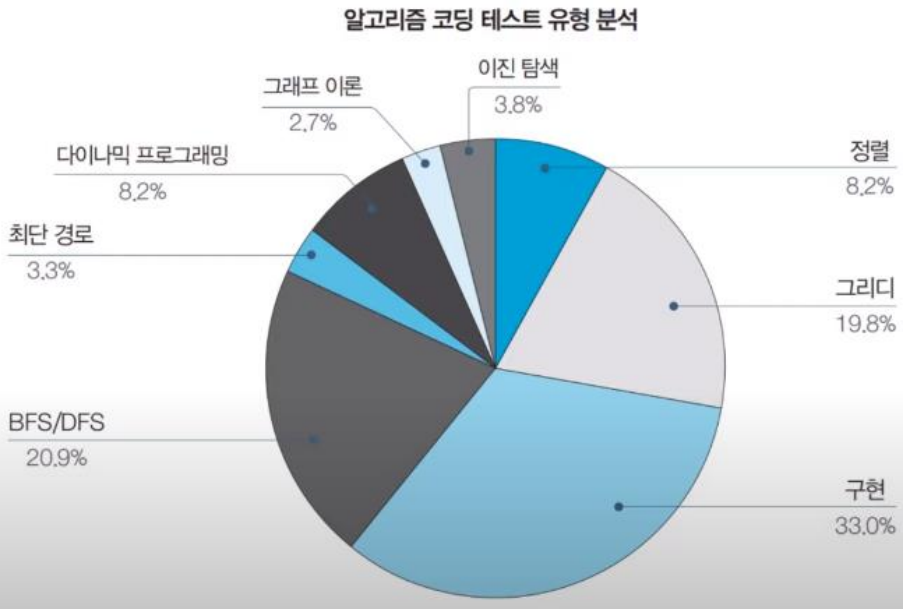
IT 기업 코딩 테스트 최신 출제 경향



2016 ~ 2019년에 출제되었던 국내외 주요 기업들의 공채에 등장한 알고리즘 유형

2019년 주요 기업 코딩 테스트 유형 분석

	날짜	풀이 시간	문제 개수	커트라인	주요 문제 유형	시험 유형
삼성전자	상반기 (2019-04-14)	3시간	2문제	2문제	완전 탐색, 시뮬레이션, 구현, DFS/BFS	오프라인
	하반기 (2019-10-20)					
카카오	1차 (2019-09-07)	5시간	7문제	4문제 (예상)	구현, 이진 탐색, 자료구조	온라인
	2차 (2019-09-21)	5시간	1문제	-	추천 시스템 개발	오프라인
라인	상반기 (2019-03-16)	3시간	5문제	3문제 (예상)	탐색, 구현, 문자열, 다이나믹 프로그래밍	온라인
	하반기 (2019-09-22)	3시간	6문제	4문제	자료구조, 완전 탐색, 구현	온라인

2018년 주요 기업 코딩 테스트 유형 분석

	날짜	풀이 시간	문제 개수	커트라인	주요 문제 유형	시험 유형
삼성전자	상반기 (2018-04-15)	3시간	2문제	1문제	완전 탐색, 구현, DFS/BFS, 시뮬레이션	오프라인
	하반기 (2018-10-21)					
카카오	1차 (2018-09-15)	5시간	7문제	3문제	그리디, 구현, 자료구조	온라인
	2차 (2018-10-06)	5시간	1문제	-	시뮬레이션 개발	오프라인
라인	상반기 (2018-04-05)	2시간	5문제	2문제	탐색, 그리디, 다이나믹 프로그래밍, 구현	온라인
	하반기 (2018-10-13)	2시간	4문제	2문제 (예상)	탐색, 그리디, 구현, 문자열	온라인

- 위 표는 다양한 후기 및 복원된 문제를 참고하여 작성된 것으로, 100% 일치하지는 않을 수 있습니다.

복잡도(Complexity)

- 복잡도는 알고리즘의 성능을 나타내는 척도입니다.
  - **시간 복잡도**: 특정한 크기의 입력에 대하여 알고리즘의 수행 시간 분석
  - **공간 복잡도**: 특정한 크기의 입력에 대하여 알고리즘의 메모리 사용량 분석
- 동일한 기능을 수행하는 알고리즘이 있다면, 일반적으로 복잡도가 낮을수록 좋은 알고리즘입니다.

빅오 표기법(Big-O Notation)

- 가장 빠르게 증가하는 항만을 고려하는 표기법입니다.
  - 함수의 상한만을 나타내게 됩니다.
- 예를 들어 연산 횟수가 $\mathbf{3N^3 + 5N^2 + 1,000,000}$인 알고리즘이 있다고 합시다.
  - 빅오 표기법에서는 차수가 가장 큰 항만 남기므로 $\boldsymbol{O(N^3)}$으로 표현됩니다.

빅오 표기법(Big-O Notation)

좋음(Better)

나쁨(Worse)

순위	명칭
$O(1)$	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

요구사항에 따라 적절한 알고리즘 설계하기

- 문제에서 가장 먼저 확인해야 하는 내용은 **시간제한(수행시간 요구사항)**입니다.
- 시간제한이 1초인 문제를 만났을 때, 일반적인 기준은 다음과 같습니다.
  - N의 범위가 500인 경우: 시간 복잡도가 $O(N^3)$인 알고리즘을 설계하면 문제를 풀 수 있습니다.
  - N의 범위가 2,000인 경우: 시간 복잡도가 $O(N^2)$인 알고리즘을 설계하면 문제를 풀 수 있습니다.
  - N의 범위가 100,000인 경우: 시간 복잡도가 $O(NlogN)$인 알고리즘을 설계하면 문제를 풀 수 있습니다
  - N의 범위가 10,000,000인 경우: 시간 복잡도가 $O(N)$인 알고리즘을 설계하면 문제를 풀 수 있습니다.

알고리즘 문제 해결 과정

- 일반적인 알고리즘 문제 해결 과정은 다음과 같습니다.
  1. 지문 읽기 및 컴퓨터적 사고
  2. **요구사항(복잡도) 분석**
  3. 문제 해결을 위한 아이디어 찾기
  4. 소스코드 설계 및 코딩
- 일반적으로 대부분의 문제 출제자들은 핵심 아이디어를 캐치한다면, 간결하게 소스코드를 작성할 수 있는 형태로 문제를 출제합니다.

수행 시간 측정 소스코드 예제

- 일반적인 알고리즘 문제 해결 과정은 다음과 같습니다.

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

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

자료형

- **모든 프로그래밍은 결국 데이터를 다루는 행위**입니다.
  - 자료형에 대한 이해는 프로그래밍의 길에 있어서의 첫걸음이라고 할 수 있습니다.
- 파이썬의 자료형으로는 정수형, 실수형, 복소수형, 문자열, 리스트, 튜플, 사전 등이 있습니다.
  - 파이썬의 자료형은 필수적으로 알아 두어야 합니다.

지수 표현 방식

- 파이썬에서는 e나 E를 이용한 **지수 표현 방식을 이용**할 수 있습니다.
  - e나 E 다음에 오는 수는 10의 지수부를 의미합니다.
  - 예를 들어 1e9라고 입력하게 되면, 10의 9제곱(1,000,000,000)이 됩니다.

$$\text{유효숫자}e^{\text{지수}} = \text{유효숫자} \times 10^{\text{지수}}$$

- 지수 표현 방식은 임의의 큰 수를 표현하기 위해 자주 사용됩니다.
- 최단 경로 알고리즘에서는 도달할 수 없는 노드에 대하여 최단 거리를 **무한(INF)**로 설정하곤 합니다.
- 이때 가능한 최댓값이 10억 미만이라면 무한(INF)의 값으로 1e9를 이용할 수 있습니다.

리스트 컴프리헨션

```python
# 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)
```

실행 결과

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


리스트 컴프리헨션

- 리스트 컴프리헨션은 **2차원 리스트를 초기화할 때 효과적으로 사용**될 수 있습니다.
- 특히 **N X M 크기**의 2차원 리스트를 한 번에 초기화 해야 할 때 매우 유용합니다.
  - 좋은 예시: array = [[0] * m for _ in range(n)]
- 만약 2차원 리스트를 초기화할 때 다음과 같이 작성하면 예기치 않은 결과가 나올 수 있습니다.
  - 잘못된 예시: array = [[0] * m] * n
  - 위 코드는 전체 리스트 안에 포함된 각 리스트가 모두 같은 객체로 인식됩니다.

리스트에서 특정 값을 가지는 원소를 모두 제거하기

```python
a = [1, 2, 3, 4, 5, 5, 5]
remove_set = {3, 5} # 집합 자료형 (집합 자료형은 추후에 다시 다룹니다.)

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

문자열 자료형

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

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

실행 결과

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


다만 문자열은 특정 인덱스의 값을 변경할 수는 없습니다. **(Immutable)**

튜플 자료형

- 튜플 자료형은 리스트와 유사하지만 다음과 같은 문법적 차이가 있습니다.
  - 튜플은 **한 번 선언된 값을 변경할 수 없습니다.**
  - 리스트는 대괄호([])를 이용하지만, 튜플은 소괄호(())를 이용합니다.
- 튜플은 리스트에 비해 상대적으로 **공간 효율적**입니다.

튜플을 사용하면 좋은 경우

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

사전 자료형

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

사전 자료형 관련 메서드

- 사전 자료형에서는 키와 값을 별도로 뽑아내기 위한 메서드를 지원합니다.
  - 키 데이터만 뽑아서 리스트로 이용할 때는 **keys() 함수**를 이용합니다.
  - 값 데이터만을 뽑아서 리스트로 이용할 때는 **values() 함수**를 이용합니다.

집합 자료형

- 집합은 다음과 같은 특징이 있습니다.
  - 중복을 허용하지 않습니다.
  - 순서가 없습니다.
- 집합은 **리스트 혹은 문자열을 이용해서 초기화**할 수 있습니다.
  - 이때 set() 함수를 이용합니다.
- 혹은 **중괄호 ({})**안에 각 원소를 콤마(,)를 기준으로 구분하여 삽입함으로써 초기화 할 수 있습니다.
- 데이터의 조회 및 수정에 있어서 O(1)의 시간에 처리할 수 있습니다.

집합 자료형

```
# 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)
```

실행 결과

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


집합 자료형의 연산

- 기본적인 집합 연산으로는 합집합, 교집합, 차집합 연산 등이 있습니다.
  - **합집합**: 집합 A에 속하거나 B에 속하는 원소로 이루어진 집합 $(A \cup B)$
  - **교집합**: 집합 A에도 속하고 B에도 속하는 원소로 이루어진 집합 $(A \cap B)$
  - **차집합**: 집합 A의 원소 중에서 B에 속하지 않는 원소들로 이루어진 집합 $(A - B)$

집합 자료형의 연산

```
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)
```

실행 결과

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

집합 자료형 관련 함수

```python
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)
```

실행 결과

```
{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 있습니다.
- 사전 자료형과 집합 자료형은 **순서가 없기 때문에** 인덱싱으로 값을 얻을 수 없습니다.
  - 사전의 키(Key) 혹은 집합의 원소(Element)를 이용해 $O(1)$의 시간 복잡도로 조회합니다.

빠르게 입력 받기

```python
import sys

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

파이썬의 pass 키워드

- **아무것도 처리하고 싶지 않을 때 pass 키워드를 사용**합니다.
- **예시)** 디버깅 과정에서 일단 조건문의 형태만 만들어 놓고 조건문을 처리하는 부분은 비워놓고 싶은 경우

```
score = 85

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

print('프로그램을 종료합니다.')
```

```
프로그램을 종료합니다.
```

조건문의 간소화

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

```
score = 85

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

```
Success
```

- 조건부 표현식(Conditional Expression)은 **if ~ else문을 한 줄에 작성**할 수 있도록 해줍니다.

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

print(result)
```

```
Success
```

global 키워드

- global 키워드로 변수를 지정하면 해당 함수에서는 지역 변수를 만들지 않고, **함수 바깥에 선언된 변수를 바로 참조**하게 됩니다.

```
a = 0

def func():
    global a
    a += 1

for i in range(10):
    func()

print(a)
```

```
10
```

람다 표현식

- 람다 표현식을 이용하면 함수를 간단하게 작성할 수 있습니다.
  - 특정한 기능을 수행하는 함수를 한 줄에 작성할 수 있다는 점이 특징입니다.

```
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))
```

```
10
10
```

람다 표현식 예시: 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]))
```

실행 결과

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

람다 표현식 예시: 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))
```

실행 결과

```
[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.

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

실행 결과

```
15
2 7
56
```

자주 사용되는 내장 함수

```
# 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)]
result = sorted(array, key=lambda x: x[1], reverse=True)
print(result)
```

실행 결과

```
[1, 4, 5, 8, 9]
[9, 8, 5, 4, 1]
[('이순신', 75), ('아무개', 50), ('홍길동', 35)]
```

순열과 조합

- **순열**: 서로 다른 n개에서 서로 다른 r개를 선택하여 일렬로 나열하는 것
  - {'A', 'B', 'C'}에서 두 개를 선택하여 나열하는 경우: 'ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA'

```
from itertools import permutations

data = ['A', 'B', 'C'] # 데이터 준비

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

실행 결과: `[('A','B','C'), ('A','C','B'), ('B','A','C'), ('B','C','A'), ('C','A','B'), ('C','B','A')]`

중복 순열과 중복 조합

```
from itertools import product

data = ['A', 'B', 'C'] # 데이터 준비

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

```
from itertools import combinations_with_replacement

data = ['A', 'B', 'C'] # 데이터 준비

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

Counter

- 파이썬 collections 라이브러리의 **Counter**는 등장 횟수를 세는 기능을 제공합니다.
- 리스트와 같은 반복 가능한(iterable) 객체가 주어졌을 때 내부의 원소가 몇 번씩 등장했는지를 알려줍니다.

```
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 반환
```

실행 결과:

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

최대 공약수와 최소 공배수

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

```
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
```

실행 결과:

```
7
42
```