 $\log(\max(a,b))$ 시간에 구하는 확장 유클리드 알고리즘
- 확장 유클리드 알고리즘을 이용한 Linear Congruence의 빠른 해결

§2.1 Basic Properties of the GCD

Definition 2.1.1 (GCD/최대공약수)**.** $a=b=0$이 아닌 정수 a,b에 대하여, $\gcd(a,b)$는 $d|a$, $d|b$를 만족하는 최대의 자연수 d로 정의된다. $\gcd(0,0)$에 대한 convention은 자료마다 다를 수 있으나, 이 책에서는 $\gcd(0,0)=0$이라고 정의한다.

GCD의 정의에서 다음 사실을 보일 수 있다.

Proposition 2.1.2

(1) : 정수 a,b에 대하여 $\gcd(a,b)=\gcd(|a|,|b|)$.
(2) : 정수 a에 대하여 $\gcd(a,0)=|a|$.
(3) : 정수 a,b에 대하여 $\gcd(a,b)=\gcd(b,a)$.

다음 사실은 Euclidean Algorithm의 핵심이 되는 사실이다.

Proposition 2.1.3

(1) : 정수 a,b에 대하여 $\gcd(a,b)=\gcd(a\pm b,b)$.
(2) : 정수 a,b,n에 대하여 $\gcd(a,b)=\gcd(a+nb,b)$.
(3) : 정수 a와 자연수 b에 대하여 $\gcd(a,b)=\gcd(a \pmod b,b)$.

Proof. (1) : 자연수 d가 $d|a$, $d|b$를 만족하면 $d|(a+b)$, $d|b$ 역시 만족한다. 반대로 자연수 d가 $d|(a+b)$, $d|b$를 만족하면 $d|a$, $d|b$ 역시 성립한다. 그러므로, a,b의 공약수의 집합은 $a+b,b$의 공약수의 집합과 같고, 각 집합의 최대 원소 역시 같다. 즉 $\gcd(a,b)=\gcd(a+b,b)$이고, 마찬가지 방법으로 $\gcd(a,b)=\gcd(a-b,b)$.
(2) : 첫 번째 결과에서 수학적 귀납법을 적용하면 충분하다.
(3) : $a \pmod b=r$이라하면, $a=nb+r$인 정수 n이 존재한다.
그러므로, 두 번째 결과를 사용하면 증명이 끝난다. □

§2.2 Euclidean Algorithm

이제 본격적으로 정수 a, b에 대하여 $\gcd(a, b)$를 구하는 방법을 생각해보자.
Proposition 2.1.2의 첫 번째 결과에 의해 우리는 a, b를 $|a|, |b|$로 대체할 수 있고, 그러니 $a, b \geq 0$인 경우만 생각해도 충분하다. 또한, 2.1.2의 두 번째 결과에서 $a, b \geq 1$인 경우만 생각해도 충분하다. 마지막으로, 2.1.2의 세 번째 결과에서 일반성을 잃지 않고 $a \geq b$인 경우만 생각해도 좋다. 결론적으로, 우리는 $a \geq b \geq 1$에 대해서만 문제를 해결하면 된다.

나눗셈 정리를 이용하여, $a = bq + r$인 q, r을 계산하면, Proposition 2.1.3의 결과에 의해

$$\gcd(a, b) = \gcd(r, b) = \gcd(b, r)$$

이라는 결과를 얻는다.

$a \geq b > r$이므로, 이는 (a, b)에 대한 문제를 (b, r)에 대한 문제로 축소/환원시킨 것과 같다. 핵심은 이렇게 문제를 줄여나갈 때, 그 속도가 얼마나 빠르냐는 것이다. 그 답은

Proposition 2.2.1

자연수 a, b가 있고 $a \geq b$이다. a를 b로 나눈 나머지가 r일 때, $2r \leq a$.

Proof. $a = bq + r$이라 쓰면, $a \geq b$이므로 $q \geq 1$이다.
그러니 $2r \leq (q + 1)r = qr + r \leq qb + r = a$가 성립한다. □

에서 얻을 수 있다. 즉, $br \leq ab/2$가 성립한다.

이는 $\mathcal{O}(\log(ab))$ 번의 축소 과정을 거치면, 결국 $\gcd(g, 0)$ 형태의 문제에 도달한다는 것이다. 물론, $\gcd(g, 0) = |g|$이므로 이 경우 문제가 해결된다. 아래 예시를 보자.

Example 2.2.2

$\gcd(576, 204)$를 구한다고 가정하면, 다음 과정을 거친다.

- $576 = 204 \times 2 + 168$이므로, $\gcd(576, 204) = \gcd(204, 168)$
- $204 = 168 \times 1 + 36$이므로 $\gcd(204, 168) = \gcd(168, 36)$
- $168 = 36 \times 4 + 24$이므로 $\gcd(168, 36) = \gcd(36, 24)$
- $36 = 24 \times 1 + 12$이므로 $\gcd(36, 24) = \gcd(24, 12)$
- $24 = 12 \times 2 + 0$이므로 $\gcd(24, 12) = \gcd(12, 0) = 12$

Algorithm 2.2.3 (Euclidean Algorithm)

정수 a, b를 입력으로 받고 $\gcd(a, b)$를 출력한다. 시간복잡도는 $\mathcal{O}(\log(ab))$.

- 먼저 a, b를 음이 아닌 정수로 바꾸기 위해 $a \leftarrow |a|$, $b \leftarrow |b|$
- 만약 $a = 0$ 또는 $b = 0$이면, $a + b$를 반환한다.
- 그렇지 않다면, $a = bq + r$이라 하고 $a, b \leftarrow b, r$을 반복한다.
- 이때 $b = 0$인 순간 a를 반환하면 그 값이 GCD가 된다.

§2.3 Extended Euclidean Algorithm

이번 section에서는 다음 정리에 대해서 다룬다.

Theorem 2.3.1 (Bezout's Lemma)
임의의 정수 a, b에 대하여, 적당한 정수 x, y가 존재하여

$$ax + by = \gcd(a, b)$$

가 성립한다.

단순히 x, y의 존재성만 다루는 것을 넘어서, 조건을 만족하는 x, y를 빠른 시간 안에 계산하는 방법까지 공부해보자. 2.2의 내용과 비슷하게, 여기서도 $a \geq b \geq 1$인 경우만 문제를 다뤄도 충분하다. a, b의 부호나 순서를 바꾸는 것은 어렵지 않게 할 수 있기 때문이다.

r을 a를 b로 나눈 나머지라고 하고, $a = qb + r$이라 하자. 우리가 2.2에서 문제를 (a, b)에서 (b, r)로 축소한 것처럼, 여기서도 비슷한 전략을 생각해보자. 즉, $ax + by = \gcd(a, b)$인 x, y를 구하는 문제를 $bx' + ry' = \gcd(b, r)$인 x', y'을 구하는 문제로 환원시켜보자.

이를 위해서는, 우리가 $bx' + ry' = \gcd(b, r)$인 (x', y')를 알 때, 이를 $ax + by = \gcd(a, b)$인 (x, y)로 쉽게 변환시킬 수 있으면 된다. 그런데

$$\gcd(a, b) = \gcd(b, r) = bx' + ry' = bx' + (a - qb)y' = ay' + b(x' - qy')$$

이므로, 단순히 $x = y'$, $y = x' - qy'$을 선택하면 충분하다.

즉, Euclidean Algorithm의 문제 축소 과정을 그대로 적용할 수 있다. 이 과정을 통해서 x, y를 계산할 수 있음은 물론이고, Bezout's Lemma의 엄밀한 증명까지 할 수 있다. 이를 위해서 우리가 지금까지 얻은 사실을 정리해보자.

Proposition 2.3.2
S를 $ax + by = \gcd(a, b)$인 정수 x, y가 존재하는 정수 순서쌍 (a, b)의 집합이라고 하자.
즉, Bezout's Lemma의 결과는 S가 $\mathbb{Z}^2$과 같다는 것이다. 이때, 다음이 성립한다.
(1) : $(a, b) \in S \iff (|a|, |b|) \in S$
(2) : $(a, b) \in S \iff (b, a) \in S$
(3) : 임의의 정수 a에 대하여 $(a, 0) \in S$
(4) : $a \geq b \geq 1$이고 a를 b로 나눈 나머지가 r일 때, $(b, r) \in S \implies (a, b) \in S$.

이제 Theorem 2.3.1을 증명할 준비가 완료되었다.

Proof. Proposition 2.3.2에 의해, $a \geq b \geq 1$일 때만 $(a, b) \in S$임을 보이면 충분하다.
귀류법으로, $a \geq b \geq 1$이면서 $(a, b) \notin S$인 (a, b)가 존재한다고 가정하자.
이제 $(a, b) \notin S$, $a \geq b \geq 1$이면서 $a + b$가 최소인 순서쌍 (a, b)를 찾자.
a를 b로 나눈 나머지가 r이라고 하면, 다음 두 경우 중 하나에 도달한다.

- $(b, r) \in S$이면, Proposition 2.3.2의 4번 결과에 의해 $(a, b) \in S$다.
- $(b, r) \notin S$이면, $r \neq 0$이므로 $b \geq r \geq 1$이 성립하며, $b + r < a + b$이다.

그러니 첫 번째 경우는 $(a, b) \notin S$에 모순, 두 번째 경우는 $a + b$의 최소성에 모순. □

$ax + by = \gcd(a, b)$인 x, y를 구하는 것은 중요한 알고리즘이므로, 정리한다.

Algorithm 2.3.3 (Extended Euclidean Algorithm)

$a, b \geq 0$을 받아, $ax + by = \gcd(a, b)$인 정수 x, y를 반환. 시간은 $\mathcal{O}(\log(ab))$.

- $a = 0$이면 $(x, y) = (0, 1)$, $b = 0$이면 $(x, y) = (1, 0)$을 반환한다.
- $a = qb + r$이라 쓰고, (b, r)에 대해서 문제를 해결, $bx' + ry' = \gcd(b, r)$을 얻는다.
- 이제 $(x, y) = (y', x' - qy')$을 계산하고 이를 반환한다.

§2.4 Results From Bezout's Lemma

Bezout's Lemma는 정수론에서 기초적인 사실들을 증명하는데 사용되는 주요 도구 중 하나다. 증명은 연습문제로 맡기고, 여기서는 결과들만 소개한다. 모두 실제 문제풀이에 충분히 쓰일 수 있는 사실이고 이 책에서도 자주 사용되는 사실이므로, 숙지하는 게 좋다.

Proposition 2.4.1

(1) : $\gcd(a, b) = d$이면 $\gcd(a/d, b/d) = 1$이다.
(2) : $\gcd(a, b) = 1$이고 $a|c$, $b|c$이면 $ab|c$이다.
(3) : $a|bc$이고 $\gcd(a, b) = 1$이면 $a|c$이다.
(4) : $d|a$, $d|b$인 것은 $d|\gcd(a, b)$와 동치이다.
(5) : $\gcd(ab, ac) = |a| \gcd(b, c)$가 성립한다.
(6) : $a|bc$은 $\frac{a}{\gcd(a,b)}|c$와 동치이다.

Definition 2.4.2 (LCM/최소공배수)**.** 0이 아닌 두 정수 a, b의 최소공배수는 $a|m$, $b|m$인 자연수 m 중 최소인 것으로 정의된다. 이를 $\mathrm{lcm}(a, b)$로 표기한다.

Proposition 2.4.3

(1) : $\gcd(a, b) \cdot \mathrm{lcm}(a, b) = |ab|$가 성립한다.
(2) : $a|m$, $b|m$인 것은 $\mathrm{lcm}(a, b)|m$인 것과 동치이다.

특히, 앞서 Chapter 1에서 생략된 증명들, 예를 들면 소수 p가 있을 때 $p|ab$이면 $p|a$ 또는 $p|b$임 역시 Bezout's Lemma를 활용하여 증명한다. 이 사실은 다시 Fundamental Theorem of Arithmetic (Theorem 1.1.10) 등을 증명하는데 사용된다. 이 자료에서는 여기까지만.

§2.5 Solving Linear Congreunce

이번 section에서는 Linear Congruence에 대하여 다룬다. 즉,

$$ax \equiv b \pmod{n}$$

을 만족하는 x의 집합을 찾는 방법에 대해서 다룬다.

먼저 $ax \equiv b \pmod{n}$라는 조건을 $ax + ny = b$인 y가 존재한다는 것으로 보자.

결국 우리가 해야하는 것은

$$ax + ny = b$$

를 만족하는 (x, y)를 모두 구하는 것이다. 이를 위해서 다음을 증명할 수 있다.

Proposition 2.5.1

$ax + ny = b$인 (x, y)가 존재할 필요충분조건은 $\gcd(a, n)|b$이다.

Proof. 우선 $ax + ny = b$인 (x, y)가 있다면, 좌변은 $\gcd(a, n)$의 배수이므로 b 역시 $\gcd(a, n)$의 배수여야 한다. 반대로, $\gcd(a, n)|b$라면, Bezout's Lemma에서 $au + nv = \gcd(a, n)$인 정수 u, v를 찾을 수 있고,

$$a\left(\frac{b}{\gcd(a,n)}u\right) + n\left(\frac{b}{\gcd(a,n)}v\right) = b$$

□

우리는 이미 Bezout's Lemma의 실례를 찾는 Algorithm 2.3.3을 알고 있으므로,

$$ax + ny = b$$

가 해를 갖는지 판단하고, 그리고 해를 갖는다면 해를 하나 찾는 것은 빠르게 할 수 있다.

문제를 해를 **전부** 찾는 것이 된다. 이 문제를 해결하기 위해서, $ax + ny = b$인 (x, y)를 하나 찾았다 하고, 다른 해 (x', y')을 모두 찾아보자. 이제 $ax + ny = ax' + ny' = b$가 되고, $a(x - x') = n(y' - y)$를 얻는다. 이를 만족하는 y'이 존재할 필요충분조건은 $n|a(x - x')$이다.

이는 Proposition 2.4.1의 6번에서

$$\frac{n}{\gcd(a,n)}|(x - x')$$

과 동치이다. 그러므로 우리는

$$x' \equiv x \pmod{n/\gcd(a, n)}$$

이 전부 해가 될 수 있고, 이것이 해 전부임을 얻는다. 이를 정리하면

Proposition 2.5.2

$d = \gcd(a, n)$이고 $d|b$일 때, $ax + ny = d$의 한 해를 (x_0, y_0)라 하자.
이때, $ax + ny = d$의 해는 정수 t에 대하여 다음과 같이 표현할 수 있다.

$$x = x_0 + \frac{n}{d}t, \quad y = y_0 - \frac{a}{d}t$$

특히, 이는 $x \equiv x_0 \pmod{n/d}$임을 의미한다.

이를 정리하여 알고리즘으로 만들 수 있다.

Algorithm 2.5.3 (Solving Linear Congruences)
정수 a, b, n이 주어졌을 때, $ax \equiv b \pmod{n}$을 푼다. 시간은 $\mathcal{O}(\log an)$.

- $d = \gcd(a, n)$을 Euclidean Algorithm으로 계산한다. $d|b$가 아니면 해가 없다.
- $ax_0 + ny_0 = b$인 (x_0, y_0)를 Extended Euclidean Algorithm으로 찾는다.
- 답은 $x \equiv x_0 \pmod{n/d}$이다. 이를 결과로 반환한다.

특히, $\gcd(a, n) = 1$이면, $ax \equiv 1 \pmod{n}$인 x를 $\pmod{n}$에서 유일하게 찾을 수 있다. 이를 $\pmod{n}$에 대한 a의 **잉여역수**라고 부르며, 경우에 따라서 a^{-1}으로 표기하기도 한다. 잉여역수는 영어로 modular inverse라고 부르며, 앞으로 이 책에서도 이렇게 부를 것이다. 이름의 뜻 그대로, 잉여역수를 곱하는 것은 결국 나눗셈을 하는 것과 같다.

§2.6 Problems

Problem 2A. https://www.acmicpc.net/workbook/view/6594 의 문제를 해결하라

Problem 2B. Proposition 2.4.1, 2.4.3을 증명하라.

Problem 2C. 이 Chapter에서 다룬 알고리즘을 모두 구현하라.

Problem 2D. 다음을 모두 증명하라.
(1) : $\gcd(a, b) = \gcd(a, c) = 1$이면 $\gcd(a, bc) = 1$이다.
(2) : $\gcd(a, b) = 1$임은 $p|a$, $p|b$인 소수 p가 존재하지 않음과 동치이다.
(3) : $\gcd(a, b) = 1$이면 임의의 자연수 k에 대하여 $\gcd(a^k, b^k) = 1$.
(4) : $\gcd(a^m - 1, a^n - 1) = a^{\gcd(m,n)} - 1$.

Problem 2E. Algorithm 2.3.3에 자연수 $a, b \geq 1$을 입력으로 줬을 때, 그 결과가 (x, y)라 하자. 이때, $|x| \leq b$, $|y| \leq a$가 성립함을 증명하라. (힌트 : 귀납법)
이는 Algorithm 2.3.3을 구현할 때 overflow에 대한 걱정을 할 필요가 없다는 의미다.

Problem 2F. OPTIONAL : Principal Ideal Domain, Euclidean Domain에 대하여 공부하라. (이인석 교수님의 학부 대수학 강의 II, 5장 참고)

3 Chinese Remainder Theorem

이번 Chapter에서 다루는 내용은 다음과 같다.

- 연립합동식을 로그 시간에 해결하는 알고리즘
- 중국인의 나머지 정리의 의미와 그 활용법

§3.1 Chinese Remainder Theorem

Theorem 3.1.1 (Chinese Remainder Theorem)
자연수 $n_1, n_2, \cdots, n_k > 1$은 쌍마다 서로소이다. 즉, 각 $1 \le i < j \le k$에 대하여 $\gcd(n_i, n_j) = 1$이다. 이때, 각 $0 \le a_i < n_i$에 대하여 연립합동식

$$x \equiv a_i \pmod{n_i} \quad 1 \le i \le k$$

는 $\pmod{N}$에서 유일한 해를 갖는다. 물론, $N = n_1 n_2 \cdots n_k$.

이 정리의 의미에 대해서는 다음 section에 대해서 다루고, 여기서는 그 해를 찾는 방법에 대해서 생각해보자. 본격적인 설명에 앞서, 몇 가지 기본적인 전처리를 하고 시작하자.

첫째 : 먼저 $\gcd(n_i, n_j) = 1$이라는 쌍마다 서로소 조건을 **제거한다**.

둘째 : 2개의 합동식으로 이루어진 연립합동식을 해결할 수 있다면, k개의 합동식으로 이루어진 연립합동식도 해결할 수 있다. k개의 합동식이 있다고 하자. 첫 2개의 합동식으로 이루어진 연립합동식을 풀어 새로운 합동식을 얻으면, k개의 합동식은 $k-1$개의 합동식이 된다. 이렇게 합동식의 개수를 하나씩 줄여나가면, 결국 합동식의 개수가 2개가 되고 이는 충분히 해결할 수가 있을 것이다. 이는 우리가 2개의 합동식으로 이루어진 연립합동식만 해결할 수 있으면 모든 문제가 해결된다는 것이다. 그러니 이를 해결해보자.

Theorem 3.1.2 (System of Two Congruences)
자연수 $0 \le a_1 < n_1$과 $0 \le a_2 < n_2$가 있다. 연립합동식

$$x \equiv a_1 \pmod{n_1}, \quad x \equiv a_2 \pmod{n_2}$$

가 해를 가질 필요충분조건은 $a_1 \equiv a_2 \pmod{\gcd(n_1, n_2)}$이다.
또한, 해는 $\pmod{\operatorname{lcm}(n_1, n_2)}$에서 유일하고, $\mathcal{O}(\log(\max(n_1, n_2)))$ 시간에 계산된다.

증명을 하면서 알고리즘까지 설명하도록 하겠다. 위 연립합동식이 해 x_0를 갖는다는 것은

$$x_0 = n_1 y_1 + a_1 = n_2 y_2 + a_2$$

이 해를 갖는다는 것이며, 이는 $n_2 y_2$ 입장에서 생각해보면

$$n_2 y_2 \equiv (a_1 - a_2) \pmod{n_1}$$

이 된다. 이를 y_2에 대한 합동식이라고 생각하면, 이러한 y_2가 존재할 필요충분조건은

$$a_1 - a_2 \equiv 0 \pmod{\gcd(n_1, n_2)}$$

임을 Proposition 2.5.1에서 확인할 수 있다. 이렇게 y_2를 찾으면

$$x_0 = n_2 y_2 + a_2, \quad y_1 = \frac{n_2 y_2 + a_2 - a_1}{n_1}$$

으로 잡으면 앞선 식이 성립함을 확인할 수 있다. 이렇게 x_0를 찾을 수 있다.

마지막으로, 해가 $\pmod{\text{lcm}(n_1, n_2)}$에 대하여 유일함을 보이자. 만약 x가 위 합동식의 해라면, x에 $\text{lcm}(n_1, n_2)$의 배수를 더해도 여전히 해임을 확인할 수 있다. 문제는 연립합동식의 두 해 x, y가 존재하면 $x \equiv y \pmod{\text{lcm}(n_1, n_2)}$인지 확인하는 것이다.

$$x \equiv a_1 \equiv y \pmod{n_1}, \quad x \equiv a_2 \equiv y \pmod{n_2}$$

이므로, $x - y$가 n_1, n_2의 배수가 된다.

그러니 $x - y$는 $\text{lcm}(n_1, n_2)$의 배수임을 알 수 있어, 증명이 끝난다. 결론적으로, 앞서 설명한 알고리즘으로 x_0를 찾는데 설명했다면, $x \equiv x_0 \pmod{\text{lcm}(n_1, n_2)}$이 그 해다.

Algorithm 3.1.3 (Solving Two Congruences)
$x \equiv a_1 \pmod{n_1}$, $x \equiv a_2 \pmod{n_2}$를 해결하는 알고리즘이다.

- $g = \gcd(n_1, n_2)$와 $l = \text{lcm}(n_1, n_2)$를 유클리드 알고리즘으로 계산한다.
- $a_1 - a_2$가 g의 배수가 아니라면, 해가 없다.
- $n_2 y_2 \equiv (a_1 - a_2) \pmod{n_1}$인 y_2를 찾는다.
- $x_0 = n_2 y_2 + a_2$를 계산한다.
- 연립합동식의 해는 $x \equiv x_0 \pmod{l}$이다.

§3.2 CRT-style Thinking

자연수 n과 그 소인수분해 $n = p_1^{e_1} \cdots p_k^{e_k}$가 있다고 하자. 중국인의 나머지 정리가 의미하는 바는, $x \pmod{n}$을 알기 위해서는 **prime power에 대한 결과**

$$x \pmod{p_1^{e_1}}, \quad x \pmod{p_2^{e_2}}, \quad \cdots \quad x \pmod{p_k^{e_k}}$$

를 알면 충분하다는 것이고, 이들 사이에는 일대일대응이 존재한다는 것이다.

이는 결국 n의 소인수분해를 알고 있다면, n에 대한 문제를 **prime power** $p_1^{e_1}$, $p_2^{e_2}$, $\cdots$, $p_k^{e_k}$에 대한 문제로 쪼갤 수 있음을 시사한다. 몇 가지 예를 들어보자.

Euler-Phi Function : $\phi(n)$은 n 이하이고 n과 서로소인 자연수의 개수이다. $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$라 할 때, $1 \le x \le n$이 n과 서로소일 필요충분조건은 x가 $p_1, p_2, \cdots, p_k$ 중 어느 하나의 배수도 아닌 것이다. 이는 결국 각 $1 \le i \le k$에 대하여, $x \pmod{p_i^{e_i}}$가 p_i의 배수가 아닌 것과 동치이다. 그런데 중국인의 나머지 정리에 의해,

$$x \pmod{p_i^{e_i}}$$

를 각 $1 \le i \le k$에 대하여 결정한다면 $x \pmod n$이 자동으로 유일하게 결정되므로, 필요한 x의 개수를 구하려면 가능한 $x \pmod{p_i^{e_i}}$의 개수를 각각 계산한 후 이를 곱하면 된다. 즉, $x \pmod{p_i^{e_i}}$ **각각을 독립적으로 생각해도 무방하다는 것이다**. 물론, $x \pmod{p_i^{e_i}}$ 중 p_i의 배수가 아닌 것은 단순히 $p_i^{e_i} - p_i^{e_i-1}$이므로, 여기서 우리는 결론인

$$\phi(n) = \prod_{i=1}^{k} \left(p_i^{e_i} - p_i^{e_i-1} \right) = n \cdot \prod_{i=1}^{k} \left(1 - \frac{1}{p_i} \right)$$

를 얻는다. 각 prime power를 독립적으로 생각할 수 있다는 것이 핵심이다.

Polynomial Roots : f가 다항식일 때, $f(x) \equiv 0 \pmod n$을 풀고 싶다고 하자. 이때,

$$f(x) \equiv 0 \pmod{p_i^{e_i}}$$

의 해를 각 $1 \le i \le k$에 대해 구한 다음, 중국인의 나머지 정리로 이를 합쳐서 해결할 수 있다. 결론은 "소인수분해를 안다는 가정하에, n에 대한 문제를 prime power에 대한 문제로 변환할 수 있다". 이것이 유의미한 결과인 이유는 물론, 일반적인 n에 대해서 직접 풀기 어려운 문제가 소수와 prime power에 대해서는 쉬운 경우가 있기 때문이다.

Proving Identities : 위 예시들과는 약간 다르다. 설명을 위해,

$$\sum_{d|n} \phi(d) = n$$

이 모든 자연수 n에 대해서 성립함을 증명해보자.

먼저, $n = p^e$가 prime power일 때 증명한다. 이때는 단순한 계산으로,

$$\sum_{d|p^e} \phi(d) = \sum_{i=0}^{e} \phi(p^i) = 1 + \sum_{i=1}^{e} \phi(p^i) = 1 + \sum_{i=1}^{e} (p^i - p^{i-1}) = p^e$$

를 얻는다. 이제 일반적인 $n = p_1^{e_1} \cdots p_k^{e_k}$에 대해서 생각해보자. n의 각 약수는

$$p_1^{f_1} \cdots p_k^{f_k}, \quad 0 \le f_i \le e_i$$

형태를 갖고 있으므로, 계산을 해보면

$$\begin{aligned}
\sum_{d|n} \phi(d) &= \sum_{0 \le f_i \le e_i, 1 \le i \le k} \phi(p_1^{f_1} \cdots p_k^{f_k}) \\
&= \sum_{0 \le f_i \le e_i, 1 \le i \le k} \phi(p_1^{f_1}) \phi(p_2^{f_2}) \cdots \phi(p_k^{f_k}) \\
&= \left(\sum_{0 \le f_1 \le e_1} \phi(p_1^{f_1}) \right) \cdots \left(\sum_{0 \le f_k \le e_k} \phi(p_k^{f_k}) \right) \\
&= p_1^{e_1} \cdots p_k^{e_k} = n
\end{aligned}$$

을 얻는다. 여기서 첫째 줄에서 둘째 줄로 넘어가는 것에서 ϕ의 multiplicative함이 사용되었고, 마지막 계산에서 $n = p^e$에 대한 증명이 사용되었다. 즉, multiplicative한 함수에 대한 계산/증명을 위해서 $n = p^e$에 대한 증명을 먼저하고, 이를 갖고 일반적인 경우를 증명하는 것은 일반적이다. multiplicative 성질과 CRT 사고방식의 관계는 잠깐 생각해보자.

§3.3 Problems

Problem 3A. https://www.acmicpc.net/workbook/view/6595

Problem 3B. https://codeforces.com/problemset?tags=chinese+remainder+theorem

Problem 3C. https://www.acmicpc.net/problem/17257

Problem 3D. https://projecteuler.net/archives 의 chinese remainder theorem 태그.

Problem 3E. OPTIONAL : Garner's Algorithm을 공부하라.
좋은 자료는 https://cp-algorithms.com/algebra/chinese-remainder-theorem.html

Problem 3F. OPTIONAL : 일반적인 ring에 대한 중국인의 나머지 정리를 공부하라.

II

Some Common Applications

4 More on Sieves

이 Chapter에서는 "체"를 사용하는 알고리즘의 다양한 응용과 확장을 살펴본다.

§4.1 Calculation of Various Functions

체를 활용하는 가장 기본적인 방법은 에라토스테네스의 체 알고리즘을 돌리는 과정에서 여러 유용한 함수를 계산하는 것이다. 여기서는 몇 가지 중요한 예시를 살펴본다.

최소 소인수 : Algorithm 1.3.4에서 정수 m이 "i를 제외한 i의 배수를 모두 제거하는" 단계에서 제거되었다면, m의 최소 소인수는 i가 된다. 그러니 이를 기록해두면 각 합성수에 대하여 최소 소인수를 계산할 수 있으며, 각 소수의 최소 소인수는 물론 자기 자신이다.

에라토스테네스의 체 알고리즘을 돌리면서 계산된 최소 소인수 배열을 활용하면 n 이하 임의의 자연수에 대한 소인수분해를 $\mathcal{O}(\log n)$ 시간에 (per query) 할 수 있다.

Algorithm 4.1.1 (Log-Time Factorization after Sieve)
각 $2 \le i \le n$에 대하여 i의 최소 소인수 lpf_i를 알고 있다고 가정하자. 이때, 다음 알고리즘으로 임의의 $2 \le k \le n$을 $\mathcal{O}(\log k)$ 시간에 소인수분해 할 수 있다.

- $k = 1$이라면 그대로 종료한다.
- $k > 1$이라면 $p = lpf[k]$라 하자.
- $p \nmid k$일 때까지 k에서 p를 나눠주고, 다시 처음부터 반복한다.

Proof. 위 알고리즘이 $\mathcal{O}(\log k)$ 시간에 작동함을 증명하자. k에서 소인수 p가 나누어질 때마다 k는 적어도 절반 이상 감소하므로, 그 횟수는 $\mathcal{O}(\log k)$이다. □

위 알고리즘의 시간복잡도에 대한 논의를 조금 더 깊게 해보자. 다음 두 함수를 정의하자.

Definition 4.1.2 (Prime Omega Function)**.** n의 소인수분해가 $n = p_1^{e_1} \cdots p_k^{e_k}$라 할 때,

$$\omega(n) = k$$
$$\Omega(n) = \sum_{i=1}^{k} e_i$$

를 prime omega function이라고 부른다.
즉, $\omega(n)$은 n의 서로 다른 소인수 개수, $\Omega(n)$은 n의 중복을 고려한 소인수 개수다.

이제 앞선 알고리즘을 돌아보면, 그 시간복잡도가 $\mathcal{O}(\Omega(k))$과 같음을 알 수 있다.
앞서 우리는 $\Omega(k) \le \log_2 k$임을 확인하였다. 이제 $\Omega(k)$의 합에 대한 bound를 보자.

Theorem 4.1.3 (Average Order of Prime Omega Functions)

$$\sum_{i=1}^{n} \omega(i) = n \log \log n + B_1 n + o(n)$$
$$\sum_{i=1}^{n} \Omega(i) = n \log \log n + B_2 n + o(n)$$

이 성립한다. 단, $B_1 \approx 0.261497$, $B_2 \approx 1.034506$이다.

즉, 위 알고리즘으로 1부터 n을 소인수분해 하는 것은 $\mathcal{O}(n \log \log n)$ 시간이 걸린다.

서로 다른 소인수의 개수 : Algorithm 1.3.4에서 "i를 제외한 i의 배수를 모두 제거하는" 단계에서는 소수 i의 배수를 모두 순회하게 된다. 이때, 순회되는 정수들이 i의 배수인 것을 알고 있으니, 각각의 서로 다른 소인수 개수를 1씩 증가시켜주면 이 값도 전부 계산된다.

오일러 파이 함수 : 뒤에서 다루게 될 대상 중 하나인 오일러 파이 함수는

$$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

로 정의된다. 단, $p_1, p_2, \cdots, p_k$는 n의 서로 다른 소인수. 이를 계산하기 위해서는 $\phi(n)$의 값을 n으로 초기화한 뒤, n의 각 소인수 p_i에 대하여 $\phi(n) \leftarrow \phi(n) \cdot (1 - 1/p_i)$를 적용시켜주면 된다. 이때, n에 대해서 각 소인수를 나열하는 것은 어렵지만, 반대로 각 소수 p에 대하여 그 배수를 나열하는 것은 쉽다는 점을 활용한다. 이를 정리하면 다음을 얻는다.

Algorithm 4.1.4 (Calculating Euler-Phi Function)

아래 알고리즘으로 $1 \le i \le n$에 대하여 $\phi(i)$의 값을 $\mathcal{O}(n \log \log n)$에 계산할 수 있다.

- 먼저 각 $1 \le i \le n$에 대해 ϕ_i를 i로 초기화한다.
- 에라토스테네스의 체 알고리즘을 돌린다. 현재 소수 p를 찾았다고 하자.
- 이때 $\phi_p = p - 1$로 두고, n 이하의 p의 배수 j를 순회하면서 $\phi_j \leftarrow \phi_j - \phi_j / p$
- 이를 각 소수 p에 대하여 반복하면 최종적으로 $\phi_i = \phi(i)$를 얻는다.

뫼비우스 함수 : 역시 뒤에서 다루게 될 대상 중 하나인 뫼비우스 함수는 $n = \prod_{i=1}^{k} p_i^{e_i}$가 n의 소인수분해일 때 다음과 같이 정의된다.

$$\mu(n) = \begin{cases} (-1)^k & e_1 = e_2 = \cdots = e_k = 1 \\ 0 & \text{otherwise} \end{cases}$$

특히 $\mu(1) = 1$이다. 예를 들면, $\mu(4) = \mu(18) = 0$이고, $\mu(15) = 1$, $\mu(30) = -1$이다.

Algorithm 4.1.5 (Calculating Mobius Function)
아래 알고리즘으로 $1 \le i \le n$에 대하여 $\mu(i)$의 값을 $\mathcal{O}(n \log\log n)$에 계산할 수 있다.

- 먼저 각 $1 \le i \le n$에 대해 μ_i를 1로 초기화한다.
- 에라토스테네스의 체 알고리즘을 돌린다. 현재 소수 p를 찾았다고 하자.
- 이때 $\mu_p = -1$로 두고, n 이하의 p의 배수 j를 순회한다.
- $p^2 | j$라면 $\mu_j \leftarrow 0$을 하고, 아니면 $\mu_j \leftarrow -\mu_j$를 한다.
- 이를 각 소수 p에 대하여 반복하면 최종적으로 $\mu_i = \mu(i)$를 얻는다.

Multiplicative Functions : 독자들은 오일러 파이 함수와 뫼비우스 함수의 계산 방식이 상당히 비슷함을 느낄 수 있었을 것이다. 이 비슷한 점이 어디에서 나오는 것인지를 알아보고, 이를 바탕으로 위 알고리즘을 확장해보자. $n = p_1^{e_1} \cdots p_k^{e_k}$가 n의 소인수분해면,

$$\frac{\phi(n)}{n} = \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$
$$\mu(n) = \prod_{i=1}^{k} [-1 \text{ if } e_k = 1,\ 0 \text{ if } e_k \ge 2]$$

가 성립하는데, 우변을 보면 n의 각 prime power들이 함수의 값에 독립적으로 기여하고 있음을 알 수 있다. 이러한 형태의 함수를 multiplicative function이라고 부른다.

Definition 4.1.6 (multiplicative function)**.** 함수 $f : \mathbb{N} \to \mathbb{C}$가 각 $n = p_1^{e_1} \cdots p_k^{e_k}$에 대해

$$f(n) = f(p_1^{e_1}) f(p_2^{e_2}) \cdots f(p_k^{e_k})$$

를 만족하면, f를 multiplicative function이라고 부른다.

즉, prime power에 대한 f의 값을 결정하면 f 자체가 결정되는 구조를 갖고 있다.

이제 다시 앞선 예시들을 돌아보자. $f(n) = \phi(n)/n$의 경우,

$$\frac{\phi(n)}{n} = \prod_{i=1}^{k} \frac{\phi(p_i^{e_i})}{p_i^{e_i}} = \prod_{i=1}^{k} \frac{p_i - 1}{p_i}$$

이며, 비슷하게 $f(n) = \mu(n)$의 경우 $\mu(p^e)$의 값을 생각해보면 같은 결론을 얻는다.

이렇게 보면 우리가 앞서 다룬 예시들이 특수한 경우임을 알 수 있는데, $f(n) = \phi(n)/n$의 경우 n의 각 prime power p^e에 대하여 e의 값이 함수의 값에 영향을 주지 않기 때문이다. 즉, $f(p^e) = f(p) = 1 - 1/p$가 성립하므로 계산 과정에서 e의 값에 신경을 쓸 필요가 없었다. 실제로 앞선 알고리즘에서 우리는 e에 대한 고려를 한 적이 없다.

$f(n) = \mu(n)$의 경우, 각 prime power p^e에 대하여 $e = 1$인가 아니면 $e \ge 2$인가만이 중요했다. 그래서 앞선 알고리즘에서는 에라토스테네스의 체를 적용하는 과정에서 p^2이 현재 보는 자연수의 약수인지만 판별했던 것이다. 이제 일반적인 알고리즘을 생각해보자.

Algorithm 4.1.7 (Calculating Multiplicative Functions)

f가 multiplicative function이라 하자. 아래 알고리즘으로 $1 \le i \le n$에 대하여 $f(i)$의 값을 계산할 수 있다.

- 먼저 각 $1 \le i \le n$에 대해 f_i를 i로 초기화한다.
- 에라토스테네스의 체 알고리즘을 돌린다. 현재 소수 p를 찾았다고 하자.
- 이때 $f_p = f(p)$로 두고, n 이하의 p의 배수 j를 순회한다.
- 각 j에 대하여, j가 갖는 p의 개수 e를 구하고, $f_j \leftarrow f_j \cdot f(p^e)$를 계산한다.
- 이를 각 소수 p에 대하여 반복하면 최종적으로 $f_i = f(i)$를 얻는다.

시간복잡도 분석을 위해서는 가정이 필요하다. $f(p^e)$라는 값이 효율적으로 계산 가능해야 한다는 것인데, 이는 웬만하면 성립하는 가정이다. 여기서 "효율적으로 계산 가능하다"는 것은, 적어도 $\mathcal{O}(e)$ 선에서 계산이 가능하다는 것이다.

이제 시간복잡도 분석을 해보자. 에라토스테네스 체의 기본 시간복잡도 $\mathcal{O}(n \log \log n)$은 기본으로 들어간다. $1 \le m \le n$에 대하여 $f(m)$을 계산하기 위한 시간복잡도를 생각해보자. $m = p_1^{e_1} \cdots p_k^{e_k}$라 하자. $f(m)$에 대한 $p_i^{e_i}$의 기여를 계산하기 위해서, 우선 e_i의 값을 계산해야 하고 이는 $\mathcal{O}(e_i)$ 시간이 걸린다. 그 후, $f(p_i^{e_i})$를 계산해야 하는데 이는 $\mathcal{O}(e_i)$ 이하의 시간이 걸릴 것이라고 가정을 했다. 결국 $p_i^{e_i}$의 기여는 $\mathcal{O}(e_i)$에 계산이 된다. 이를 모두 합치면, $f(m)$을 계산하는 것에는 총 $\mathcal{O}(\Omega(m))$의 시간이 걸린다는 것이고, 그 총합은 정리 4.1.3.에서 $\mathcal{O}(n \log \log n)$이다. 그러므로 총 시간복잡도는 $\mathcal{O}(n \log \log n)$임을 알 수 있다.

이는 매우 일반적이고 강력하며 구현하기도 어렵지 않은 알고리즘이다.

§4.2 Factorization of Batches

Algorithm 4.1.1은 $\mathcal{O}(n \log \log n)$ 시간에 $1 \le i \le n$을 모두 소인수분해하는 알고리즘을 제시한다. 이 section에서는 Algorithm 4.1.1을 확장하여, 특정 형태를 갖고 있는 자연수들을 전부 소인수분해하는 알고리즘을 설명하고자 한다. 즉, 우리의 목표는

문제 : f는 d차 정수 계수 다항식이다. $f(1), f(2), f(3), \cdots, f(n)$을 전부 소인수분해하라.
를 해결하는 것이다. Algorithm 4.1.1은 $f(n) = n$인 경우라고 해석할 수 있다.

문제를 풀기 전에, Algorithm 4.1.1을 다음과 같은 형태로 바꾸어보자.

Algorithm 4.2.1 (Batch Factorization : Easy Case)
다음 알고리즘으로 각 $1 \le i \le n$의 소인수분해를 $\mathcal{O}(n \log \log n)$에 계산할 수 있다.

- $1 \le i \le n$에 대하여 $v_i = i$라 하고, multiset $fac_i = \emptyset$을 준비한다.
- 에라토스테네스의 체를 돌려서, $1 \le i \le \sqrt{n}$까지의 소인수를 전부 계산한다.
- 각 소수 $2 \le p \le \sqrt{n}$와 각 n 이하 p의 배수 i에 대해 다음을 시행한다.
- $p | v_i$가 성립하는 동안 v_i를 v_i/p로 바꾸고 fac_i에 p를 추가하는 것을 반복한다.
- 마지막으로, 각 $1 \le i \le n$에 대하여 $v_i \neq 1$이면 fac_i에 v_i를 추가한다.
- 이제 각 $1 \le i \le n$에 대하여 fac_i는 i의 소인수분해가 된다.

즉, $m = p_1^{e_1} \cdots p_k^{e_k}$면 fac_m에는 각 $1 \le i \le k$에 대하여 p_i가 e_i개 존재하게 된다.

즉, 먼저 에라토스테네스의 체로 $\sqrt{n}$ 이하인 소수를 전부 구한다. 그 후, 각 $\sqrt{n}$ 이하의 소수 p에 대하여, p의 배수들에서 p를 전부 나누어주는 작업을 거친다. 그런데 각 $1 \le i \le n$은 $\sqrt{n}$ 초과의 소인수를 많아야 하나 가질 수 있다. 그러므로 $\sqrt{n}$ 이하의 소인수를 전부 나누어준 다음 남는 값이 1이 아니라면, 이는 $\sqrt{n}$ 초과의 소인수가 된다. 이는 우리가 앞서 Algorithm 1.2.5와 1.2.6에서 활용한 사실로, 이 소인수까지 고려하면 소인수분해가 완료된다.

이제 위 알고리즘을 생각하면서, 목표로 하는 문제를 해결해보자.

Step 1 : 기본적인 계산부터 하자. $f(1), f(2), \cdots, f(n)$을 전부 계산하고,

$$B = \max(|f(1)|, |f(2)|, \cdots, |f(n)|)$$

을 계산한다. 이는 $\mathcal{O}(nd)$에 가능하며, f에 따라서 더 빠르게 계산할 수도 있다.

Step 2 : 이제 정수론적인 부분을 고민해보자. Algorithm 4.2.1을 그대로 복사하면,

- $1 \le i \le n$에 대하여 $v_i = f(i)$라 하고, multiset $fac_i = \emptyset$을 준비한다.
- 에라토스테네스의 체를 돌려서, $1 \le i \le \sqrt{B}$까지의 소인수를 전부 계산한다.
- 각 소수 $2 \le p \le \sqrt{B}$와 $p | f(i)$인 각 $i \le n$에 대해 다음을 반복한다.
- $p | v_i$가 성립하는 동안 v_i를 v_i/p로 바꾸고 fac_i에 p를 추가하는 것을 반복한다.
- 마지막으로, 각 $1 \le i \le n$에 대하여 $v_i \neq 1$이면 fac_i에 v_i를 추가한다.

- 이제 각 $1 \le i \le n$에 대하여 fac_i는 $f(i)$의 소인수분해가 된다.

를 순서대로 하면 문제를 해결할 수 있음을 알 수 있다. 여기서 Algorithm 4.2.1과 명확하게 난이도 차이가 발생하는 부분은 $p|f(i)$인 $i \le n$을 효율적으로 iterate 하는 것이다. 기존에 $f(m) = m$인 경우에서는 $p|f(i)$는 i가 p의 배수인 것과 같고, p의 배수를 iterate 하는 것은 어렵지 않았다. 이제는 다르다. 우리는 $f(i) \equiv 0 \pmod p$라는 합동식을 해결해야 한다.

이 난이도는 f의 차수 d에 따라 달라진다.

- $d = 1$인 경우 우리는 $ax + b \equiv 0 \pmod p$를 풀어야 한다. 이는 Chapter 2에서 다룬다.
- $d = 2$인 경우 우리는 $ax^2 + bx + c \equiv 0 \pmod p$를 푼다. 이는 Chapter 9에서 다룬다.
- $d \ge 3$인 경우, 가장 일반적인 경우는 마지막 Chapter에서 다룬다.

결과적으로 $f(i) \equiv 0 \pmod p$를 푸는데 걸리는 시간을 $T(p, d)$라고 하자. 여러 알고리즘이 있지만, 이 책에서 우리가 배울 알고리즘들은 대략

$$T(p, d) = \mathcal{O}(d^2 \log p)$$

정도를 만족한다. $f(x)$가 $ax^d + b$ 등 특수한 형태라면, 더욱 효율적인 알고리즘이 있다. 이에 대한 논의는 추후에 하는 것으로 하자. 어쨌든 $f(i) \equiv 0 \pmod p$를 풀면 그 결과는 $i \equiv i_1, i_2, \cdots, i_t \pmod p$ 형태로 나올 것이며, 이를 순회하는 것은 어렵지 않다.

Step 3 : 시간복잡도를 분석하자. 에라토스테네스 체는 $\mathcal{O}(\sqrt{B} \log \log B)$. $\sqrt{B}$ 이하 소인수를 제거하기 위해 $f(i) \equiv 0 \pmod p$를 푸는 시간복잡도는 $\sum_{p \le \sqrt{B}} T(p, d)$인데,

Theorem 4.2.2 (Sum of Logarithms of Primes)

$$\sum_{p \le n} \log p = \mathcal{O}(n)$$

라는 정리를 사용하면

$$\sum_{p \le \sqrt{B}} T(p, d) = \mathcal{O}(d^2 \sqrt{B})$$

정도를 얻는다. 마지막으로 $\sqrt{B}$ 이하의 소수를 실제로 모두 나누는 것은

$$\sum_{i=1}^{n} \Omega(|f(i)|) \le \sum_{i=1}^{n} \log_2 |f(i)| \le n \log_2 B$$

에 가능하다. $f(i)$들을 전부 계산하고 $\sqrt{B}$ 초과 소인수를 다루는 것은 $\mathcal{O}(nd)$.

그러니 총 시간복잡도는 약 $\mathcal{O}(nd + n \log B + \sqrt{B}(d^2 + \log \log B))$ 정도가 된다.

이후에 우리는 $\tilde{\mathcal{O}}(n^{1/4})$ 시간에 소인수분해를 하는 Pollard-Rho 알고리즘을 배울 것이다. 이 알고리즘을 사용하면, 대략 $\tilde{\mathcal{O}}(nB^{1/4})$ 시간에 목표를 달성할 수 있다. 이를 우리가 체를 이용하여 얻은 알고리즘과 비교하여, 파라미터의 크기에 알맞는 알고리즘을 선택하면 된다.

§4.3 Sieving in Linear Time

이번 section에서는 체 알고리즘에 붙은 $\log\log n$ factor를 제거하는 작업을 한다. 대부분의 정수론 문제에서는 $\mathcal{O}(n\log\log n)$ 알고리즘으로 충분하지만, 극한의 시간 최적화를 요구하는 문제가 등장할 경우 이번 section의 알고리즘이 도움이 될 수 있다.

에라토스테네스의 체의 문제는 한 자연수를 여러 번 확인한다는 것이다. 예를 들어, 6이 합성수라는 사실을 2의 배수라는 점에서도, 3의 배수라는 점에서도 모두 확인한다. 이를 해소하기 위해서, 전략을 다시 세운다. 자연수 n이 있고, n의 가장 작은 소인수를 p라고 하자. 이때 $n=(n/p)\cdot p$라는 결과 **한 번으로만** n이 합성수임을 확인하도록 하자.

Algorithm 4.3.1 (Basic Linear Sieve)

pr_i는 i가 소수인지 나타내는 배열이다. $\mathcal{O}(n)$에 $pr_1,\cdots,pr_n$을 계산한다.

- pr_1을 거짓으로, 각 $2\le i\le n$에 대해 pr_i를 참으로 둔다.
- 또한, 소수들을 저장하는 std::vector v를 준비한다.
- 각 $2\le i\le n$에 대하여, 아래 과정을 반복한다.
- 만약 pr_i가 참이라면, v에 i를 추가한다.
- v에 있는 각 소수 p에 대해서 다음을 과정한다.
- pr_{ip}를 거짓으로 한다. 또한, 만약 $p|i$라면 v에 대한 iteration을 break.

위 알고리즘이 $\mathcal{O}(n)$에 잘 작동된다는 것을 보이려면, 각 합성수 m에 대하여 pr_m이 거짓으로 설정되는 횟수가 정확히 한 번임을 보이면 된다. 이를 위해서, m의 최소 소인수를 p라고 하자. m이 합성수이므로, $p\le m/p$가 성립한다. 또한, m/p의 최소 소인수는 p 이상이다. 그러므로, $i=m/p$인 단계에서 v에 대한 iteration을 할 때 p 역시 계산 과정에 들어오고 (iteration이 break 되지 않는다) $m/p\cdot p=m$의 pr 값이 거짓으로 바뀌게 된다. 그렇다면 다른 i에 대해서는 어떻게 될까? pr_m이 거짓으로 설정되려면 $ip=m$인 소수 p가 있어야 하고, p는 m/p의 최소 소인수보다 작거나 같아야 한다. 만약 p가 m의 최소 소인수가 아니라면, m/p의 최소 소인수는 m의 최소 소인수와 같게 되고, 이는 p보다 작게 된다. 그러니 pr_m을 거짓으로 설정하는 과정을 거치기 전에 iteration이 break가 된다.

결론적으로 각 합성수를 정확히 한 번씩 방문하게 되고, 시간복잡도는 $\mathcal{O}(n)$이 된다.

Example 4.3.2

90이 합성수임을 언제 확인할까? $ip=90$이려면 $(p,i)=(2,45),(3,30),(5,18)$인데,

- $(p,i)=(2,45)$인 경우 실제로 pr_{90}이 거짓으로 바뀌고
- $(p,i)=(3,30)$인 경우 $2|30$이므로 $p=2$에서 바로 iteration이 break.
- $(p,i)=(5,18)$인 경우도 $2|18$이므로 $p=2$에서 바로 iteration이 break.

여기서 몇 가지 재밌는 점은, 각 합성수 m에 대해, pr_m이 거짓으로 바뀌는 순간 p의 값이 m의 최소 소인수가 된다는 점이다. 그러므로, 앞서 4.1에서 언급한 최소 소인수의 계산이 Linear Sieve를 활용하면 자동으로 된다. 그렇다면 Multiplicative Function은 어떻게 될까?

이를 계산하기에 앞서, i의 최소 소인수 lpf_i와 i가 갖는 lpf_i의 개수 ex_i를 계산하자. 이는

- $m = ip$일 때, $p \nmid i$라면 $lpf_m = p$, $ex_m = 1$이고
- $m = ip$일 때, $p | i$라면 $lpf_m = p$, $ex_m = ex_i + 1$
- $m = p$가 소수라면, $lpf_m = p$, $ex_m = 1$임

을 이용하면 동적 계획법과 같은 방법으로, Linear Sieve 과정에서 동시에 계산할 수 있다.

특히, $pp_i = lpf_i^{ex_i}$ 역시도 비슷하게 동적계획법으로 전부 $\mathcal{O}(n)$에 계산할 수 있을 것이다.

만약 우리가 multiplicative function f를 계산하고 싶다면, 비슷한 방식으로 $f(i)$를 가지고 $f(ip)$를 계산할 방법을 찾으면 된다. 이는 앞선 방법과 비슷하게, 점화식

- $m = ip$일 때, $p \nmid i$라면 $f(m) = f(ip) = f(i)f(p)$
- $m = ip$일 때, $p | i$라면 $f(m) = f(i) \cdot f(pp_m)/f(pp_i)$
- $m = p$가 소수라면, $f(m) = f(p)$

가 된다. 결국 i에서 ip로 넘어가는 과정에서 필요한 것은 prime power에 대한 f 값의 계산 한두번이다. 그런데 f를 동적계획법으로 계산한다고 하면, 잘 생각해보면

- $m = ip$이고 $p \nmid i$면, $f(i)$와 $f(p)$는 이미 계산되었고, 곱셈 한 번으로 $f(m)$을 얻는다.
- $m = ip$이고 $p | i$면, $m = pp_m$, 즉 m이 prime power가 아닌 이상 $f(pp_m), f(pp_i)$는 이미 계산되어있다. 즉, 이 경우에는 곱셈과 나눗셈 한 번으로 $f(m)$이 계산이 된다.
- 만약 m이 prime power라면, f를 직접 계산해야 할 것이다.
- 마지막으로, m이 소수인 경우에도, f를 직접 계산해야 할 것이다.

즉, 결론을 내리자면, prime power에 대해서 f의 값과 (함수의 값을 modulo로 계산해야 한다면 그 modular inverse)를 직접 계산해놓기만 한다면, 나머지 f의 값에 대해서는 동적계획법으로 하나의 값 당 $\mathcal{O}(1)$에 계산이 될 것이다.

특히, prime power에 대해서면 f를 계산하는 것은 어려운 일이 아니다. 예를 들어, $f(p^e)$의 계산에 $\mathcal{O}(e)$에 정도의 계산이 필요하다면, 다음 Theorem

Theorem 4.3.3 (Prime Number Theorem Strikes Again)

$$\sum_{p^e \le n} e = \pi(n) + \pi(n^{1/2}) + \cdots = \mathcal{O}(\frac{n}{\log n})$$

이다. 단, $\pi(x)$는 x 이하 소수의 개수이다.

이 성립하므로, f 전부를 계산하는 것에는 $\mathcal{O}(n)$ 미만의 시간이 걸린다. 특히, modular inverse까지 취하는 것까지 고려하더라도, $\pmod p$에서 계산한다 할 때 총 계산량이 $\mathcal{O}(n \log_n p)$ 수준으로 적다. 그러므로, 시간복잡도로 보았을 때 이 알고리즘 역시 상당히 효율적임을 알 수 있다. 다만, 구현을 효율적으로 하여 상수를 줄여야 시간적 이득을 볼 수 있을 것이다.

위 Chapter는 전체적으로 상당히 어려운 내용이므로, 천천히 공부하는 것을 추천한다. 특히 이번 Chapter의 내용들은 구현을 직접 해보는 것이 매우 좋은 연습이 될 것이다.

§4.4 Problems

Problem 4A. Algorithm 4.1.1을 구현하라.

Problem 4B. https://www.acmicpc.net/problem/16563

Problem 4C. Algorithm 4.1.4, 4.1.5를 구현하라.

Problem 4D. Algorithm 4.1.7을 구현하라.

Problem 4E. Section 4.2에서 설명한 알고리즘을 구현하라. f가 일차함수인 경우만.

Problem 4F. https://www.acmicpc.net/problem/8293

Problem 4G. https://www.acmicpc.net/problem/4798

Problem 4H. Chapter 9를 공부한 이후, https://projecteuler.net/problem=216

Problem 4I. Chapter 9를 공부한 이후, https://www.acmicpc.net/problem/20704

Problem 4J. Algorithm 4.3.1을 구현하라.

Problem 4K. Section 4.3에서 설명한 알고리즘을 구현하라.

Problem 4L. OPTIONAL : Theorem 4.1.3, 4.2.2, 4.3.3의 증명을 공부하라.

Problem 4M. OPTIONAL : 소수만 구한다고 했을 때, 극한의 시간 최적화를 해보자. 좋은 시작점은 https://codeforces.com/blog/entry/75852

5 Fermat's Theorem, Euler's Theorem

§5.1 The Two Theorems

Theorem 5.1.1 (Euler's Theorem)
자연수 n, a가 $\gcd(a, n) = 1$를 만족시킨다면, 다음이 성립한다.

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

$n = p$가 소수인 경우를 생각하면, 다음 결과

Theorem 5.1.2 (Fermat's Little Theorem)
소수 p와 자연수 a가 $p \nmid a$를 만족한다면, 다음이 성립한다.

$$a^{p-1} \equiv 1 \pmod{p}$$

Proposition 5.1.3 (Modular Inverse)
자연수 n, a가 $\gcd(a, n) = 1$을 만족한다면,

$$a^{-1} \equiv a^{\phi(n)-1} \pmod{n}$$

이 성립한다. 특히, $n = p$가 소수이고 $p \nmid a$라면,

$$a^{-1} \equiv a^{p-2} \pmod{p}$$

이는 $n = p$가 소수인 경우, $a^{-1} \pmod{p}$의 계산을 위해 유클리드의 알고리즘을 사용하는 대신, $a^{p-2} \pmod{p}$를 직접 계산하는 방법을 사용할 수 있음을 의미한다. a^{p-2}과 같은 거듭제곱의 계산은 binary exponentiation으로 로그 시간에 계산할 수 있다.

n이 소수가 아닌 경우에는 이야기가 달라지는데, $\phi(n)$을 계산하는 일이 간단하지 않기 때문이다. 예를 들어, $n = pq$가 서로 다른 두 소수의 곱이라면, $\phi(n)$을 계산하는 것은 n을 소인수분해하는 것만큼이나 어려운 문제이다. 소인수분해가 어려운 문제라는 사실은 정수론 알고리즘과 암호학에서 핵심적인 사실이다. 이에 비해, 유클리드 알고리즘을 이용하여 modular inverse를 계산하는 것은 소인수분해를 필요로 하지 않으며, 로그 시간에 계산이 보장된다. 그러니 modular inverse의 계산은 유클리드 알고리즘을 이용하는 것이 좋다.

§5.2 Properties of the Euler-Phi Function

Theorem 5.2.1 (Euler-Phi Function)
(1) : $\sum_{d|n} \phi(d) = n$
(2) : $m|n$이면 $\phi(m)|\phi(n)$이 성립한다.
(3) : $\phi(n)$은 $n \geq 3$이면 짝수이다.
(4) : $n > 2$에 대하여, 다음 부등식이 성립한다.

$$\phi(n) > \frac{n}{e^{\gamma} \log \log n + 3(\log \log n)^{-1}}$$

단, γ는 Euler's constant로, $e^{\gamma} \approx 1.7810724$.

(4)는 증명하지 않고, (1), (2), (3)만 증명한다.

Proof. (1) : 앞서 Chapter 3에서 증명하였다.
(2) : 소수 p와 $a \leq b$에 대하여 $\phi(p^a)|\phi(p^b)$가 성립함은 쉽게 증명할 수 있다.
이제 $m = p_1^{e_1} \cdots p_k^{e_k}$, $n = p_1^{f_1} \cdots p_k^{f_k} \cdot q_1^{d_1} \cdots q_l^{d_l}$이라 하자.
이때, $m|n$이므로 $e_i \leq f_i$가 각 $1 \leq i \leq k$에 대해 성립해야 한다. 그러므로,

$$\phi(m) = \prod_{i=1}^{k} \phi(p_i^{e_i}) | \prod_{i=1}^{k} \phi(p_i^{f_i}) | \phi(n)$$

이 성립하게 되어 증명이 끝난다.
(3) : 만약 n이 홀수라면, $\gcd(a, n) = 1$이면 $\gcd(n - a, n) = 1$ 역시 성립하게 된다. 특히, n이 홀수이므로 $a \neq n - a$가 성립하고, 결국 n과 서로소인 자연수들을 $(a, n - a)$ 형태로 짝지을 수 있다. 만약 n이 짝수라면, 마찬가지로 $(a, n - a)$를 짝지을 수 있는데, $a = n/2$인 경우 $a = n - a$이므로 짝이 지어지지 않는다. 그러나 $n \geq 3$이므로, $\gcd(n/2, n) = n/2 > 1$이 되어 이는 고려할 필요가 없다. 그러니, 두 경우 모두에서 $\phi(n)$은 짝수. □

(4)의 증명은 매우 어려운 것으로 알고 있으며, weaker version은 연습문제로 남긴다.

§5.3 Computation of Powers and Power Towers

앞서 우리는 Fermat's Theorem, Euler's Theorem을 소개하고 이를 통해서 modular inverse를 구하는 방법을 설명했다. 그러나 이는 유클리드 알고리즘의 하위호환 알고리즘임 역시 설명했다. 그렇다면 한 Chapter를 이 정리를 위해 투자한 이유는 무엇일까?

Proposition 5.3.1
자연수 a, n이 $\gcd(a, n) = 1$을 만족한다면, 다음 합동식이 성립한다.

$$a^x \equiv a^{x \pmod{\phi(n)}} \pmod{n}$$

이는 $a^{\phi(n)} \equiv 1 \pmod{n}$이니, 이를 반복해서 적용한 것으로 보면 된다. 만약

- x가 지나치게 커서 x를 계산하거나 $\mathcal{O}(\log x)$ 알고리즘을 적용하기 어렵고
- 대신 $x \pmod{\phi(n)}$은 계산하기 어렵지 않은 상황이라면

이 사실이 $a^x \pmod n$을 계산하는 것에 도움을 줄 수 있다. 이런 상황의 예시 중 하나로 Power Tower라는 유형의 문제가 있다. 이제부터의 설명은 기존 글인 https://rkm0959.tistory.com/181 의 설명을 압축한 것이다. 유도 과정을 처음부터 끝까지 확인하고 싶다면, 이 글을 읽는 것을 추천한다. 여기서는 핵심 결과와 그 증명만을 다룬다. notation의 편의를 위해

$$[a_1, a_2, \cdots, a_k] = a_1^{a_2^{\cdots^{a_k}}} = a_1^{[a_2, \cdots, a_k]}$$

라고 하자. 우리의 목표는 주어진 $a_1, a_2 \cdots, a_k, n$에 대하여

$$[a_1, a_2, \cdots, a_k] \pmod n$$

의 값을 계산하는 것이다. 한편, 1은 아무리 거듭제곱을 해도 1이므로,

$$[a_1, a_2, \cdots, a_k, 1, b_1, b_2, \cdots, b_l] = [a_1, \cdots, a_k]$$

가 성립한다. 그러니 1 뒤에 오는 값들은 무시해도 좋다.

그러므로, $a_i \geq 2$인 경우만 다루어도 충분함을 알 수 있으며, 앞으로 이를 가정하겠다.

이제 값을 계산하기 위해서, 문제를 더 작은 문제로 축소하는 접근을 사용한다.

Proposition 5.3.2

$n \leq 2^{64}$, $[a_2, \cdots, a_k] > 100$인 경우, 다음 합동식이 성립한다.

$$[a_1, \cdots, a_k] = a_1^{[a_2, \cdots, a_k] \pmod{\phi(n)} + 100 \cdot \phi(n)} \pmod n$$

Proof. 중국인의 나머지 정리에 의하여, n의 각 prime power p^e에 대해서 양변이 $\pmod{p^e}$로 같음을 보이면 충분하다. 이때, $n \geq p^e \geq 2^e$이므로 $e \leq \log_2 n \leq 64$가 성립한다.

Case 1 : a_1이 p의 배수가 아니다.
Theorem 5.2.1의 (2)에 의해 $\phi(p^e) | \phi(n)$이 성립한다. 그러므로

$$[a_2, \cdots, a_k] \equiv ([a_2, \cdots, a_k] \pmod{\phi(n)} + 100 \cdot \phi(n)) \pmod{\phi(p^e)}$$

가 성립하고, Euler's Theorem에 의하여 원하는 식을 얻는다.

$$[a_1, \cdots, a_k] \equiv a_1^{[a_2, \cdots, a_k]} \equiv a_1^{[a_2, \cdots, a_k] \pmod{\phi(n)} + 100 \cdot \phi(n)} \pmod{p^e}$$

Case 2 : a_1이 p의 배수이다.
이 경우, $[a_2, \cdots, a_k] > 100 > e$이고 $100 \cdot \phi(n) > 100 > e$이므로

$$[a_1, \cdots, a_k] \equiv a_1^{[a_2, \cdots, a_k]} \equiv a_1^{[a_2, \cdots, a_k] \pmod{\phi(n)} + 100 \cdot \phi(n)} \equiv 0 \pmod{p^e}$$

Remark : 이 증명 역시 CRT-style 사고방식의 예시임을 알 수 있다. □

특히, 여기서 $n \leq 2^{64}$와 100이란 상수들은 크게 중요하지 않으며, 100이라는 값을 $\log_2 n$ 이상의 값으로 두면 충분함을 알 수 있다. 어쨌든 위 proposition을 사용하면 기존의 문제를

$$[a_2, \cdots, a_k] \pmod{\phi(n)}$$

을 계산하는 문제로 축소할 수 있다.

다만, $[a_2, \cdots, a_k]$의 값이 충분히 큰지 판단하는 과정이 필요한데, $k \geq 5$만 되더라도

$$[a_2, \cdots, a_k] \geq [2, 2, 2, 2] = 65536$$

이므로 충분히 크다는 것이 보장되어 있다. 그러니, $k \leq 4$인 경우만 조심해서 따로 처리해 주면 된다. 이 부분은 정수론적인 지식이 필요없는 casework이므로 설명하지 않는다.

문제가 축소되면, Power Tower의 길이와 modulus의 값이 감소한다. 그러므로,

- Power Tower의 길이가 1이 될 때까지 이를 반복하면 문제가 해결된다.
- modulus가 1이 될 때까지 이를 반복하면 문제가 해결된다.

그 중 아래의 결과에 대해서 조금 더 깊게 생각을 해보자. n을 $\phi(n)$으로 바꾸는 것을 몇 번 해야 1을 얻을 수 있을까? 이를 알기 위해서는 Theorem 5.2.1의 (3)이 필요하다. $\phi(n)$은 $n \geq 3$이면 무조건 짝수이고, m이 짝수라면 모든 짝수가 m과 서로소가 아니게 되므로 $\phi(m) \leq m/2$가 성립하게 된다. 그러므로, $\phi(\phi(n)) \leq \phi(n)/2$가 성립하고 그 뒤로 ϕ를 적용할 때마다 값이 절반 이하로 감소하게 된다. 이는 결국 $\mathcal{O}(\log n)$번이면 n이 1로 바뀌게 된다는 것을 의미한다. 이는 중요한 결과이므로 proposition으로 기록한다.

Proposition 5.3.3

n을 $\phi(n)$으로 바꾸는 과정을 반복하여 1이 될 때까지 반복해야 하는 횟수를 k라 하면,

$$k = \mathcal{O}(\log n)$$

연습문제에서 이 분석이 up to constant tight 함을 증명할 것이다. 즉, $k = \Theta(\log n)$.

Algorithm 5.3.4 (Power Towers)

$a_1, a_2, \cdots, a_k \geq 2$와 $n \leq 2^{64}$가 주어진다. $[a_1, \cdots, a_k] \pmod{n}$을 구한다.

- $k \leq 4$이면 $[a_2, \cdots, a_k] \leq 100$인지 확인한다.
- 만약 100 이하라면, $[a_2, \cdots, a_k]$를 직접 계산하고 $[a_1, \cdots, a_k]$을 계산한다.
- 만약 100 초과라면, $\phi(n)$을 계산하고 $[a_2, \cdots, a_k] \pmod{\phi(n)}$을 구한다.
- Proposition 5.3.2를 적용하여 $[a_1, \cdots, a_k] \pmod{n}$를 계산한다.

위 알고리즘에서 $n \to \phi(n)$ 과정은 $\mathcal{O}(\log n)$번 이루어진다. 알고리즘에서 가장 시간이 많이 필요한 계산은 $\phi(n)$의 계산이고, 이는 단순하게 $\mathcal{O}(\sqrt{n})$에 할 수 있다. 그러므로, 전체 시간복잡도는 $\mathcal{O}(\sqrt{n} \log n)$임을 쉽게 확인할 수 있다. 하지만 실제로는 시간복잡도가 대강

$$\sqrt{n} + \sqrt{\phi(n)} + \sqrt{\phi(\phi(n))} + \cdots$$

에 비례하므로, ϕ가 결국에는 절반씩 감소한다는 점을 이용하면 시간복잡도가 $\mathcal{O}(\sqrt{n})$임을 알 수 있다. 물론, ϕ를 계산하기 위해서 Pollard-Rho를 이용하면, 이는 $\tilde{\mathcal{O}}(n^{1/4})$가 된다.

§5.4 Problems

Problem 5A. https://www.acmicpc.net/workbook/view/6596

Problem 5B. Theorem 5.1.1과 5.1.2를 증명하라.

Problem 5C. $n = pq$가 서로 다른 두 소수의 곱이라고 하자.
$\phi(n), n$을 알면, n의 소인수분해를 빠르게 계산할 수 있음을 보여라.

Problem 5D. Theorem 5.2.1의 (1)을 다음과 같은 다른 방법으로 증명하라.
(1) : 각 $d|n$에 대하여, $\gcd(k, n) = n/d$인 $1 \le k \le n$의 개수가 $\phi(d)$임을 보여라.
(2) : (1)의 결과를 사용하여, Theorem 5.2.1의 (1)을 증명하여라.

Problem 5E. Theorem 5.2.1의 (4)의 weaker version인,

$$\phi(n) > \frac{n}{4 \log n}$$

을 증명하라. 풀이는 https://tamref.com/58 참고.

Problem 5F. OPTIONAL : USA TSTST 2016. 자연수 n, k가 있어

$$(\phi \circ \phi \circ \cdots \phi)(n) = 1$$

이라 하자. 단, ϕ는 총 k번 적용되었다. 이때, $n \le 2 \cdot 3^{k-1}$을 보여라.

이는 n을 1로 만들기 위해 필요한 ϕ의 적용 횟수가 $\Omega(\log n)$임을 의미한다. 앞서 우리는 이 적용 횟수가 $\mathcal{O}(\log n)$임까지 보였으니, 이는 결국 ϕ 적용 횟수가 $\Theta(\log n)$임을 의미한다.

Problem 5G. https://codeforces.com/problemset/problem/776/E

6 Factorials and Binomials

7 More on Euclidean Algorithm

III

Hard Problems in Number Theory

8 Prime Testing and Prime Factorization

9 Discrete Logarithm/Roots

10 Counting Primes

IV

Multiplicative Functions

11 Mobius Inversion

12 Sum of Multiplicative Functions

V

Appendix

13 Further Notes