이 예습자료에서는 이번 시간 수업에 필요한 몇 가지 주제를 차례대로 보도록 하겠습니다. 먼저 서로소의 개념과 최대공약수를 구하는 방법에 대해 알아보고 그 후에는 소인수 분해하는 방법에 대해 알아보겠습니다. 이미 개념을 알고 있더라도 다시 한번 복습하면서 구현 방법까지 이해해 보세요. 이 자료에서 활용하는 코드는 이번 시간 수업 자료 중 GCD.py에 있습니다.

<서로소(relative prime)와 최대공약수(gcd, greatest common divisor)>

'서로소'와 '최대공약수'는 연관된 개념으로 함께 자주 사용됩니다. 이들의 의미를 복습해 보고 효율적인 구현 방법에 대해서도 알아보겠습니다. **개념을 다시 정확히 보다 보면 구현 방법에 대한 아이디어도 떠올릴 수 있습니다.**

두 수 A와 B 간에 공통된 약수가 1뿐일 때 A와 B를 서로소라 합니다. 즉 1 이외에는 A와 B를 동시에 나눌 수 있는 수가 존재하지 않을 때 A와 B를 서로소라 합니다.

예를 들어 21과 22는 서로소입니다.
21의 약수는 {1, 3, 7, 21} 이며
22의 약수는 {1, 2, 11, 22} 이므로
두 수의 공통된 약수는 **1** 뿐이기 때문입니다.

하지만 21과 24는 서로소가 아닙니다.
21의 약수는 {1, 3, 7, 21} 이며
24의 약수는 {1, 2, 3, 4, 6, 8, 12, 24} 이므로
두 수의 공통된 약수는 {1, 3}이기 때문입니다.

'서로소' 라는 단어는 영문 'relatively prime'에서 나왔습니다. Relatively prime은 어떤 수가 그 수 하나만을 보았을 때 절대적으로 소수(absolutely prime)라는 뜻이 아니라 **두 수를 함께 보았을 때 이들 간에 상대적으로** 소수(relatively prime)라는 의미입니다. 앞에서 본 예에서도 21과 22 각각은 절대적으로 소수가 아닙니다. 1과 자신 외에도 다른 약수가 있기 때문입니다. 하지만 이들을 함께 보면 상대적으로 소수입니다. 공통적인 약수는 1뿐이기 때문입니다. 이렇게 '상대적으로' 혹은 '서로 간에' 소수라는 의미로 '서로소'라고 번역해 사용하게 되었습니다.

서로소를 나타내는 단어는 여러 가지입니다. 아래 3가지 문장은 같은 의미입니다.
"A and B are **relatively prime,**"
"A and B are **coprime** to each other,"
"A and B are **mutually prime**."

두 수 A와 B가 서로소임을 확인하기 위해 많이 사용하는 방법은 **서로소의 정의 그대로** A와 B의 공약수가 1뿐임을 확인하는 것입니다. 이를 다르게 표현하면 **A와 B의 최대공약수가 1임을 확인**하는 것입니다. '**최대공약수**'는 'greatest(최대) common(공통된) divisor(약수)'를 그대로 번역한 단어로 종종 줄여서 gcd라 합니다. **이 문서에서도 앞으로 최대공약수를 gcd라 하겠습니다.**

최대공약수 gcd를 계산하는 방법으로 많이 사용하는 알고리즘 중 하나가 유클리드 알고리즘(Euclidean algorithm, 유클리드 호제법) 입니다. 이 방법은 다음 사실에 기초합니다.

두 수 A와 B (A > B)가 있을 때 A를 'A - B'로 대체하더라도 두 수의 gcd는 변하지 않는다. - (1)

예를 들어, 252(=21×12)와 105(=21×5)의 gcd는 21입니다. 이때 둘 중 더 큰 수 252를 두 수의 차인 147(=252-105)로 대체하더라도 gcd는 여전히 21 입니다. 147=21×7 이고 105=21×5 이기 때문입니다. 정리 (1)은 아래 [표 1]과 같이 증명할 수 있습니다.

먼저 A=c×p, B=c×q로 두겠습니다. c가 A와 B의 gcd이고, {p,q}는 서로소입니다. 이제 A를 A-B로 대체해 봅시다. A-B=c×(p-q) 이고 B=c×q 이므로 c는 여전히 A-B와 B 간의 공약수입니다. 하지만 만약 p-q와 q가 서로소가 아니고 1보다 큰 공약수 a를 갖는다면 gcd는 c×a로 변해야 할 것입니다. 여기서 p-q와 q가 서로소이며, 1 외에는 공약수가 없음을 증명하기 위해 모순에 의한 증명(proof by contradiction)을 사용해 보겠습니다. p-q와 q가 서로소가 아니고 1보다 큰 공약수 a를 갖는다고 가정합니다. 만약 그러하다면 (p-q)=a×j, q=a×k라 둘 수 있습니다. 이렇게 둔 후 p-q와 q를 더해보면 (p-q)+q=p=a×(j+k)가 됩니다. 즉 p=a×(j+k), q=a×k가 되므로 p와 q 간 1보다 큰 공약수 a가 있다는 뜻이 되며, 이는 p와 q가 서로소라는 사실에 모순됩니다. 따라서 'p-q와 q가 서로소가 아니고 1보다 큰 공약수 a를 갖는다'라는 가정은 거짓이며, p-q와 q는 서로소여야 합니다. p-q와 q가 서로소임을 보였으므로 A-B와 B 간의 gcd가 A와 B 간의 gcd와 같게 c임이 증명되었습니다.

[표 1]. 정리 (1)에 대한 증명

정리 (1)을 사용하여 유클리드의 알고리즘을 구현하면 아래 [표 2]와 같습니다. 함수의 두 인자 값이 a=c×p, b=c×q (c는 gcd, p와 q는 서로소)에서 시작했다고 가정합시다. 정리 (1)에 보인 바와 같이 두 인자 중 더 큰 수를 큰 수와 작은 수의 차로 계속 대체하다 보면 (라인 02~03) 두 값이 점점 작아지게 되며 결국 두 수 모두 gcd인 c가 됩니다 (라인 01).

```
00 def gcd1(a, b):
01     if a == b: return a  # if a == b, then gcd is a or b
02     if a > b: return gcd1(a - b, b)
03     else: return gcd1(a, b - a)
```

[표 2]. 유클리드 알고리즘의 첫 번째 구현>

[Q] [표 2]의 함수(유클리드 알고리즘의 첫 번째 구현)를 a=21과 b=24에 대해 실행하였다. 이 함수는 a==b가 되어 gcd를 찾을 때까지 재귀호출을 계속할 것이다. 이 함수를 재귀호출하는 과정을 순서대로 볼 때 아래의 괄호에 들어갈 값은 보기 중 무엇인가?

gcd1(21,24); -> gcd1(21,3); -> gcd1(18,3); -> ( ) -> gcd1(12,3); -> gcd1(9,3); -> gcd1(6,3); -> gcd1(3,3); -> return 3

gcd1(17,3)
gcd1(16,3)
gcd1(15,3)
gcd1(14,3)
gcd1(13,3)

※ 이 자료에서 [Q]로 제시된 문제는 학습 후 풀이하는 온라인 퀴즈에 그대로 나오니 학습하면서 그때그때 문제를 풀어 두세요. 온라인 퀴즈에서 보기의 순서는 바뀔 수 있으니 유의하세요. (예: 보기 1이 보기 2가 되고, 보기 2가 보기 1이 될 수 있음)

[표 2]의 구현 방법에는 한 가지 문제점이 있습니다. 두 인자 중 한 값이 다른 값보다 매우 큰 경우에 많은 뺄셈 연산을 수행해야 한다는 것입니다. 예를 들어 a=3×2000, b=3×1에서 시작했다면 a를 a-b로 대체하는 뺄셈 연산을 1,999번 수행해야 할 것입니다.

이러한 단점을 해결한 것이 아래 [표 3]의 방법입니다. 큰 수 A를 'A - B'로 대체하는 대신 'A % B (A를 B로 나눈 나머지)'로 대체합니다 (라인 12~13). 'A % B'는 A가 B보다 작아질 때까지 A를 'A-B'로 대체하는 연산을 여러 번 계속해서 수행한 결과와 같기 때문입니다. 단 % 연산은 뺄셈(-)과 다르게 두 수가 같아질 때 뺄셈을 멈추지 않고 0이 되어서야 뺄셈을 멈추는 연산으로 볼 수 있으므로 재귀호출의 종료 조건으로 한쪽 수가 0이 되었는지를 확인합니다 (라인 11). 그리고 두 수 중 % 연산을 거친 수가 더 작아지는데, 이렇게 더 작아지는 수가 함수의 두 번째 인자 b가 되도록 재귀호출하고 (라인 12~13) 최종적으로 b가 0이 되는지를 확인했습니다 (라인 11).

```
10 def gcd2(a, b):
11     if b == 0: return a
12     if a > b: return gcd2(b, a % b)
13     else: return gcd2(a, b % a)
```

[표 3] 유클리드 알고리즘의 두 번째 구현

[Q] [표 3]의 함수(유클리드 알고리즘의 두 번째 구현)를 a=21과 b=24에 대해 실행하였다. 이 함수는 a==0 또는 b==0이 되어 gcd를 찾을 때까지 재귀호출을 계속할 것이다. 이 함수를 재귀호출하는 과정을 순서대로 볼 때 아래의 괄호에 들어갈 값은 무엇인가? 직전 문제에서 [표 2]의 함수를 사용하는 경우와 비교할 때 함수를 호출하는 횟수가 크게 줄었음을 유의해서 보세요.

gcd2(21,24); -> gcd2(21,3); -> ( ) -> return 3

gcd2(18,3)
gcd2(3,0)
gcd2(6,3)
gcd2(3,3)

[표 3]의 방법은 재귀호출을 사용하므로 최대 호출하는 횟수에 제한이 있으며, 또한 함수를 호출하고 반환하는 오버헤드가 있습니다. 이를 줄이기 위해 기존에 배운 것처럼 재귀호출을 사용하지 않고 구현할 수 있습니다. 아래 [표 4]는 재귀호출을 사용하는 대신 while loop을 사용한 버전입니다. [표 3]과 같은 일을 함을 비교해 확인하세요.

```
20 def gcd3(a, b):
21     while b != 0:
22         if a > b: a, b = b, a % b
23         else: b = b % a
24     return a
```

[표 4] 유클리드 알고리즘의 세 번째 구현

라인 22의 'a, b = b, a % b'는 a에는 b 값을 저장하고, b에는 a%b 값을 저장함을 의미합니다. 이는 Python에서 여러 변수에 값을 동시에 할당할 때 사용하는 문법으로 일반적으로

$$a_1, a_2, a_3, a_4, \ldots, a_k = b_1, b_2, b_3, b_4, \ldots, b_k$$

는 a_i에 b_i를 할당함을 의미합니다. 이 문법을 사용하면

$$a, b = b, a$$

와 같이 (tmp 등의 임시 저장소를 사용하지 않고) 두 변수 a와 b에 저장된 값을 swap할 수도 있습니다.

정리하면 두 수가 서로소임을 확인하기 위해 gcd가 1임을 확인하는데, 이때 지금까지 보았던 유클리드의 알고리즘을 사용합니다.

유클리드의 알고리즘은 서로소를 확인하는 외에도 분수(fraction) 계산에 자주 사용됩니다. 즉 분수 계산의 결과로 $\frac{n}{d}$를 얻었다면 n과 d 각각을 gcd(n,d)로 나눈 수로 대체하여 저장합니다. 예를 들어 분수 계산의 결과 $\frac{21}{24}$를 얻었다면 $\frac{21}{24} = \frac{21/\gcd(21,24)}{24/\gcd(21,24)} = \frac{21/3}{24/3} = \frac{7}{8}$로 단순화하여 저장합니다.

이처럼 분수를 저장하고 계산하는 클래스의 구현을 보면 gcd 함수를 반드시 포함하고 있으며 연산(+-*/ 등) 후마다 결과를 단순화하기 위해 gcd 함수를 호출하므로 gcd 계산은 가능한 한 효율적으로 구현할 필요가 있습니다. 유클리드 알고리즘의 시간복잡도는 ([표 3]과 [표 4]의 경우) 평균 log(min(a,b))으로 상당히 빠른 편입니다.

<소인수 분해(prime factorization)>

이번에는 '소인수 분해'의 개념을 복습하고 효율적인 구현 방법에 대해서도 알아보겠습니다.

소인수 분해는 주어진 수를 소수인 약수의 곱으로 분해하는 것입니다. 예를 들어 12=2×2×3 으로 소인수 분해할 수 있습니다. 2와 3이 소수이기 때문입니다. 소수(prime number)인 약수(factor)로 분해하므로 소인수 분해를 prime factorization이라 합니다.

기존 수업에서 product-sum number에 대한 문제를 풀 때도 주어진 수를 인수로 분해하였습니다. 이때의 인수분해는 소수가 아닌 수로의 분해도 허용하였으므로 소인수분해와는 다릅니다. 소수가 아닌 수로도 분해 가능하다면 12 = 2×**6** = 3×**4** 등과 같이 여러 다른 방식으로 분해할 수 있으며 이러한 모든 경우를 찾기 위해 tree 형태의 공간을 탐색하였습니다. 하지만 이와 달리 소수인 약수로만 분해하는 소인수분해는 **유일**합니다. 예를 들어 12를 소인수 분해하는 방법은 12 = 2×2×3만으로 유일합니다. 따라서, 기존에 사용하던 방법과는 탐색 방법이 다릅니다.

또한, 기존 수업에서는 주어진 범위의 소수를 모두 찾는 prime sieve 방법에 대해서도 배웠습니다. 소인수분해를 할 때도 소수가 필요하지만, prime sieve를 사용해 미리 소수를 찾을 필요는 없습니다. 곧 소인수분해 방법을 보겠습니다.

매우 큰 수를 (예: $>10^{100}$) 소인수 분해하는 데는 시간이 오래 걸리며 이를 활용해 복호화를 어렵게 한 암호 시스템도 있습니다. 하지만 너무 크지 않은 수에 대해서는 아래 [표 5]의 소인수분해 방법도 상당히 빠릅니다. 이 함수를 호출하면 소인수를 오름차순으로 리스트에 담아 반환합니다. 예를 들어 n=12에 대해 호출하면 "[2, 2, 3]"을 반환합니다.

```
30 def primeFactorization(n):
31     result = []
32     while n % 2 == 0: # Check to see if 2 is a prime factor
33         n /= 2
34         result.append(2)
35
36     p = 3 # Check to see if odd numbers in 3 ~ sqrt(n) are prime factors
37     while p*p <= n:
38         while n % p == 0:
39             n /= p
40             result.append(p)
41         p += 2
42
43     if n > 2: result.append(int(n)) # What is left in n must also be a prime factor if n > 2
44
45     return result
```

[표 5] 소인수분해 알고리즘

[표 5]의 함수는 다음 순서로 동작합니다.

① (라인 32~34) n을 더는 나눌 수 없을 때까지 2로 나눕니다. 2로 나눌 때마다 2를 소인수로 가진다는 의미이므로 2를 결과 리스트 result에 추가합니다. 짝수인 소수는 2밖에 없으므로 이 과정을 거치면 남은 수 n은 홀수가 되며 이후에 나올 모든 소인수도 홀수일 것입니다.

② (라인 36~41) while loop을 사용해 n을 p = 3 ~ $\sqrt{n}$ 까지의 홀수로 나누어 봅니다. while loop의 'p += 2'로 p를 2씩 증가시키므로 (라인 41) 홀수로만 나누어보게 됩니다. 각 인수 p에 대해서는 n을 더는 나눌 수 없을 때까지 p로 나누며 나눌 때마다 p를 결과 리스트에 추가합니다 (라인 38~40). while loop에서 p = n까지가 아닌 p = $\sqrt{n}$ 까지만 나누어보는 이유는 기존에 배운 prime sieve 알고리즘에서 p = $\sqrt{n}$ 까지 진행한 이유와 유사합니다. n이 소수가 아니라서 n보다 작은 소수로 나누어진다면 그중 한 소수는 반드시 $\leq \sqrt{n}$ 일 것이기 때문입니다. 만약 $\sqrt{n}$ 까지 나누어보았음에도 나누어지지 않는다면 n이 소수인 경우입니다.

③ (라인 43) 남은 수 n이 2보다 크다면 (n이 2일 수는 없습니다. 단계 ①에서 소인수 2를 모두 제거했기 때문입니다) 이 수는 소수이므로 결과 리스트에 추가합니다. 그렇지 않고 n이 1이라면 출력하지 않습니다. 만약 $n = p^k$ 형태였다면 라인 38~40의 while loop을 거치며 p로 k번 나누어진 후 1이 됩니다.

[표 5]의 함수를 n = 42인 경우에 대해 실행해 보겠습니다.

① (라인 32~34) 42 / 2 = 21로 나누어떨어지므로 **첫 번째 소인수 2**를 발견하고 결과 리스트에 추가합니다. 이때 n을 2로 나누므로 남은 수 n = 21이 됩니다.

② (라인 36~41) while loop이 p = 3에서 시작합니다. 21 / 3 = 7로 나누어떨어지므로 **두 번째 소인수 3**을 발견하고 결과 리스트에 추가합니다. 이때 n을 3으로 나누므로 남은 수 n = 7이 됩니다. 이제 while loop에서 p가 2 증가하여 p = 5가 되면 p > $\sqrt{7}$ 이므로 while loop은 종료합니다.

③ (라인 43) 남은 수 **n = 7** > 2이므로 **세 번째 소인수 7**을 발견하고 결과 리스트에 추가합니다.

마지막으로 지금까지 찾은 소인수 [2, 3, 7]을 반환하며 종료합니다 (라인 45).

지금까지 본 소인수분해 방법의 시간복잡도는 평균적으로 log(n)에 비례합니다. 예를 들어 $8 = 2^3$이라면 8을 2로 $\log_2(8) = 3$회 나누면 (라인 32~34) 소인수분해가 완료되며 소인수 [2, 2, 2]를 찾습니다.

[Q] [표 5]의 함수(소인수분해 알고리즘)를 n = 29에 대해 실행하였다. 라인 36~41에서는 어떤 p 값에 대해 while loop이 실행되는가?

3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
3, 4, 5
3, 5

<Python에서 연산 결과를 정수형으로 만드는 방법>

Python 언어에서 연산의 결과로 (예: +-*/) 반드시 정숫값이 필요한 때가 있습니다. 예를 들어 연산의 결과를 리스트의 index로 사용하려면 반드시 정수형으로 치환해야 합니다.

다음과 같은 두 연산은 float 타입의 결과를 냅니다.

```
>>> 5 / 4
1.25
>>> 4 / 4   # 나눗셈 연산은 두 피연산자가 모두 정수이고 나누어떨어지더라도 결과는 float 타입
1.0
>>> 5 * 4.0  # 곱셈 연산은 하나 이상의 피연산자가 float 타입이라면 그 결과는 float 타입
20.0
```

이러한 연산 결과를 int() 함수에 대입하면 정수 타입으로 변환할 수 있습니다. 아래 결과를 위 결과와 비교해 보세요. 이때 소수점 아래 값이 절삭 된다고 보면 됩니다.

```
>>> int(5 / 4)
1
>>> int(4 / 4)
1
>>> int(5 * 4.0)
20
```

특히 나눗셈 연산의 경우 a와 b가 모두 정수라면 'a / b' 대신 'a // b'를 사용하면 정수형 결과를 얻을 수 있습니다. 혹은 'a // b'를 a를 b로 나눈 '몫'이라고 볼 수도 있습니다. 아래 결과를 참조해 보세요.

```
>>> 5 // 4
1
>>> 4 // 4
1
>>> 21 // 5
4
```

[Q] 변수 a와 b에는 정숫값이 저장되어 있다. a를 b로 나눈 몫을 리스트의 index로 사용하고 싶다. 보기 중 올바른 방법은?

```
c[a / b]
c[a // b]
```