어찌어찌 인연이 닿아서 동아리 후배 기수 분들과 함께 방학 동안 파이썬 알고리즘 스터디를 진행하게 되었다.
이왕 공부하는 거 열심히 공부하고 문제 많이 풀어서 백준 플래티넘 달성해보려고 한다. 신년 초 + 돌아온 호랑이 해 버프 받아 일단 목표는 크게...
아래 강의 영상에서 배운 내용을 정리해보려고 한다. 아는 내용이라 생략된 부분이 있을 수 있다.
https://youtu.be/m-9pAwq1o3w?list=PLRx0vPvlEmdAghTr5mXQxGpHjWqSz0dgC

온라인 문제 풀이 사이트 목록

코드포스(Codeforces): http://codeforces.com/
탑코더(Topcoder): https://www.topcoder.com/
릿코드(Leetcode): https://leetcode.com/
코드셰프(Codechef): https://www.codechef.com/
백준(BOJ): https://www.acmicpc.net/
코드업(Codeup): https://codeup.kr/
프로그래머스(Programmers): https://programmers.co.kr/
SW Expert Academy: https://swexpertacademy.com/

파이썬을 코딩 테스트 언어로 가장 추천하며,
코딩 테스트는 온라인 개발 환경에서 준비하는 것을 추천한다.

온라인 개발 환경 목록

Replit: https://replit.com/
PythonTutor: https://pythontutor.com/

오프라인 개발 환경 목록

Pycharm: https://www.jetbrains.com/ko-kr/pycharm/download
Dev-C++: https://sourceforge.net/projects/orwelldevcpp/

알고리즘 코딩 테스트 준비하는 과정에서 소스 코드 관리하는 습관 들이는 것을 추천(깃허브에 올리는 등)한다.
자주 사용하는 알고리즘 코드는 라이브러리화하면 좋다.

출제 빈도가 높은 알고리즘은 그리디, 구현, DFS/BFS를 이용한 탐색이다.

복잡도

복잡도: 알고리즘의 성능을 나타내는 척도
시간 복잡도: 특정한 크기의 입력에 대하여 알고리즘의 **수행 시간** 분석
공간 복잡도: 특정한 크기의 입력에 대하여 알고리즘의 **메모리 사용량** 분석
복잡도가 낮을수록 좋은 알고리즘이다.

빅오 표기법(Big-O Notation)

빅오 표기법은 가장 빠르게 증가하는 항만을 고려하는 표기법으로,
함수의 상한만을 나타내며 계수를 무시한다.
O(연산횟수 방정식에서 계수 뺀 최고차항) ← 이런 형태로 표기한다.
예를 들어

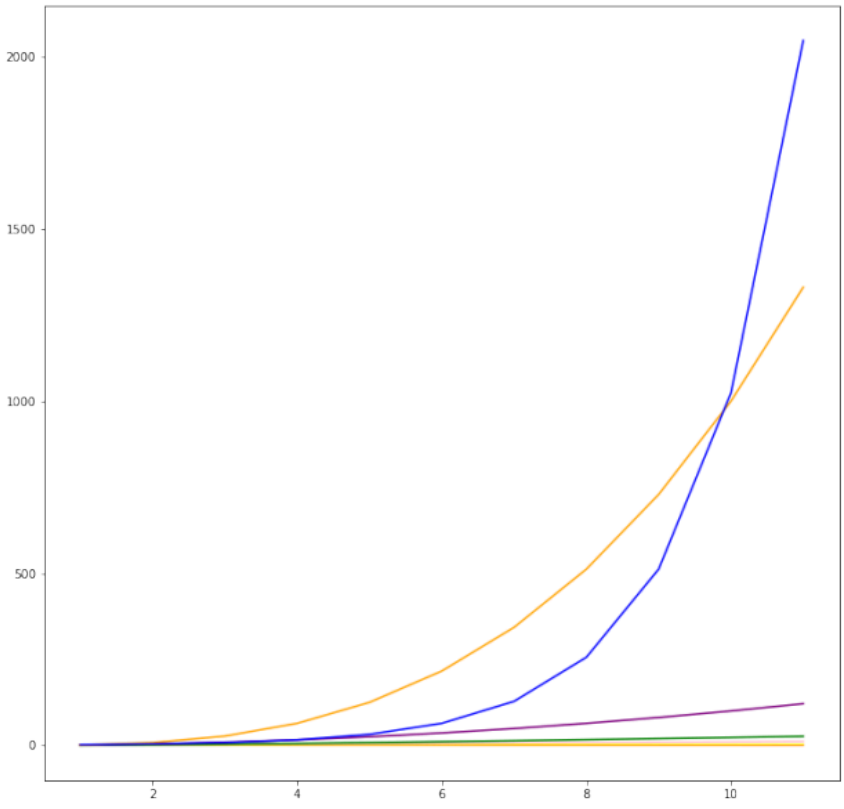
연산횟수가 $3N^3 + 5N^2 + 1000000$인 알고리즘 $\rightarrow O(N^3)$

빅오 표기법으로 자주 쓰이는 표기와 명칭

좋은 순서대로 작성(좋은 순서는 N이 증가하더라도 값이 적게 증가하는 순서 정도로 생각하면 될 것 같다.)

O(1): 상수 시간(constant time)
O(logN): 로그 시간(Log time)
O(N): 선형 시간
O(NlogN): 로그 선형 시간
O(N^2): 이차 시간
O(N^3): 삼차 시간
O(2^n): 지수 시간

그래프로 표현해보면 이해하기 쉬울 것 같아 빅오 함수들 비교 그래프를 만들어보았다.



그래프의 소스 코드

```python
import numpy as np
import matplotlib.pyplot as plt

# X는 1 이상 12 미만
start_num = 1
end_num = 12
X = np.arange(start_num, end_num)

# 상수 시간(constant time)
Y0 = [1 for i in range(start_num, end_num)]
# 로그 시간(Log time)
Y1 = [np.math.log(i) for i in range(start_num, end_num)]
# 선형 시간
Y2 = X
# 로그 선형 시간
Y3 = [i * np.math.log(i) for i in range(start_num, end_num)]
# 이차 시간
Y4 = X**2
# 삼차 시간
Y5 = X**3
# 지수 시간
Y6 = 2**X

# 그림(figure)의 크기를 가로 12인치,세로 12인치로 설정
plt.rcParams["figure.figsize"] = (12, 12)

# 그래프 그리기
plt.plot(X, Y0, color='red')
plt.plot(X, Y1, color='yellow')
plt.plot(X, Y2, color='pink')
plt.plot(X, Y3, color='green')
plt.plot(X, Y4, color='purple')
plt.plot(X, Y5, color='orange')
plt.plot(X, Y6, color='blue')

# 그래프 출력
plt.show()

# 그래프 저장
# plt.savefig('graph.png')
```

그래프를 무지개 색상으로 표현하다 알게 된 사실인데, 나라마다 무지개 색상을 다르게 보는 듯하다.
한국에서는 보통 빨주노초파남보라고 하는데, 어릴 때 배운 영어 동요(I Can Sing A Rainbow) 가사 생각해보니까 거기선 빨노분초보주파이다.
소스 파일은 동요 가사 생각하며 적다보니 미국 기준으로 되어 있다. 찾아보니 무지개 색상 개수를 다르게 보는 경우도 있는 것 같다.
각설하고 다시 돌아와서

알고리즘 설계 팁

일반적으로 연산 횟수가 5억을 넘어가면 통상 C는 1~3초, python은 5~15초 가량의 시간이 소요된다.
단, PyPy는 때때로 C보다 빠르게 동작하기도 한다.
코딩 테스트 문제에서 시간제한은 통상 1~5초 가량으로,
문제에 명시되어 있지 않다면 5초 정도라고 생각하고 문제를 푸는 것이 합리적이다.

요구사항에 따라 적절한 알고리즘 설계하기

문제를 보면 제일 먼저 시간제한(수행시간 요구사항)부터 체크해야 한다.

시간제한이 1초인 문제에서 일반적인 기준

앞서 파이썬은 연산횟수가 1억일 때 1~5초정도 걸린다고 했으니, 2천만~1억 정도의 시간복잡도가 나오면 되는 것으로 보인다.
replit에서는 단순 덧셈 반복문 천만번 돌려도 1초 정도 걸리는데... 참고만 하고 넘어가려고 한다.

n의 범위가 500 → 시간복잡도 O(n^3) → 1억 2천 5백만(?? 1억 비스무리하니까 일단 넘어간다.)
n의 범위가 2000 → 시간복잡도 O(n^2) → 4백만
n의 범위가 100000 → 시간복잡도 O(NlogN) → 5십만
n의 범위가 10000000 → 시간복잡도 O(N) → 천만

해당 범위 내에서 쓸 수 있는 가장 높은 복잡도의 알고리즘이라 생각하면 될 듯.
당연히 이보다 더 낮은 복잡도의 알고리즘을 쓴다면 좋을 것이다.

알고리즘 문제 해결 과정

일반적인 알고리즘 문제 해결 과정은 다음과 같다.

1. 지문 읽기 및 컴퓨터적 사고
2. 요구사항(복잡도) 분석
3. 문제 해결을 위한 아이디어 찾기
4. 소스코드 설계 및 코딩

일반적으로 문제는 핵심 아이디어를 캐치한다면, 간결하게 소스 코드를 작성할 수 있는 형태로 나온다.
즉, 출제자가 일부러 복잡한 구현 문제를 낸 게 아닌 이상 대부분의 소스 코드 자체는 쉽게 작성할 수 있다.
무작정 코드부터 작성하지 말고, 생각을 충분히 한 다음 문제의 핵심 아이디어를 캐치하는 데 집중해야 한다.
생각해보니 이거 예전에 프로그래밍 수업에서 교수님이 하셨던 말씀이다.

파이썬 수행 시간 측정 소스코드 예제

```
Import time
start_time = time.time() # 측정 시작

# 프로그램 소스 코드

end_time = time.time() # 측정 종료
print("time: ", end_time - start_time) # 수행 시간 출력
```

파이썬 자료형

프로그래밍은 데이터를 다루는 행위이므로 자료형 이해가 필수이다.

정수형 Integer

양의 정수, 0, 음의 정수를 포함한다.

실수형 Real Number

변수에 소수점을 붙인 수를 대입하면 실수형 변수로 처리한다.
소수부, 정수부가 0인 경우 생략 가능하다.
ex)
-0.7 → -.7

5.0 → 5.

컴퓨터 시스템은 고정된 크기의 메모리를 할당하므로 실수 정보를 표현하는 정확도에 한계를 가져 미세한 오차가 발생한다.
따라서 실수값 비교 과정에서 원하는 결과를 얻지 못할 수도 있다.
ex) 파이썬에서 0.3 + 0.6을 더한 값은 0.9가 아니다.

이럴 때는 반올림 함수인 round() 함수를 사용하는 것이 권장된다.
int()를 통해 실수형을 정수형으로 변환할 수도 있다.

파이썬에서의 지수 표현 방식

파이썬에서는 e나 E를 이용한 지수 표현 방식을 이용할 수 있다.

$$aeb = a \times 10^b$$

a는 유효숫자, b는 지수이다. 실수로 처리되며, a와 b는 1이더라도 생략하면 안 된다.

```
a = 1e9 # 10의 9제곱
print(a)

# 출력
# 1000000000.0
```

무한(INF), 임의의 큰 수 설정이 필요할 때 사용하곤 한다.
ex) 최댓값이 10억 미만일 때 INF을 1e9로 매크로 설정하여 사용

수 자료형의 연산

나누기 연산자(/)로 나눠진 결과는 실수형으로 반환한다.
몫을 얻기 위해서는 몫 연산자(//)를 사용해야 한다.
파이썬에는 거듭 제곱 연산자(**)를 비롯한 다양한 연산자가 존재한다.

리스트 자료형

여러 개의 데이터를 연속적으로 담아 처리하기 위해 사용하는 자료형을 리스트라고 한다.
리스트 대신 배열, 테이블이라 부르기도 한다.

리스트 초기화

대괄호([])안에 원소를 쉼표(,)로 구분하여 초기화한다.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

비어 있는 리스트를 선언하고자 할 때는 list() 혹은 []을 이용한다.

```
a = list()
b = []
print(a)
print(b)

# 출력
# [ ]
# [ ]
```

이런 식으로 초기화할 수도 있다.

```
# 크기가 n이고, 모든 값이 0인 1차원 리스트 초기화
n = 10
a = [0] * n
print(a)

# 출력
# [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

리스트의 인덱싱과 슬라이싱

리스트의 원소에 접근할 때는 Index 값을 대괄호에 넣는다. **Index는 0부터 시작한다는 점에 주의한다.**
이처럼 인덱스 값을 입력하여 리스트의 특정한 원소에 접근하는 것을 인덱싱(Indexing)이라고 한다.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[0]) # 인덱싱

# 출력
# 1
```

파이썬에서 인덱스 값은 양의 정수, 음의 정수, 0 모두 사용 가능하다.
음의 정수를 넣으면 원소를 거꾸로 탐색한다.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(a[-3]) # 뒤에서 세 번째 원소 출력

# 출력
# 7
```

리스트에서 연속적인 위치를 갖는 원소들을 가져와야 할 때는 슬라이싱(Slicing)을 이용한다.
대괄호 안에 콜론(:)을 넣어서 시작 인덱스와 끝 인덱스를 설정할 수 있다.
끝 인덱스는 실제 인덱스보다 1 더 크게 설정한다.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 1 인덱스 원소부터 3 인덱스 원소까지 출력
print(a[1:4])

# 출력
# [2, 3, 4]
```

리스트 컴프리헨션

리스트 초기화 방법 중 하나로, 대괄호 안에 조건문과 반복문을 적용하여 리스트를 초기화하는 것을 말한다.

```
# 0부터 9까지의 수를 포함하는 리스트
array = [i for i in range(10)]

print(array)

# 출력
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

다양한 방식으로 응용할 수 있다.

```
# 0부터 19까지의 수 중에서 홀수만 포함하는 리스트
array = [i for i in range(20) if i % 2 == 1]

print(array)

# 1부터 9까지의 수들의 제곱 값을 포함하는 리스트
array = [i * i for i in range(1, 10)]

print(array)

'''
출력
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
'''
```

일반적인 코드보다 간결하게 작성할 수 있다.

```
# 0부터 19까지의 수 중에서 홀수만 포함하는 리스트

# 리스트 컴프리헨션
array = [i for i in range(20) if i % 2 == 1]

print(array)

# 일반적인 코드
array = []
for i in range(20):
    if i % 2 == 1:
        array.append(i)

print(array)

'''
출력
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
'''
```

라스트 컴프리헨션은 2차원 리스트를 초기화할 때 효과적으로 사용될 수 있다.

특히 n* m 크기의 2차원 리스트를 한 번에 초기화 해야 할 때 매우 유용하다.
단, 2차원 리스트를 초기화할 때 두 번째 방식으로 작성하면 **전체 리스트 안에 포함된 각 리스트가 모두 같은 객체로 인식**되므로 주의해야 한다.
인덱싱하여 요소 값을 바꿀 때 같은 객체인 다른 리스트의 요소 값도 바뀌어버릴 수 있다.

```
n = 4
m = 2

# n * m 크기의 2차원 리스트 초기화
array = [[0] * m for _ in range(n)]
print(array)
array[3][1] = 42
print(array)
'''
출력
[[0, 0], [0, 0], [0, 0], [0, 0]]
[[0, 0], [0, 0], [0, 0], [0, 42]]
'''

# n * m 크기의 2차원 리스트 초기화 (잘못된 방법)
array = [[0] * m] * n
print(array)
array[3][1] = 42
print(array)
'''
출력
[[0, 0], [0, 0], [0, 0], [0, 0]]
[[0, 42], [0, 42], [0, 42], [0, 42]]
'''
```

언더바

파이썬에서는 반복을 수행하되 반복을 위한 변수의 값을 무시하고자 할 때 언더바(_)를 자주 사용한다.

```
# 1부터 9까지의 자연수를 더하기
summary = 0
for i in range(1, 10):
  summary += i
print(summary)

# "Hello World"를 5번 출력하기
for _ in range (5):
  print("Hello World")

'''
출력
45
Hello World
Hello World
Hello World
Hello World
Hello World
'''
```

리스트 관련 기타 메서드

함수명	사용법	설명	시간 복잡도
append()	변수명.append()	리스트에 원소를 하나 삽입할 때 사용한다.	O(1)
sort()	변수명.sort()	기본 정렬 기능으로 오름차순으로 정렬한다.	O(NlogN)
	변수명.sort(reverse = True)	내림차순으로 정렬한다.	
reverse()	변수명.reverse()	리스트의 원소의 순서를 모두 뒤집어놓는다.	O(N)
insert()	insert(삽입할 위치 인덱스, 삽입할 값)	특정한 인덱스 위치에 원소를 삽입할 때 사용한다.	O(N)
count()	변수명.count(특정 값)	리스트에서 특정한 값을 가지는 데이터의 개수를 셀 때 사용한다.	O(N)
remove()	변수명.remove(특정 값)	특정한 값을 갖는 원소를 제거하는데, 값을 가진 원소가 여러 개면 하나만 제거한다.	O(N)

```
a = [1, 4, 3]
print("기본 리스트: ", a)

# 리스트에 원소 삽입
a.append(2)
print("삽입: ", a)

# 오름차순 정렬
a.sort()
print("오름차순 정렬: ", a)

# 내림차순 정렬
a.sort(reverse = True)
print("내림차순 정렬: ", a)

a = [4, 3, 2, 1]

# 리스트 원소 뒤집기
a.reverse()
print("원소 뒤집기: ", a)

# 특정 인덱스에 데이터 추가
a.insert(2, 3)
print("인덱스 2에 3 추가: ", a)

# 특정 값인 데이터 개수 세기
print("값이 3인 데이터 개수: ", a.count(3))

# 특정 값 데이터 삭제
a.remove(1)
print("값이 1인 데이터 삭제: ", a)

'''
출력
기본 리스트:  [1, 4, 3]
삽입:  [1, 4, 3, 2]
오름차순 정렬:  [1, 2, 3, 4]
내림차순 정렬:  [4, 3, 2, 1]
원소 뒤집기:  [1, 2, 3, 4]
인덱스 2에 3 추가:  [1, 2, 3, 3, 4]
값이 3인 데이터 개수:  2
값이 1인 데이터 삭제:  [2, 3, 3, 4]
'''
```

리스트에서 특정 값을 가지는 원소를 모두 제거하기

```
a = [1, 2, 3, 4, 5, 5, 5]
remove_set = {3, 5} # 집합 자료형

# remove_set에 포함되지 않은 값만을 저장
result = [i for i in a if i not in remove_set]
print(result)

''' 출력
[1, 2, 4]
'''
```

문자열 자료형

문자열 변수를 초기화할 때는 큰 따옴표(")나 작은 따옴표(')를 이용한다.
전체 문자열을 큰 따옴표로 구성하는 경우, 내부적으로 작은 따옴표를 포함할 수 있으며,
반대로 전체 문자열을 작은 따옴표로 구성하는 경우, 내부적으로 큰 따옴표를 포함할 수 있다.
백슬래시(₩)를 사용하면, 큰 따옴표나 작은 따옴표를 원하는 만큼 포함시킬 수 있다.
상단 백슬래시 문자가 원화 모양으로 보인다고 해서 이상한 것이 아니다. 글꼴에 따라 모양이 다르게 보인다고 한다.

```
data = 'Hello World'
print(data)

data = 'Don\'t you know "Python"?'
print(data)

data = "Don't you know \"Python\"?"
print(data)

''' 출력
Hello World
Don't you know "Python"?
Don't you know "Python"?
'''
```

문자열 연산

문자열 변수에 덧셈(+)을 이용하면 문자열이 더해져서 연결(Concatenate)된다.
문자열 변수를 특정한 양의 정수와 곱하는 경우, 문자열이 그 값만큼 여러 번 더해진다.
문자열도 리스트와 마찬가지로 인덱싱과 슬라이싱을 이용할 수 있다.

```
a = "Hello"
b = "World"
print(a + " " + b)

a = "String"
print(a * 3)

a = "ABCDEF"
print(a[2 : 4])

''' 출력
Hello World
StringStringString
CD
'''
```

단, 문자열은 특정 인덱스의 값을 변경할 수 없다. (Immutable)

```
a = "Hello"
a[1] = 'a'

''' 출력
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    a[1] = 'a'
TypeError: 'str' object does not support item assignment
'''
```

튜플 자료형

튜플 자료형은 리스트와 유사하지만 다음과 같은 문법적 차이가 있다.

- 튜플은 한 번 선언된 값을 변경할 수 없다.
- 리스트는 대괄호([])를 이용하지만, 튜플은 소괄호(())를 이용한다.

튜플은 리스트에 비해 기능이 제한적이고, 따라서 상대적으로 공간 효율적이다.

```
a = (1, 2, 3, 4, 5, 6, 7, 8, 9)

# 네 번째 원소만 출력
print(a[3])

# 두 번째 원소부터 네 번째 원소까지
print(a[1 : 4])

# 오류 발생(문자열과 동일한 TypeError)
a[2] = 7

''' 출력
4
(2, 3, 4)
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    a[2] = 7
TypeError: 'tuple' object does not support item assignment
'''
```

튜플을 사용하면 좋은 경우

- 서로 다른 성질의 데이터를 묶어서 관리해야 할 때
ex) 최단 경로 알고리즘에서는 (비용, 노드 번호)의 형태로 튜플 자료형을 자주 사용한다.
- 데이터의 나열을 해싱(Hashing)의 키 값으로 사용해야 할 때, 튜플은 변경이 불가능하므로 리스트와 다르게 키 값으로 사용될 수 있다.
- 리스트보다 메모리를 효율적으로 사용해야 할 때

사전(Dictionary) 자료형

사전 자료형은 키(Key)와 값(Value)의 쌍을 데이터로 가지는 자료형으로, 앞서 다루었던 리스트나 튜플이 값을 순차적으로 저장하는 것과는 대비된다.
사전 자료형은 키와 값의 쌍을 데이터로 가지며, 원하는 '변경 불가능한 자료형'을 키로 사용할 수 있다.
파이썬의 사전 자료형은 해시 테이블(Hash Table)을 이용하므로 데이터의 조회 및 수정에 있어서 O(1)의 시간에 처리할 수 있다.

```
data = dict()
data['사과'] = 'Apple'
data['바나나'] = 'Banana'
data['코코넛'] = 'Coconut'

print(data)

if '사과' in data:
  print("'사과'를 키로 가지는 데이터가 존재합니다.")

''' 출력
{'사과': 'Apple', '바나나': 'Banana', '코코넛': 'Coconut'}
'사과'를 키로 가지는 데이터가 존재합니다.
'''
```

사전 자료형 관련 메서드

사전 자료형에서는 키와 값을 별도로 뽑아내기 위한 메서드를 지원한다.
키 데이터만 뽑아서 리스트로 이용할 때는 keys() 함수를 이용하고,
값 데이터만 뽑아서 리스트로 이용할 때는 values() 함수를 이용한다.

```
data = dict()
data['사과'] = 'Apple'
data['바나나'] = 'Banana'
data['코코넛'] = 'Coconut'

# 키 데이터만 담은 리스트
key_list = data.keys()
# 값 데이터만 담은 리스트
value_list = data.values()

print(key_list)
print(value_list)

# 각 키에 따른 값을 하나씩 출력
for key in key_list:
  print(data[key])

''' 출력
dict_keys(['사과', '바나나', '코코넛'])
dict_values(['Apple', 'Banana', 'Coconut'])
Apple
Banana
Coconut
'''
```

사전 자료형의 선언 및 초기화는 아래와 같은 방식으로도 가능하다.
keys(), values()가 리턴한 dict_keys(), dict_values()를 제대로 리스트로서 사용하고 싶다면 list()를 이용해 형변환을 해주어야 한다.

```
data = {
  '사과': 'Apple',
  '바나나': 'Banana',
  '코코넛': 'Coconut'
}

# 키 데이터만 담은 리스트
key_list = list(data.keys())
# 값 데이터만 담은 리스트
value_list = list(data.values())

print(key_list)
print(value_list)

''' 출력
['사과', '바나나', '코코넛']
['Apple', 'Banana', 'Coconut']
'''
```

집합 자료형

집합은 중복을 허용하지 않으며, 순서가 없다는 특징이 있다.
따라서 중복되는 요소가 있으면 하나만을 저장하며, 순서가 없기 때문에 리스트처럼 인덱스 번호를 사용하여 특정 요소에 접근할 수 없다.
집합은 리스트 혹은 문자열을 이용해서 초기화할 수 있다. 이 때, set() 함수를 이용한다.
혹은 중괄호({}) 안에 각 원소를 쉼표(,)를 기준으로 구분하여 삽입함으로써 초기화 할 수 있다.
데이터의 조회 및 수정에 있어서 O(1)의 시간에 처리할 수 있다.

```
# 집합 자료형 초기화 방법 1
data = set([1, 1, 2, 3, 4, 4, 5])
print(data)

# 집합 자료형 초기화 방법 2
data = {1, 1, 2, 3, 4, 4, 5}
print(data)

''' 출력
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
'''
```

집합 자료형의 연산

기본적인 집합 연산으로는 합집합, 교집합, 차집합 연산 등이 있다.
논리 연산자를 생각하면 어렵지 않다.

```
a = set([1, 2, 3, 4, 5])
b = set([3, 4, 5, 6, 7])

# 합집합
print(a | b)

# 교집합
print(a & b)

# 차집합
print(a - b)

''' 출력
{1, 2, 3, 4, 5, 6, 7}
{3, 4, 5}
{1, 2}
'''
```

집합 자료형 관련 함수

add(), update(), remove()가 있다.

```
data = set([1, 2, 3])
print(data)

# 새로운 원소 추가
data.add(4)
print(data)

# 새로운 원소 여러 개 추가
data.update([5, 6])
print(data)

# 특정한 값을 갖는 원소 삭제
data.remove(3)
print(data)

''' 출력
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6}
'''
```

사전 자료형과 집합 자료형의 특징

리스트나 튜플은 순서가 있기 때문에 인덱싱을 통해 자료형의 값을 얻을 수 있으나,
사전 자료형과 집합 자료형은 순서가 없기 때문에 인덱싱으로 값을 얻을 수 없다.
대신, 사전의 키(Key) 또는 집합의 원소(Element)를 이용해 O(1)의 시간 복잡도로 조회한다.

기본 입출력

모든 프로그램은 적절한 (약속된) 입출력 양식을 가지고 있다.
프로그램 동작의 첫 번째 단계는 데이터를 입력 받거나 생성하는 것이다.

자주 사용되는 표준 입력 방법

input() 함수는 한 줄의 문자열을 입력 받는 함수이다.
map() 함수는 리스트의 모든 원소에 각각 특정한 함수를 적용할 때 사용한다.
공백을 기준으로 구분된 데이터를 입력 받을 때는 다음과 같이 사용한다.

```
list(map(int, input().split()))
```

공백을 기준으로 구분된 데이터의 개수가 많지 않다면, 단순히 다음과 같이 사용한다.

```
a, b, c = map(int, input().split())
```

사용 예시는 아래와 같다.
데이터의 개수와 데이터를 입력받은 후, 내림차순으로 정렬하는 소스 코드이다.

```
# 데이터의 개수 입력
n = int(input())
# 각 데이터를 공백을 기준으로 구분하여 입력
data = list(map(int, input().split()))

data.sort(reverse=True)
print(data)
```

빠르게 입력 받기

사용자로부터 입력을 최대한 빠르게 받아야 하는 경우가 있다(시간 초과 등).
파이썬의 경우 sys 라이브러리에 정의되어 있는 sys.stdin.readline() 메서드를 이용한다.
단, 입력 후 엔터(Enter)가 줄 바꿈 기호로 입력되므로 rstrip() 메서드를 함께 사용한다.

```
import sys

# 문자열 입력 받기
data = sys.stdin.readline().rstrip()
print(data)
```

자주 사용되는 표준 출력 방법

파이썬에서 기본 출력은 print() 함수를 이용한다.
각 변수를 쉼표를 이용하여 띄어쓰기로 구분하여 출력할 수 있다.
print()는 기본적으로 출력 이후에 줄 바꿈을 수행한다.
줄 바꿈을 원치 않는 경우 'end' 속성을 이용할 수 있다.

```
# 출력할 변수들
a = 1
b = 2
print(a, b)
print(7, end=' ')
print(8, end=' ')

# 출력할 변수
answer = 7
print("정답은 " + str(answer) + "입니다.")

''' 출력
1 2
7 8 정답은 7입니다.
'''
```

f-string 예제

파이썬 3.6부터 사용 가능하며, 문자열 앞에 접두사 'f'를 붙여 사용한다.
중괄호 안에 변수명을 기입하여 간단히 문자열과 정수를 함께 넣을 수 있다.

```
answer = 7
print(f"정답은 {answer}입니다.")

''' 출력
정답은 7입니다.
'''
```

들여쓰기

파이썬에서는 코드의 블록(Block)을 들여쓰기(Indent)로 지정한다.
탭을 사용하는 쪽과 공백문자(space)를 여러 번 사용하는 쪽으로 두 진영이 있다.
이에 대한 논쟁은 지금까지도 활발하다.
파이썬 스타일 가이드라인에서는 **4개의 공백 문자를 사용하는 것**을 표준으로 설정하고 있다.

조건문의 기본 형태

조건문의 기본적인 형태는 if ~ elif ~ else이다.
조건문을 사용할 때 elif 혹은 else 부분은 경우에 따라서 사용하지 않아도 된다.

비교 연산자

비교 연산자는 특정한 두 값을 비교할 때 이용할 수 있다.
대표적인 비교 연산자로는 ==, !=, >, <, >=, <= 여섯 가지가 있다.

논리 연산자

논리 연산자는 논리 값 (True/False) 사이의 연산을 수행할 때 사용한다.
논리 연산자로는 and, or, not이 있다.
보통 다른 프로그래밍 언어에서는 기호를 사용하는데, 파이썬은 영단어 그대로 사용하여 직관적이다.

파이썬의 기타 연산자

다수의 데이터를 담는 자료형을 위해 in 연산자와 not in 연산자가 제공된다.
리스트, 튜플, 문자열, 딕셔너리 모두에서 사용이 가능하다.

파이썬의 pass 키워드

아무것도 처리하고 싶지 않을 때 pass 키워드를 사용한다.
디버깅 과정에서 일단 조건문의 형태만 만들어 놓고 조건문을 처리하는 부분은 비워놓고 싶은 경우 사용할 수 있다.

```
score = 85

if score >= 80:
    pass # 나중에 작성할 소스 코드
else:
    print('성적이 80점 미만입니다.')

print('프로그램을 종료합니다.')

''' 출력
프로그램을 종료합니다.
'''
```

조건문의 간소화

조건문에서 실행될 소스코드가 한 줄인 경우, 굳이 줄 바꿈을 하지 않고도 간략하게 표현할 수 있다.

```
score = 85

if score >= 80: result = "Success"
else: result = "Fail"

print(result)

''' 출력
Success
'''
```

조건부 표현식(Conditional Expression)은 if ~ else문을 한 줄에 작성할 수 있도록 해준다.
삼항연산자 느낌이다. 왼쪽과 가운데가 바뀐 느낌.

```
score = 85
result = "Success" if score >= 80 else "Fail"

print(result)

''' 출력
Success
'''
```

파이썬 조건문 내에서의 부등식

다른 프로그래밍 언어와 다르게 파이썬은 조건문 안에서 수학의 부등식을 그대로 사용할 수 있다.
예를 들어 x > a and x < 20과 0 < x < 20은 같은 결과를 반환한다.

반복문

파이썬에는 while문과 for문이 있는데, 어떤 것을 사용해도 상관 없다.

다만 코딩 테스트에서의 실제 사용 예시를 확인해 보면, for문이 더 간결한 경우가 많다.

반복문에서의 무한 루프

무한 루프(Infinite Loop)란 끊임없이 반복되는 반복 구문을 의미한다.
코딩 테스트에서 무한 루프를 구현할 일은 거의 없으니 유의해야 한다.
반복문을 작성한 뒤에는 항상 반복문을 탈출할 수 있는지 확인한다.

for문

특정한 변수를 이용하여 in 뒤에 오는 **데이터(리스트, 튜플 등)에 포함되어 있는 원소를 첫 번째 인덱스부터 차례대로 하나씩 방문**한다.

```
tuple = (9, 8, 7, 6, 5)

for x in tuple:
    print(x)

''' 출력
9
8
7
6
5
'''
```

for문에서 연속적인 값을 차례로 순회할 때는 range()를 주로 사용한다.
이 때 range(시작 값, 끝 값 + 1) 형태로 사용한다.
인자를 하나만 넣으면 끝 값으로 인식하며, 자동으로 시작 값은 0이 된다.

```
result = 0

# i는 0부터 9까지의 모든 값을 순회
for i in range(10):
    result += i

print(result)

''' 출력
45
'''
```

파이썬의 continue 키워드

반복문에서 남은 코드의 실행을 건너뛰고, 다음 반복을 진행하고자 할 때 continue를 사용한다.
1부터 9까지의 홀수의 합을 구할 때 다음과 같이 작성할 수 있다.

```
result = 0

for i in range(1, 10):
    if i % 2 == 0:
        continue
    result += i

print(result)

''' 출력
25
'''
```

파이썬의 break 키워드

반복문을 즉시 탈출하고자 할 때 break을 사용한다.

함수

함수(Function)란 특정한 작업을 하나의 단위로 묶어 놓은 것을 의미한다.
함수를 사용하면 불필요한 소스 코드의 반복을 줄일 수 있다.

함수의 종류

내장 함수: 파이썬이 기본적으로 제공하는 함수
사용자 정의 함수: 개발자가 직접 정의하여 사용할 수 있는 함수

함수의 정의

C와 비교해보면 파이썬은 함수 앞에 자료형 대신 def가 들어간다.

파라미터 지정하기

파이썬에서는 파라미터의 변수를 직접 지정할 수 있다.
이 경우 매개변수의 순서가 달라도 상관이 없다.

```
def add(a, b):
    print(a, '+', b, '=', a + b)

add(b = 3, a = 7)

''' 출력
7 + 3 = 10
'''
```

global 키워드

global 키워드로 변수를 지정하면 해당 함수에서는 지역 변수를 만들지 않고, **함수 바깥에 선언된 변수를 바로 참조**하게 된다.
단, 함수 안에서 단순히 값을 참조하는 경우(변수 출력 등) global 키워드가 없어도 된다.
같은 이름의 지역 변수와 전역 변수가 있는 경우 함수 내부에서는 지역변수를 우선 참조한다.
다만 코딩 테스트에서 이것이 중요한 문제가 되는 경우는 거의 없을 것이다. 그래도 잘 알아두자.

```
a = 0

def func():
    global a
    a += 1

def func2():
    print(a)

for _ in range(10):
    func()

print(a)
func2()

''' 출력
10
10
'''
```

여러 개의 반환 값

파이썬에서 함수는 여러 개의 반환 값을 가질 수 있다.
여러 개의 변수가 한 번에 반환되는 것을 패킹이라 하고, 반환된 값을 변수에 대입하는 것을 언패킹이라고 한다.

```
def operator(a, b):
    add = a + b
    subtract = a - b
    multiply = a * b
    divide = a / b
    return add, subtract, multiply, divide

a, b, c, d = operator(b = 3, a = 7)
print(a, b, c, d)

''' 출력
10 4 21 2.3333333333333335
'''
```

람다 표현식

람다 표현식을 이용하면 함수를 간단하게 작성할 수 있다.
특정한 기능을 수행하는 함수를 한 줄에 작성할 수 있다는 점이 특징이다.
함수의 기능이 간단하고 재사용 할 것 같지 않을 때, 함수 자체를 입력으로 받는 또 다른 함수가 존재할 때 유용하게 쓰일 수 있다.

람다 표현식 예시

```
def add(a, b):
    return a + b

# 일반적인 add() 메서드 사용
print(add(3, 7))

# 람다 표현식으로 구현한 add() 메서드
print((lambda a, b: a + b)(3, 7))

''' 출력
10
10
'''
```

내장 함수에서 자주 사용되는 람다 함수

```
array = [("홍길동", 50), ("이순신", 32), ("아무개", 74)]

def my_key(x):
    return x[1]

print(sorted(array, key=my_key))
print(sorted(array, key=lambda x: x[1]))

''' 출력
[('이순신', 32), ('홍길동', 50), ('아무개', 74)]
[('이순신', 32), ('홍길동', 50), ('아무개', 74)]
'''
```

여러 개의 리스트에 적용

```
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8, 9, 10]

result = map(lambda a, b: a + b, list1, list2)

print(list(result))

''' 출력
[7, 9, 11, 13, 15]
'''
```

실전에서 유용한 표준 라이브러리

내장 함수	기본 입출력 함수부터 정렬 함수까지 기본적인 함수들을 제공합니다. 파이썬 프로그램을 작성할 때 없어서는 안 되는 필수적인 기능을 포함하고 있습니다.
itertools	파이썬에서 반복되는 형태의 데이터를 처리하기 위한 유용한 기능들을 제공합니다. 특히 순열과 조합 라이브러리는 코딩 테스트에서 자주 사용됩니다.
heapq	힙(Heap) 자료구조를 제공합니다. 일반적으로 우선순위 큐 기능을 구현하기 위해 사용됩니다.

bisect	이진 탐색(Binary Search) 기능을 제공합니다.
collections	덱(deque), 카운터(Counter) 등의 유용한 자료구조를 포함합니다.
math	필수적인 수학적 기능을 제공합니다. 팩토리얼, 제곱근, 최대공약수(GCD), 삼각함수 관련 함수부터 파이(pi)와 같은 상수를 포함합니다.

자주 사용되는 내장 함수

```
# sum()
result = sum([1, 2, 3, 4, 5])
print(result)

# min(), max()
min_result = min(7, 3, 5, 2)
max_result = max(7, 3, 5, 2)
print(min_result, max_result)

# eval()
result = eval("(3+5)*7")
print(result)

# sorted()
result = sorted([9, 1, 8, 5, 4])
reverse_result = sorted([9, 1, 8, 5, 4], reverse=True)
print(result)
print(reverse_result)

# sorted() with key
array = [('홍길동', 35), ('이순신', 75), ('아무개', 50)]
print(sorted(array, key=lambda x: x[1], reverse=True))

''' 출력
15
2 7
56
[1, 4, 5, 8, 9]
[9, 8, 5, 4, 1]
[('이순신', 75), ('아무개', 50), ('홍길동', 35)]
'''
```

순열과 조합

itertools 라이브러리를 이용할 수 있다.

순열(Permutation)
서로 다른 n개에서 서로 다른 r개를 선택하여 일렬로 나열하는 것

```
from itertools import permutations

data = ['A', 'B', 'C'] # 데이터 준비

result = list(permutations(data, 3)) # 모든 순열 구하기
print(result)

''' 출력
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A', 'B'), ('
'''
```

조합(Combination)
서로 다른 n개에서 순서에 상관 없이 서로 다른 r개를 선택하는 것

```
from itertools import combinations

data = ['A', 'B', 'C'] # 데이터 준비

result = list(combinations(data, 2)) # 2개를 뽑는 모든 조합 구하기
print(result)

''' 출력
[('A', 'B'), ('A', 'C'), ('B', 'C')]
'''
```

중복 순열과 중복 조합

```
from itertools import product, combinations_with_replacement

data = ['A', 'B', 'C'] # 데이터 준비

result = list(product(data, repeat=2)) # 2개를 뽑는 모든 순열 구하기 (중복 허용)
print(result)

print()

result = list(combinations_with_replacement(data, 2)) # 2개를 뽑는 모든 조합 구하기 (중복
print(result)

''' 출력
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'), ('C', 'A'), ('C

[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
'''
```

Counter

파이썬 collections 라이브러리의 Counter는 등장 횟수를 세는 기능을 제공한다.
리스트와 같은 반복 가능한(iterable) 객체가 주어졌을 때 내부의 원소가 몇 번씩 등장했는지를 알려준다.

예전에 해당 메소드를 이용해 워드 카운트 웹페이지를 만들어서 제출하는 멋사 과제가 있었던 것 같다.
그 때 제대로 코딩할 줄도 몰라서 과제 제출 마감일 앞두고 그냥 멘토 형 코드를 후다닥 복붙해서 냈던 기억이 난다.
시간 되면 강의 찾아서 처음부터 따라해봐야겠다. 안 보고 어설프게나마 따라할 수 있을 것 같기도 하고...?

```
from collections import Counter

counter = Counter(['red', 'blue','red','green','blue','blue'])

print(counter['blue']) # 'blue'가 등장한 횟수 출력
print(counter['green']) # 'green'이 등장한 횟수 출력
print(dict(counter)) # 사전 자료형으로 변환

''' 출력
3
1
{'red': 2, 'blue': 3, 'green': 1}
'''
```

최대 공약수와 최소 공배수

최대 공약수를 구해야 할 때는 math 라이브러리의 gcd() 함수를 이용할 수 있다.

어떻게 최대 공약수를 가지고 최소 공배수를 구하나 잠시 고민해봤는데, 생각보다 간단하게 이해가 됐다.
두 수는 각각 최대 공약수 * n, 최대 공약수 * m의 형태로 이루어져 있을 것인데, 최소 공배수는 최대 공약수 * n * m일테니
두 수를 곱한 값에서 최대 공약수를 나눈 몫(값)이 바로 최소 공배수가 되는 것이다.
정확히 설명이 잘 안 되긴 하는데 인터넷으로 찾아보니 내가 이해한 게 맞는 것 같아서 일단 이 정도로만 정리하고 넘어간다.

```
import math

# 최소 공배수(LCM)를 구하는 함수
def lcm(a, b):
    return a * b // math.gcd(a , b)

a = 21
b = 14

print(math.gcd(21, 14)) # 최대 공약수(GCD) 계산
print(lcm(21, 14)) # 최소 공배수(LCM) 계산
''' 출력
7
42
'''
```

이제보니 네이버 블로그 소스 코드가 파이썬에 맞지 않게 하이라이트가 되고 있다.
언어 선택 기능 지원이 안 되나...
그나저나 2시간 반 안 되는 강의를 너무 오래 보고 정리했다. 그래도 뿌듯하다!
어차피 뭉텅이로 정리한 거, 나중에 이 글이 치트 시트처럼 읽혔으면 좋겠다.