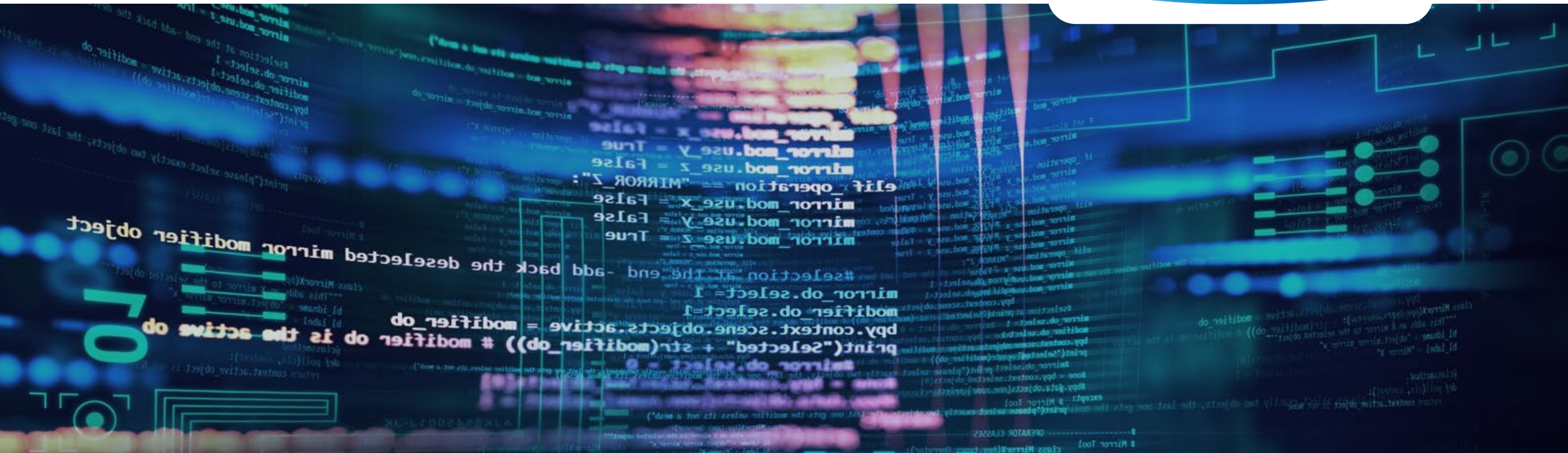
Lecture 7

리스트와 튜플

KAIST



파이썬의 자료형

- 파이썬에는 다양한 **빌트인(built-in: 내장) 자료형**들이 있음을 배웠다
- 파이썬의 수많은 자료형 중에서도 자주 사용되는 자료형들을 선정하면 다음과 같다
  - (부울) bool
  - (숫자) int, float, complex
  - (문자열) str
  - (시퀀스) str, **list, tuple**
  - (매핑) dict
  - (세트) set
- 이번 차시에서는 그 중에서도 **리스트(list)와 튜플(tuple)** 자료형에 대해 배워보자

리스트(list)

KAIST

- 리스트 자료형은 자유도와 활용도가 매우 높아 파이썬 개발자들에게 자주 사용되는 자료형이다
- 리스트 자료형이 갖고있는 **네 가지 큰 특징**은 다음과 같다
  - **반복 가능(iterable)**
  - **원소 자료형 무관**
  - **변경 가능(mutable)**
  - **유익한 메소드**
- 위의 특징들을 다음 슬라이드에서부터 하나씩 살펴보자

리스트 - 반복 가능(iterable)

KAIST

- 리스트는 반복 가능한(iterable) 자료형이다
- 수학에서 배운 배열(array)을 일반화한 것으로 간주하면 이해하기 쉽다

[예제 1]

```
1) my_first_list = [0, 1, 2, 3, 4, 5]
2)  print(type(my_first_list))
3)  print(len(my_first_list))
4)  for item in my_first_list:
5)      print(item, end=" ")
```

```
<class 'list'>
6
0 1 2 3 4 5
```

- 리스트는 **대괄호[]로 원소(요소)를 감싸**으로써 표현하며 원소간의 구분은 **콤마(,)**로 한다
- 리스트와 같은 iterable 자료형들은 내부 각 원소에 대한 접근 및 전달이 가능한 것이 특징이다

잠깐!
print 함수의 end옵션은 출력을 완료한 뒤 어떠한 출력을 추가로 할 것인가를 입력받는다. (디폴트 값은 '\n')

리스트 - 반복 가능(iterable)

KAIST

- 리스트와 같은 iterable 자료형들은 내부 각 원소에 대한 접근 및 전달이 가능한 것이 특징이다
- 인덱싱과 슬라이싱은 iterable 객체 내부 원소(들)에 대한 접근 방식을 의미한다
- **인덱싱(indexing): 원소 한 개에 대한 접근**

[예제 2]

```
1) my_first_list = [0, 1, 2, 3, 4, 5]
2) print(my_first_list[0]) # 맨 처음 원소 인덱싱
3) print(my_first_list[-1]) # 마지막 원소 인덱싱
4) print(my_first_list[-3]) # 뒤에서 세번째 원소 인덱싱
```

```
0
5
3
```

잠깐!
파이썬은 0-based 넘버링을 취한다. (첫번째 원소의 인덱스는 1이 아니라 0이다!)
오해의 소지를 줄이고자 지금부터 가장 처음에 나타나는 원소(인덱스 0)를 0번째라고 서술하겠다.

```
5) print(my_first_list[6]) # 인덱싱 에러 (범위 밖)
```

IndexError: list index out of range

- 이처럼 범위 밖의 원소에 접근하고자 할 때는 IndexError가 발생한다
(객체에 원소가 n개 있을 때, **-n부터 n-1까지**가 유효한 index range)

리스트 - 반복 가능(iterable)

KAIST

- 리스트와 같은 iterable 자료형들은 내부 각 원소에 대한 접근 및 전달이 가능한 것이 특징이다
- 인덱싱과 슬라이싱은 iterable 객체 내부 원소(들)에 대한 접근 방식을 의미한다
- **슬라이싱(slicing): 원소 N개에 대한 접근**

[예제 3]

```
1) my_first_list = [0, 1, 2, 3, 4, 5]
2) print(my_first_list[0:3]) # 0 이상 3 미만 인덱스의 원소들 슬라이싱
3) print(type(my_first_list[0:3])) # 슬라이싱의 결과는 리스트
4) print(my_first_list[2:3]) # 2 이상 3 미만 (즉, 두번째 원소 only)
5) print(type(my_first_list[2:3])) # 슬라이싱의 결과는 리스트
6) print(my_first_list[:])     # 전체 원소
7) print(type(my_first_list[:])) # 슬라이싱의 결과는 리스트
```

```
[0, 1, 2]
<class 'list'>
[2]
<class 'list'>
[0, 1, 2, 3, 4, 5]
<class 'list'>
```

- 슬라이싱은 **콜론(:)**을 사용하여 접근하고자 하는 원소들의 구간을 설정한다
- 구간은 **단일구간**이며 **단방향**으로 지정할 수 있다

리스트 - 반복 가능(iterable)

KAIST

- 리스트와 같은 iterable 자료형들은 내부 각 원소에 대한 접근 및 전달이 가능한 것이 특징이다
- 인덱싱과 슬라이싱은 iterable 객체 내부 원소(들)에 대한 접근 방식을 의미한다
- **슬라이싱(slicing): 원소 N개에 대한 접근**

[예제 4]

```
1) my_first_list = [0, 1, 2, 3, 4, 5]
2) print(my_first_list[2:5])     # 전체 원소
3) print(my_first_list[2:5:1])  # 위와 동치 (하나씩 건너뛰며 순회)
4) print(my_first_list[2:5:2])
```

→

```
[2, 3, 4]
[2, 3, 4]
[2, 4]
```

- 사실 온전한 슬라이싱은 **두 개의 콜론**을 사용하여 표현한다
- **[start:end:step]**형태에서 start은 슬라이싱 시작 인덱스, end는 슬라이싱 마지막+1 인덱스, step은 건너뛰는 크기를 의미한다 (양수는 오른쪽 방향, 음수는 왼쪽 방향)

```
5) print(my_first_list[5:2])      # 5 이상 2 미만에 걸치는 구간 없음
6) print(my_first_list[5:2:-1])  # step을 음수로 표현하면 해결
7) print(my_first_list[-1::-2])   # 맨 뒤부터 처음까지 2칸씩 건너기
```

→

```
[]
[5, 4, 3]
[5, 3, 1]
```

리스트 - 원소 자료형 무관

- 리스트 내부 원소들의 자료형은 일관되지 않아도 된다
- 즉, 원소가 어떠한 자료형을 가지는지 신경을 쓰지 않고 자유자재로 다룰 수 있다

[예제 5]

```
1  my_second_list = ["a", None, 3, 4.01, [5]]
2  print(my_second_list)
3  for value in my_second_list:
4      print(value, end=" ")
5  print(my_second_list[-1])
```

```
['a', None, 3, 4.01, [5]]
a None 3 4.01 [5] [5]
```

리스트 - 변경 가능(mutable)

KAIST

- 파이썬에서 객체는 변경이 가능한(mutable) 자료형과 아닌(immutable) 것이 있다
- list, dict, set의 경우는 mutable 자료형에 속하고 나머지는 immutable하다고 생각하면 된다
- mutable 객체의 장점은 내부 원소의 수정과 삭제가 자유롭다는 것이다

[예제 6]

```
1  my_first_list = [0, 1, 2, 3, 4, 5]
2  print(my_first_list)
3  my_first_list[0] = "Start" # 0번째 원소 재할당
4  print(my_first_list) # 수정된 리스트 출력
5  my_first_list[3:] = "0" # 3번째 이후 구간은 “0” 원소 하나로 재할당
6  print(my_first_list)
7  del(my_first_list[0]) # 빌트인 함수 del을 활용한 0번째 원소 제거
8  print(my_first_list)
```

```
[0, 1, 2, 3, 4, 5]
['Start', 1, 2, 3, 4, 5]
['Start', 1, 2, '0']
[1, 2, '0']
```

리스트 - 유익한 메소드 제공

KAIST

- 리스트에는 유익한 메소드(클래스 함수)를 다양하게 제공하고 있다
- 이들을 적재적소로 잘 활용하는 것이 개발자로서는 무척 중요하다
- 무엇이든 개발 도중에 궁금한 것이 발생하면 빌트인 함수인 help를 사용하거나 구글링을 하자

[예제 8]

```
1  help(list) # 이렇게 자료형 자체를 검색할 수도 있고
2  my_first_list = [0, 1, 2, 3, 4, 5]
3  help(my_first_list) # 변수명을 통해 검색할 수 있다
```

- 본 수업에서 다뤄볼 메소드들은 총 아홉가지이며 모두 빈번히 사용되므로 사용법을 잘 숙지하자

- **append**
- **pop**
- **count**
- **index**
- **remove**
- **sort**
- **reverse**
- **insert**
- **extend**

리스트 - 유익한 메소드 제공

KAIST

- **append** : 특정 값을 리스트의 마지막 자리에 원소로써 추가한다
- **pop** : 리스트의 마지막 원소를 추출 및 반환한다
- **count** : 리스트 내에 특정 값이 얼마나 존재하는지 빈도수를 반환한다

[예제 9]

```
1) hobby = ["Book", "Movie", "Bike", "Golf"]
2) print(f"hobby: {hobby}")
3) hobby.append("Travel") # 마지막 자리 원소 추가
4) print(f"hobby(1): {hobby}")
5) print(hobby.pop()) # 마지막 원소 추출 및 반환
6) print(f"hobby(2): {hobby}")
7) hobby.append("Book") # "Book" 중복 추가
8) print(f"hobby(3): {hobby}")
9) print(f"How many 'Book'?: {hobby.count('Book')}") # "Book" 원소 카운트
```

```
hobby: ['Book', 'Movie', 'Bike', 'Golf']
hobby(1): ['Book', 'Movie', 'Bike', 'Golf', 'Travel']
Travel
hobby(2): ['Book', 'Movie', 'Bike', 'Golf']
hobby(3): ['Book', 'Movie', 'Bike', 'Golf', 'Book']
How many 'Book'?: 2
```

잠깐!
f-string 포맷팅은 파이썬 3.6버전부터 지원하는 문자열 포맷팅 방법이다.
문자열 맨 앞에 f를 붙이고 문자열 중간에 출력하고 싶은 변수의 값이 있다면 중괄호 안에 변수명을 넣어준다

리스트 - 유익한 메소드 제공

- **index** : 특정 값이 존재하는 인덱스를 반환 (여러개가 있다면 가장 낮은 인덱스를 반환)
- **remove** : 리스트에서 특정 값의 원소를 제거 (여러개가 있다면 가장 낮은 인덱스의 값을 제거)
- **sort** : 리스트의 원소들을 정렬 (ascending order가 default; reverse 옵션으로 조절 가능)
- **reverse** : 리스트의 원소들을 마지막 인덱스에서부터 거꾸로 정렬

```
10)  print(f"Where is the first 'Book'?: {hobby.index('Book')}")
       # "Book"이 존재하는 가장 낮은 인덱스(위치)를 반환
11)  hobby.remove("Book")
12)  print(f"hobby(4): {hobby}")
13)  hobby.sort()
14)  print(f"hobby(5): {hobby}")
15)  hobby.reverse()
16)  print(f"hobby(6): {hobby}")
17)  hobby.sort(reverse=True)
18)  print(f"hobby(7): {hobby}")
```

```
hobby(3): ['Book', 'Movie', 'Bike', 'Golf', 'Book']

Where is the first 'Book'?: 0
hobby(4): ['Movie', 'Bike', 'Golf', 'Book']
hobby(5): ['Bike', 'Book', 'Golf', 'Movie']
hobby(6): ['Movie', 'Golf', 'Book', 'Bike']
hobby(7): ['Movie', 'Golf', 'Book', 'Bike']
```

리스트 - 유익한 메소드 제공

- **insert** : 리스트의 특정 인덱스에 특정 값을 삽입
- **extend** : 리스트 뒤에 또 다른 리스트를 연장 (하나의 리스트로 존재; A.extend(B) -> A가 B까지 흡수)

```
19) hobby.insert(2, "Jogging")
    # 두번째 자리에 "Jogging" 원소를 삽입
20) print(f"hobby(8): {hobby}")
21) hobby.extend(["Singing", "Dancing"])
    # hobby 리스트 뒤에 해당 리스트를 연장
22) print(f"hobby(9): {hobby}")
```

```
hobby(7): ['Movie', 'Golf', 'Book', 'Bike']

hobby(8): ['Movie', 'Golf', 'Jogging', 'Book', 'Bike']
hobby(9): ['Movie', 'Golf', 'Jogging', 'Book', 'Bike',
'Singing', 'Dancing']
```

리스트 – 그 외

- 리스트에는 **+** 과 *** 연산자**를 지원한다
- **+**는 리스트의 **extend 메소드** 역할과 동일하다
- *****은 "리스트 자기 자신을 몇 번 반복하여 연장할 것인가"를 수행하는 연산자이다

[예제 10]

```
1) print(["Hi"] + ["There"])
2) print(["Hi"] * 3)
```

→

```
['Hi', 'There']
['Hi', 'Hi', 'Hi']
```

- 추가적으로, 리스트와 문자열 간의 형 변환은 꽤 빈번히 활용되므로
  문자열의 메소드이지만 리스트와 연관되어 있는 join과 split 메소드에 대해 미리 알아보자

[예제 11]

```
1) hobby_list = ["Book", "Movie", "Bike", "Golf"]
2) hobby_str = ", ".join(hobby_list) # list -> str
3) print(hobby_str)
4) new_hobby_list = hobby_str.split(", ") # str -> list
5) print(new_hobby_list)
```

→

```
Book, Movie, Bike, Golf
['Book', 'Movie', 'Bike', 'Golf']
```

리스트 – 그 외

KAIST

특히 split 메소드의 경우에는 구분자(separator)를 기준으로
문자열을 분리하는데 default 값은 공백 문자 혹은 시퀀스이다

[예제 12]

```
1) hobby_list = ["Book", "Movie", "Bike", "Golf"]
2) hobby_str = " ".join(hobby_list) # 공백 문자
3) hobby_str2 = "    ".join(hobby_list) # 공백 시퀀스 (1)
4) hobby_str3 = "   \n\t ".join(hobby_list) # 공백 시퀀스 (2)
5) print(hobby_str == hobby_str2 == hobby_str3)
6) new_list = hobby_str.split()
7) new_list2 = hobby_str2.split()
8) new_list3 = hobby_str3.split()
9) print(new_list == new_list2 == new_list3)
```

```
False
True
```

튜플(tuple)

KAIST

- 튜플은 리스트와 유사하지만 자유도가 다소 떨어지는 자료형이다
- 바로 **내부 원소의 수정, 삭제가 불가능**하다는 점 때문이다
- 튜플 자료형이 갖고있는 **네 가지 큰 특징**은 다음과 같다
  - 반복 가능(iterable)
  - 원소 자료형 무관
  - **변경 불가능(immutable)**
  - 유익한 메소드
- 튜플은 **소괄호()로 원소를 감쌈**으로써 표현하며 원소간의 구분은 **콤마(,)**로 한다
- 튜플이 **단일 원소**일 경우 **콤마를 꼭 찍어줘야** 인터프리터가 튜플로 인식을 한다

[예제 13]

```
1)  print((1), (1,), ())
2)  print(type((1)), type((1,)), type(()))
```

```
1 (1,) ()
<class 'int'> <class 'tuple'> <class 'tuple'>
```

- 튜플 자료형이 갖고있는 **네 가지 큰 특징**은 다음과 같다

튜플 - 반복 가능(iterable)

KAIST

튜플은 리스트와 마찬가지로 반복 가능한(iterable) 자료형이다

[예제 13]

```
1)  my_first_tuple = (0, 1, 2, 3, 4, 5)
2)  print(type(my_first_tuple))
3)  print(len(my_first_tuple))
4)  for item in my_first_tuple:
5)      print(item, end=" ")
```

```
<class 'tuple'>
6
0 1 2 3 4 5
```

file에는 열람할 파일 경로를 입력한다

```
6)  print("\n"+str(my_first_tuple[3]))
7)  print(my_first_tuple[2:4])
8)  print(my_first_tuple[-1:-4:-2])
```

```
3
(2, 3)
(5, 3)
```

튜플 - 원소 자료형 무관

- 튜플 내부 원소들의 자료형 역시 리스트와 마찬가지로 일관되지 않아도 된다
- 즉, 원소가 어떠한 자료형을 가지는지 신경을 쓰지 않고 자유자재로 다룰 수 있다

```
1) my_tuple = (0, 1, "2")
2) my_second_tuple = (3, 4.0)
3) my_tuple = my_tuple.__add__(my_second_tuple)
4) print(my_tuple)
```

```
(0, 1, '2', 3, 4.0)
```

튜플 - 변경 불가능(immutable)

KAIST

- 파이썬에서 객체는 변경이 가능한(mutable) 자료형과 아닌(immutable) 것이 있다
- list, dict, set의 경우는 mutable 자료형에 속하고 나머지는 immutable하다고 생각하면 된다
- 튜플은 immutable하기에 내부 원소의 수정과 삭제가 불가능하다

[예제 16]

```
1) my_first_tuple = (0, 1, 2, 3, 4, 5)
2) del my_first_tuple[0] # 자료형 오류
```

TypeError: 'tuple' object doesn't support item deletion

[예제 17]

```
1) my_first_tuple = (0, 1, 2, 3, 4, 5)
2) my_first_tuple[0] = -1 # 자료형 오류
```

TypeError: 'tuple' object does not support item assignment

튜플 - 유익한 메소드 제공 및 그 외

KAIST

튜플에도 리스트와 마찬가지로 유익한 메소드(클래스 함수)를 제공하고 있다

- count, index, etc.

[예제 18]

```
1) my_tuple = (0, 1, "2")
2) print(my_tuple.count(0))
3) print(my_tuple.index("2"))
4) print(my_tuple.index(2))
```

```
1
2
```

ValueError: tuple.index(x): x not in tuple

다만, 튜플은 immutable하므로 원소 값의 수정 및 삭제와 관련된 메소드는 존재하지 않는다

내장 함수 len, max, min을 통해 튜플의 원소 개수, 최댓값, 최솟값을 구할 수 있다

[예제 19]

```
1) my_first_tuple = (0, 1, 2, 3, 4, 5)
2) print(len(my_first_tuple))
3) print(max(my_first_tuple))
4) print(min(my_first_tuple))
```

```
6
5
0
```

튜플 - 유익한 메소드 제공 및 그 외

KAIST

- 튜플에서도 + 과 * 연산자를 지원한다
- +는 튜플의 __add__ 메소드 역할과 동일하다
- *은 "튜플 자기 자신을 몇번 반복하여 연장할 것인가"를 수행하는 연산자이다

[예제 20]

```
1  tuple_A = (1, 2, 1, 3)
2  tuple_B = ("a", "B", 10.0)
3  print(tuple_A + tuple_B)
4  print(tuple_A * 3)
5  print((tuple_A * 3).count(1))
```

```
(1, 2, 1, 3, 'a', 'B', 10.0)
(1, 2, 1, 3, 1, 2, 1, 3, 1, 2, 1, 3)
6
```