이 예습자료에서는 이번 시간 수업에 필요한 몇 가지 주제를 차례대로 보도록 하겠습니다. 이 자료에서 활용하는 코드는 이번 시간 수업자료 중 Preview.py에 있습니다.

<프로그램의 구간별 소요 시간 측정>

프로그램의 실행 속도를 개선할 필요가 있을 때는 먼저 **많은 시간이 소요되는 부분(bottleneck이라고도 합니다)을 찾아 이 부분을 개선하는데 먼저 시간을 투자**하는 것이 좋습니다. Bottleneck을 개선한다면 다른 부분을 개선하는 것보다 실행 속도를 더 많이 개선할 수 있기 때문입니다. 예를 들어 프로그램이 4개의 구간으로 나누어지고 이들 각각의 소요 시간을 측정한 결과 2ms, 10ms, 2000ms, 5ms를 얻었다면 세 번째 구간(2000ms)을 우선하여 개선하는 것이 좋습니다. 다른 구간은 개선하더라도 10ms 이내의 시간밖에 줄일 수 없지만, 세 번째 구간을 개선하면 1000ms 이상의 시간도 줄일 수 있기 때문입니다.

주어진 프로그램에서 어느 부분이 bottleneck인지 판단하려면 각 부분의 복잡도(complexity, 보통 $O(N^2)$ 등의 형식으로 표현)를 대략 계산해 보아야 합니다. 머릿속으로 간단히 떠올려 보는 것도 좋습니다. 하지만 복잡도를 정확히 계산하는 것이 늘 쉬운 것은 아니며 실수할 가능성도 있습니다. 따라서 이를 검증하기 위해 **각 부분을 수행하는 데 실제로 걸리는 시간도 함께 측정**해 보는 것이 도움이 됩니다. 또한 시간을 측정해 봄으로써 프로그램의 성능을 개선할 필요가 있는지, 프로그램의 수정 전후에 성능이 어느 정도로 개선되었는지, 그리고 성능이 충분히 개선되었으므로 성능 개선 작업을 중단해도 괜찮은지 등을 판단하는 데 도움이 됩니다.

프로그램의 수행에 걸리는 시간과 메모리 사용량 등 수행 과정을 관찰하는 것을 프로그램을 'profile(프로파일) 한다'고도 합니다. 이러한 profiling 기능은 많은 IDE(Integrated Development Environment, 통합개발환경으로 Visual Studio, Eclipse 등을 포함)에서 제공됩니다. 이 예습자료에서는 특히 Python 언어에서 자주 활용하는 timeit() 함수를 사용해 함수의 수행 시간을 측정해 보겠습니다. 이 함수는 Python 기본 라이브러리에서 제공하는 함수이므로 Python 언어만 설치되어 있다면 사용할 수 있습니다. 아래 코드는 timeit() 함수의 사용 예를 보여줍니다.

```
00 import timeit
01 import math
02
03 t = timeit.timeit(lambda: math.sqrt(1000), number=10000)
04 print(t)
```

위 코드는 먼저 **timeit 모듈**을 import 합니다 (라인 00). 우리가 사용하고자 하는 함수 timeit()이 이 모듈에 포함된 함수이기 때문입니다. timeit()을 사용해 예제로 math 모듈에 속한 함수의 수행 시간을 측정해 볼 것입니다. 따라서 math 모듈도 import 합니다 (라인 01).

이제 math 모듈에서 제곱근을 구하는 함수인 math.sqrt()의 수행 시간을 측정해 보겠습니다 (라인 03). 먼저 timeit() 함수는 **timeit 모듈**에 속한 함수이므로 **timeit**.timeit()으로 호출합니다. timeit() 함수는 여러 인자를 입력으로 받을 수 있는데, 예제에서는 두 인자를 사용했습니다. 첫 번째 인자는 수행 시간을 측정하고자 하는 함수이며, 키워드 'lambda:' 뒤에 기술합니다. 위 예제에서는 math.sqrt(1000), 즉 1000의 제곱근을 구하는 기능의 수행 시간을 측정합니다. 두 번째 인자는 이 함수를 몇 번 실행한 시간을 측정할 것인지를 기술하며, 키워드 'number=' 뒤에 기술합니다. 위 예제에서는 10,000번 반복 실행한 시간을 (10,000번 반복 실행한 시간의 총합) 측정하도록 하였습니다. 이제 timeit()함수는 math.sqrt(1000)을

10000번 실행한 후 측정한 시간을 초 단위로 반환합니다. 반환한 값은 변수 t에 저장되고, 화면에 출력됩니다 (라인 04). 예를 들어 0.0012014999999999942가 출력되었다면, math.sqrt(1000)을 10,000번 실행하는데 약 0.0012초가 걸렸다는 뜻입니다. 따라서 1번 실행하는 데는 평균 0.0012/10,000 초가 걸렸음을 알 수 있습니다.

이제 Preview.py에 있는 3개의 예제 함수를 실행한 시간을 측정하고 비교해 보겠습니다. 이 3개의 함수는 findAllSequenceDiv(), findAllSequenceDivSqrt(), findAllSequenceMult() 입니다. 세 함수 모두 같은 일을 하며, 입력으로 받는 3개 인자 N, smallestFactor, print2stdout의 의미도 같습니다. smallestFactor ~ N 범위의 수를 smallestFactor 이상의 수로 인수 분해할 수 있는 모든 경우를 탐색하되, print2stdout = True이면 찾은 해를 출력하고, False이면 출력하지 않습니다. 먼저 __main__ 아래의 다음 라인을 실행해 출력을 보며 어떤 일을 하는 함수인지 확인해 보세요.

```
10 # observe output of findAllSequence()
11 findAllSequenceDiv(12, 2, True)
12 print()
13 findAllSequenceDivSqrt(12, 2, True)
14 print()
15 findAllSequenceMult(12, 2, True)
```

이제 이 3개 함수의 수행 시간을 측정해 보겠습니다. __main__ 아래의 다음 라인이 이 중 한 함수의 수행 시간을 측정해 출력합니다.

```
20 # measure execution time of findAllSequence()
21 repeat = 10
22 findAllSequence = findAllSequenceDiv
23 tfindAllSequence = timeit.timeit(lambda: findAllSequence(1000, 2, False), number=repeat) / repeat
24 print(f"Average running time for findAllSequence is {tfindAllSequence:.10f}")
```

변수 repeat은 (라인 21) 함수를 몇 번 수행한 시간을 측정할지를 결정합니다. 이 값이 timeit() 함수의 number 인자로 사용되었음을 확인하세요 (라인 23). 특히 라인 23 마지막에는 time()이 측정해 반환한 수행 시간의 총합을 다시 repeat으로 나누어 1회 수행 시간의 평균을 변수 tfindAllSequence에 저장하도록 하였습니다.

timeit()을 사용해 수행 시간을 측정하는 함수는 findAllSequence인데 (라인 23), 이러한 이름을 가진 함수는 정의되지 않았습니다. 대신 라인 22와 같이 findAllSequence에 findAllSequenceDiv를 저장함으로써 라인 23에서 findAllSequenceDiv 함수의 수행 시간을 측정하도록 하였습니다. 이때 findAllSequenceDiv 함수의 코드를 findAllSequence에 저장한다기보다는, findAllSequenceDiv 함수의 코드가 메모리에 저장된 위치를 가리키는 reference가 findAllSequence 변수에 저장되며, 이때부터는 findAllSequence를 마치 findAllSequenceDiv를 의미하는 것처럼 (즉, alias(별칭)으로) 사용할 수 있습니다. 만약 findAllSequenceDiv가 아닌 다른 함수의 수행 시간을 측정하고 싶다면 라인 22에서 findAllSequence에 다른 함수의 reference를 저장하면 됩니다. 정리하면 클래스 객체도 그 reference를 여러 다른 별칭에 저장할 수 있듯이 함수도 그 reference를 여러 다른 별칭에 저장할 수 있습니다.

이제 위 코드를 실행해 함수의 수행 시간을 여러 차례 측정해 보세요. 그리고 라인 22에서 저장한 함수의

reference를 변경해 다른 함수의 수행 시간도 측정하고 비교해 보세요. 같은 함수의 수행 시간도 측정할 때마다 조금씩 다를 수 있습니다. 이는 PC에서 그때그때 수행하는 일이 달라지기 때문입니다. 예를 들어 갑작스러운 오류나 네트워크를 통해 들어오는 데이터 등을 처리하기 위해 시간 측정 작업에 지연이 생기기도 합니다.

[Q] Preview.py에 정의된 3개 함수 findAllSequenceDiv(), findAllSequenceDivSqrt(), findAllSequenceMult()의 수행 시간을 측정해 보자. 이들의 수행 시간 간의 대소 관계로 올바른 것은?

※ 'a < b'는 a보다 b를 수행하는데 시간이 더 걸림을 의미함

findAllSequenceDiv() < findAllSequenceDivSqrt() < findAllSequenceMult()
findAllSequenceDivSqrt() < findAllSequenceDiv() < findAllSequenceMult()
findAllSequenceMult() < findAllSequenceDiv() < findAllSequenceDivSqrt()
findAllSequenceMult() < findAllSequenceDivSqrt() < findAllSequenceDiv()

※ 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)

[Q] 다음 코드를 실행했을 때 출력 결과로 올바른 것은?

```
import math
f1 = math.sqrt
f2 = math.ceil
print(f1(9), f2(2.5))
```

함수 f1, f2가 정의되지 않고 호출되었으므로 오류 발생
3.0 3

<인수분해 vs. 모든 인수 찾기>

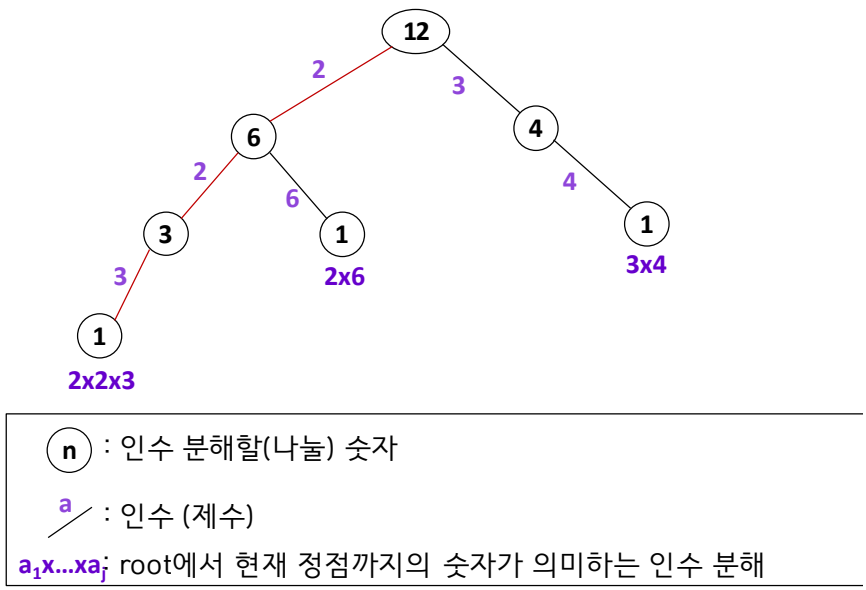
지난 시간에는 주어진 정수 N을 **인수 분해**해 보았습니다. 이번 시간에는 N의 **모든 약수**를 구해 볼 것입니다. 두 가지 모두 약수를 구해야 한다는 점에서 유사합니다. 특히 N이 큰 정수일 때 N의 약수를 찾고 분해하는 데 많은 시간이 걸리며, 이를 활용해 RSA 등의 암호 시스템이 개발되었습니다. 이렇게 유사한 점도 있지만 아래와 같이 차이점도 있으며, 이 때문에 코드도 약간 달라집니다.

① **인수분해(Factorization):** 정수 N을 '인수분해' 함은 N을 여러 **정수들의 곱으로 표현**하는 것을 말합니다. 예를 들어 정수 8은 1×8, 2×4, 2×2×2 등 여러 방식으로 인수 분해될 수 있습니다. 이 중 N을 소수(prime number)들만의 곱으로 표현하는 것을 소인수 분해(prime factorization)라고 하며 8을 소인수 분해하면 2×2×2가 됩니다.

② **모든 약수(factor, divisor) 구하기:** 정수 N의 '모든 약수를 구함'은 N을 나누었을 때 나누어떨어지는 모든 수를 구함을 의미합니다. 예를 들어 정수 8의 약수는 {1, 2, 4, 8}입니다. 음수 {-1, -2, -4, -8}도 약수가 될 수 있지만, 이번 시간에는 양의 약수만을 다루겠습니다. 모든 약수를 구할 때는 인수 분해할 때와는 달리 N을 나눈 몫이 1이 될 때까지 인수로 여러 차례 계속해 나누어 갈 필요는 없습니다. 대신 N을 나누었을 때 나머지가 0인 숫자만 찾으면 됩니다.

이제 ①과 ②의 구현 방법을 알아보겠습니다.

① **인수분해(Factorization):** 지난 시간에 배운 것처럼 N을 나눌 수 있는 수로 계속 나누어 나가되 1이 되면 중단합니다. 이렇게 가능한 수로 나누어 가는 과정은 아래 [그림 1]과 같이 트리 형태로 볼 수 있습니다. 최종적으로 나눈 결과가 1이라면 현재까지 나누어 온 수들의 곱이 하나의 유효한 인수분해가 됩니다. 예를 들어 그림 가장 왼쪽의 붉은 선을 따라 나누어 간다면 인수분해 2×2×3을 얻습니다. 같은 인수분해를 두 번 이상 중복해서 탐색하는 것을 막기 위해 직전에 나눈 수와 같거나 큰 수로만 나누어 갑니다. [그림 1]의 예에서 탐색한 인수분해 2×2×3, 2×6, 3×4 각각도 이러한 조건에 따라 탐색한 결과입니다 ($2 \le 2 \le 3$, $2 \le 6$, $3 \le 4$).



[그림 1] 12를 인수 분해하는 예 (1과 12 자신으로 인수분해 하는 경우는 제외)

② 모든 인수(factor, divisor) 구하기: 정수 N을 나누어떨어지는 모든 수를 구하는 가장 기본적인 방법은 아래 코드와 같이 N을 1~N까지의 각 정수로 차례로 나누어 나누어떨어짐을 확인하는 방법입니다. 이 방법은 N번의 나눗셈이 필요합니다.

```
30 def findDivisors(n):
31     result = []
32     for i in range(1, n+1):
33         if n % i == 0:
34             result.append(i)

35     return result
```

위 함수에 입력 n=100을 주어 실행한 결과를 출력해 보면 (print(findDivisors(100))), 아래와 같이 100의 모든 약수를 얻을 수 있습니다.

[1, 2, 4, 5, 10, 20, 25, 50, 100]

위 코드는 N을 인수 분해할 때와는 달리 재귀호출을 사용할 필요가 없고 훨씬 간단합니다. 왜 그런지 비교해 생각해 보세요. 또한 이 함수를 더 효율적으로 만들 방법이 있을지도 생각해 보세요.

[Q] 다음 중 올바른 것은?

16의 모든 약수를 구하면 {1,2,4,8,16} 이며, 16을 인수 분해하는 방법은 2×2×2×2, 2×2×4, 2×8, 4×4 등 여러 가지가 있다.
16을 인수 분해한 결과는 {1,2,4,8,16} 이며, 16의 모든 약수를 구하면 2×2×2×2, 2×2×4, 2×8, 4×4이다.

답: 16의 모든 약수를 구하면 {1,2,4,8,16} 이며, 16을 인수 분해하는 방법은 2×2×2×2, 2×2×4, 2×8, 4×4 등 여러 가지가 있다.

<Chain(사이클) 구하고 저장하기>

이번에는 집합의 원소가 이루는 사이클을 확인하고 이를 저장하는 방법에 대해 알아보겠습니다. 특히 0~10 사이의 정수가 집합의 원소이며, 원소 a의 다음 원소가 아래 표와 같은 함수 f(a)로 주어진다고 가정하겠습니다. 이 예제에서는 두 개의 사이클이 있는데, 하나는 2에서 시작해 2로 돌아오는 사이클 2→5→6→8→2입니다. f(2)=5, f(5)=6, f(6)=8, f(8)=2이기 때문입니다. 사이클의 원소를 한 번씩만 포함해 [2, 5, 6, 8]로 나타낼 수도 있습니다. 또 다른 사이클은 0에서 시작해 0으로 돌아오는 사이클입니다. 이러한 사이클을 찾아 저장하고, 필요할 때 이를 다시 재구성하는 것이 목표입니다.

a	0	1	2	3	4	5	6	7	8	9	10
f(a)	0	0	5	1	3	6	8	1	2	4	5

아래 코드는 원소의 집합과 함수 f()가 입력으로 주어졌을 때 사이클을 찾고 저장하는 함수를 보여줍니다. 이번 시간 실습에서 유사한 로직을 활용해 실습과제를 구현할 수 있도록 이 코드를 이해해 보겠습니다.

```
40 def findNstoreCycle(elements, f):
41     def recur(a):
42         if result[a] != None: return -1
43 
44         if a in dictionary: return a
45         else:
46             dictionary[a] = len(dictionary) # a가 사이클에서 몇 번째 원소인지 기록 (디버깅 위함)
47             firstElementInCycle = recur(f[a])
48             if firstElementInCycle == -1: # no cycle found that includes a
49                 result[a] = -1
50                 return -1
51             else:
52                 result[a] = f[a]
53                 if a == firstElementInCycle:
54                     cycleElements.append(a)
55                     return -1
56                 else: return firstElementInCycle
57 
58     result = [None for _ in range(len(elements))]
59     dictionary = {}
60     cycleElements = []
61     for e in elements:
62         dictionary.clear()
63         recur(e)
64 
65     return cycleElements
```

먼저 findNstoreCycle() 함수는 2개의 인자 elements와 f를 입력으로 받습니다 (라인 40). elements는

원소의 집합을 나타내고, f는 각 원소의 다음 원소인 f(a)를 나타냅니다. 이러한 입력을 받아 최종적으로 리스트 cycleElements를 반환합니다 (라인 65). 이 리스트에는 찾은 사이클마다 그 사이클에 속하는 원소 중 하나를 기록합니다. 예를 들어 아래와 같이 앞에서 본 예제와 같은 입력으로 함수를 호출한다면

elements = [0,1,2,3,4,5,6,7,8,9,10]
f = [0,0,5,1,3,6,8,1,2,4,5]

findNstoreCycle() 함수가 반환하는 값은 다음과 같습니다.

cycleElements = [0, 2] # 두 개의 사이클을 찾았으며, 하나는 0을, 다른 하나는 2를 포함함을 의미

이러한 결과가 있다면 필요한 경우 사이클을 구성하는 전체 원소를 찾을 수 있습니다. 예를 들어 2를 포함하는 사이클 전체를 찾고자 한다면 f[2] = 5, f[5] = 6, f[6] = 8, f[8] = 2를 차례로 계산해가다가 첫 원소인 2로 다시 돌아왔으므로 탐색을 멈추고 그때까지 거쳐온 원소인 [2, 5, 6, 8]을 출력하면 됩니다. 또 다른 예로 0을 포함하는 사이클 전체를 찾고자 한다면 f[0] = 0만 계산하면 바로 첫 원소인 0으로 돌아오므로 [0]을 출력합니다.

다음으로 findNstoreCycle() 함수가 사이클을 찾기 위해 사용하는 자료구조를 보겠습니다. 이 함수는 두 개의 자료구조를 사용하는데 리스트인 result와 심볼 테이블인 dictionary입니다. results에는 각 원소가 사이클에 속하는지의 여부를 기록합니다. 특히, 원소 a가 사이클에 속한다면 result[a]에 사이클에서 a 다음 원소를 (즉 f(a)) 저장하고, 사이클에 속하지 않는다면 -1을 저장합니다. 앞에서 본 예제와 같은 입력으로 함수를 호출했다면 최종적으로 result에는 아래와 같은 값이 저장됩니다. result[1], result[3], result[4], result[7], result[9], result[10]에는 -1이 저장되어 있는데, 이들은 사이클에 속하지 않기 때문입니다.

result = [0, -1, 5, -1, -1, 6, 8, -1, 2, -1, -1] # 각 원소가 사이클에 포함되는지 여부를 기록

두 번째 자료구조인 심볼 테이블 dictionary에는 f()에 따라 다음 원소를 탐색해 갈 때 지금까지 거쳐온 원소를 기록함으로써 기존에 보았던 원소로 다시 돌아가는지 (즉 사이클을 이루는지) 확인합니다. 예를 들어 2에서 탐색을 시작했다면 2, 5, 6, 8이 차례로 dictionary에 기록되며, 이어서 2로 다시 돌아간다면 2가 이미 dictionary에 기록되어 있으므로 사이클을 이룸을 확인하게 됩니다.

이제 findNstoreCycle() 함수의 코드를 차례로 따라가며 동작을 이해해 보겠습니다. 라인 41~56은 이 함수가 내부적으로 재귀 호출하는 recur() 함수를 정의합니다. findNstoreCycle() 함수를 호출하면 그 후인 라인 58부터 실행됩니다. 먼저 내부적으로 사용하는 자료구조 result와 dictionary 및 반환 값인 cycleElements를 초기화합니다 (라인 58~60). 특히 리스트 result는 모두 None으로 초기화하는데 (라인 58), 아직 어떤 원소에 대해서도 사이클에 속하는지 확인하지 않았다는 뜻입니다.

다음으로 입력 elements에 속한 각 원소 e에서 시작해 다음 원소를 차례로 따라가 보며 이 원소가 사이클에 속하는지 확인하는데 (라인 61~63), 이를 위해 recur(a) 함수를 호출합니다. recur(a) 함수는 원소 a에서 시작해 새로운 사이클을 찾을 수 있다면 이 사이클의 첫 원소를 반환하고, 더는 새로운 사이클을 찾을 가능성이 없다면 -1을 반환합니다. 이러한 반환 값에 따라 result에 사이클 여부를 기록합니다. 또한 recur() 함수는 다음 원소를 따라갈 때마다 재귀 호출되는데, 거쳐 가는 원소를 차례로 dictionary에 기록합니다.

recur(a) 함수는 다음과 같이 동작합니다. 먼저 result[a]가 None이 아니라면 이미 a를 포함하는 사이클이 존재하는지 기존에 확인했다는 뜻이므로 탐색을 중단하고 -1을 반환합니다 (라인 42). 다음으로 현재 다다른 원소 a가 이미 dictionary에 기록되어 있다면 사이클을 찾은 것이므로 a를 사이클의 첫 원소로 반환합니다 (라인 44). 만약 그렇지 않다면 현재 원소 a를 dictionary에 기록하고 (라인 46), 이어지는 다음 원소 f[a]를 탐색하기 위해 recur() 함수를 재귀호출합니다 (라인 47).

재귀호출한 결과 사이클이 발견되지 않는다면 result[a]에 사이클이 없음을 의미하는 -1을 기록하고 -1을 반환합니다 (라인 48~50).

만약 사이클이 발견되었다면 result[a]에 다음 원소 f[a]를 기록하고 (라인 52) 다시 사이클의 첫 원소를 반환합니다 (라인 56). 이렇게 반환하다가 사이클의 첫 원소까지 돌아왔다면 (라인 53) 이 원소를 결과 리스트인 cycleElements에 기록하는데 (라인 54) 첫 원소 이전에 거쳐온 원소는 더는 사이클의 일부가 아니므로 이때부터는 -1을 반환합니다 (라인 55).

지금까지 본 코드에 따라 리스트 result[]가 변해가는 과정은 아래와 같습니다. 먼저 reslut는 모두 None으로 초기화됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	None	None	None	None	None	None	None	None	None	None	None

이제 첫 원소인 0에서 시작해 다음 원소를 따라가 본 결과 0 자신에게 바로 돌아오는 사이클을 발견하게 되므로 result[0]에 0이 저장됩니다. 결과 리스트인 cycleElements에도 0이 추가됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	None	None	None	None	None	None	None	None	None	None

두 번째 원소인 1에서 시작해 다음 원소를 따라가 본 결과 0에 다다르게 되는데 0에서 시작하는 사이클과 이에 속하는 모든 원소는 이미 발견되었으므로 1은 이 사이클의 일부가 될 수 없습니다. 따라서 result[1]에는 -1이 저장됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	None	None	None	None	None	None	None	None	None

이제 세 번째 원소인 2에서 시작해 다음 원소를 따라가 본 결과 2, 5, 6, 8을 포함한 사이클을 발견합니다. 따라서 result[2], result[5], result[6], result[8]에는 사이클에서 다음 원소가 저장됩니다. 결과 리스트인 cycleElements에는 이 중 첫 원소인 2가 추가됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	5	None	None	6	8	None	2	None	None

이제 네 번째 원소인 3에서 시작해 다음 원소를 따라가 본 결과 1에 다다르는데 1에서 시작하는 사이클은

없다고 기록되어 있으므로 3은 사이클의 일부가 될 수 없습니다. 따라서 result[3]에는 -1이 저장됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	5	-1	None	6	8	None	2	None	None

다섯 번째 원소인 4에서 시작해 다음 원소를 따라가 본 결과 3에 다다르는데, 3에서 시작하는 사이클은 없다고 기록되어 있으므로 4도 사이클의 일부가 될 수 없습니다. 따라서 result[4] = -1이 됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	5	-1	-1	6	8	None	2	None	None

여섯 번째 원소 5와 일곱 번째 원소 6에 대한 사이클은 이미 발견되었다고 기록되어 있으므로 바로 탐색이 중단됩니다.

여덟 번째 원소 7에서 시작해 다음 원소를 따라가 본 결과 1에 다다르는데, 1에서 시작하는 사이클은 없다고 기록되어 있으므로 7도 사이클의 일부가 될 수 없습니다. 따라서 result[7] = -1이 됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	5	-1	-1	6	8	-1	2	None	None

아홉 번째 원소 8에 대한 사이클은 이미 발견되었다고 기록되어 있으므로 바로 탐색이 중단됩니다.

열 번째 원소 9에서 시작해 다음 원소를 따라가 본 결과 4에 다다르는데, 4에서 시작하는 사이클은 없다고 기록되어 있으므로 9도 사이클의 일부가 될 수 없습니다. 따라서 reuslt[9] = -1이 됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	5	-1	-1	6	8	-1	2	-1	None

마지막으로 원소 10에서 시작해 다음 원소를 따라가 본 결과 5에 다다르는데, 5를 포함한 사이클과 이에 속하는 모든 원소는 이미 발견되었으므로 10은 이 사이클의 일부가 될 수 없습니다. 따라서 result[10]에는 -1이 저장됩니다.

a	0	1	2	3	4	5	6	7	8	9	10
result	0	-1	5	-1	-1	6	8	-1	2	-1	-1

이렇게 모든 탐색이 종료되고 결과 리스트인 cycleElements에는 지금까지 발견한 두 사이클의 첫 원소인 [0, 2]가 담긴 상태로 반환됩니다. 지금까지의 탐색 과정은 result에 저장된 기존 탐색 결과를 재활용했기에 (동적 프로그래밍, DP) 각 원소를 단 한 번씩만 탐색하고 탐색을 마칠 수 있었음을 유의해 보세요.

[Q] elements = [0,1,2,3,4,5], f = [5,0,1,2,3,1]라면 몇 개의 사이클을 발견하게 되는가?

0을 포함하는 사이클 하나를 발견한다.
0을 포함하는 사이클 하나와 4를 포함하는 또다른 사이클을 발견한다.