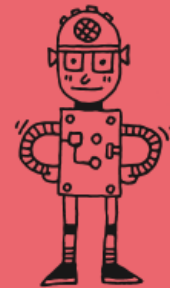


# 모두의 알고리즘 with 파이썬

---

컴퓨팅 사고를 위한 기초 알고리즘



# 본격적인 알고리즘 공부에 앞서

---

**01** 알고리즘

**02** 알고리즘 분석

**03** 파이썬 프로그래밍 언어

# 1 알고리즘

- 알고리즘은 어떤 문제를 풀기 위한 절차나 방법
- 알고리즘은 주어진 '입력'을 '출력'으로 만드는 과정
- 알고리즘의 각 단계는 구체적이고 명료해야 함

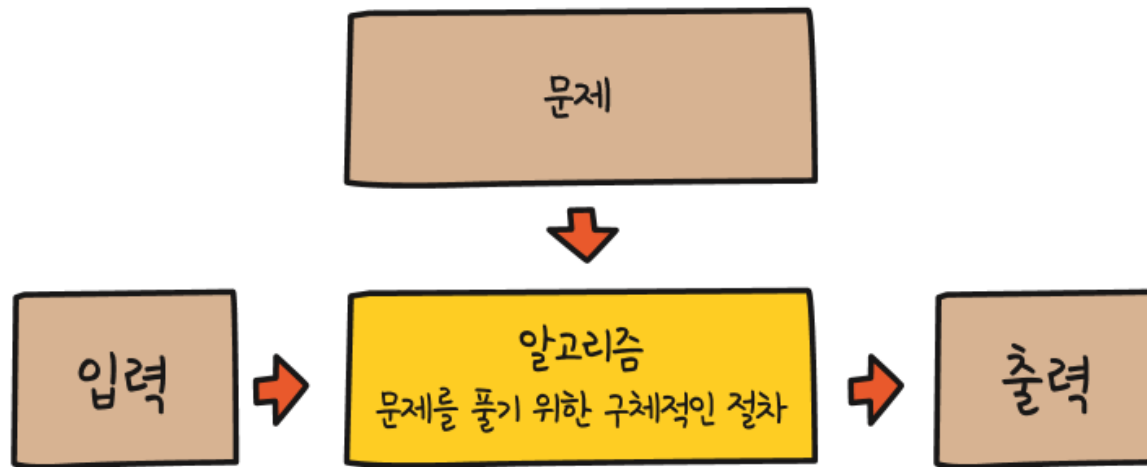


그림 0-1 알고리즘

# 1 알고리즘

- 문제: 어떤 숫자의 절댓값 구하기
- 입력: 절댓값을 구할 실수  $a$
- 출력:  $a$ 의 절댓값

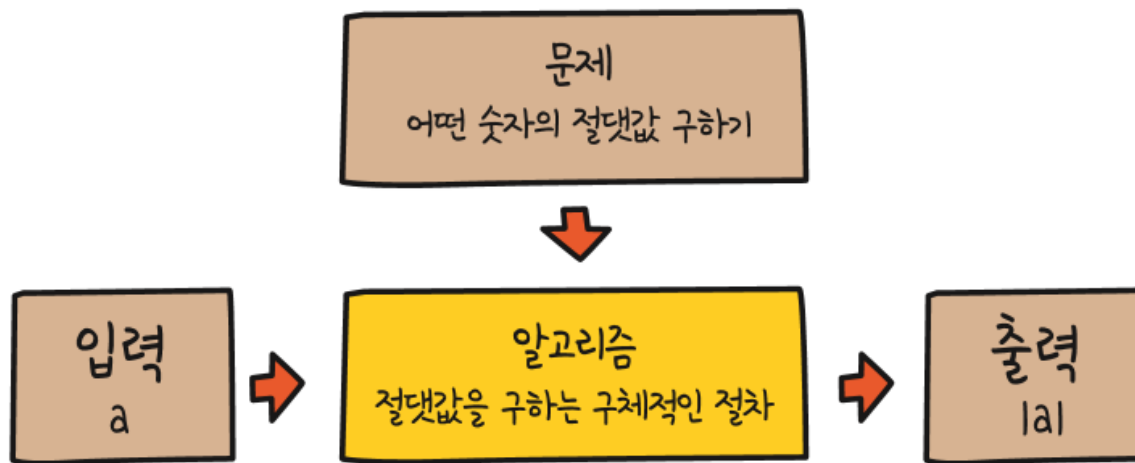


그림 0-2 절댓값을 구하는 알고리즘의 구조

## 1 알고리즘

- 절댓값: 0부터 그 수까지의 거리에 해당하는 값
- $|a|$ 와 같이 절댓값을 나타내는 기호인 세로 선( $|$ )으로 수를 둘러싸서 표현
- 주어진 실수  $a$ 가 양수 혹은 0이면  $a$  값이 그대로 절댓값이 됨
- $a$ 가 음수이면  $a$ 에 마이너스(-)를 붙이면 절댓값이 됨



명료하게

- 1)  $a$ 가 0보다 크거나 같은지 확인 → 그렇다면  $a$ 를 결과로 돌려줌
- 2) 1의 경우가 아니라면( $a$ 가 0보다 작으면)  $-a$ 를 결과로 돌려줌

$$|a| = \begin{cases} a, & a \geq 0 \\ -a, & a < 0 \end{cases}$$

## 1 알고리즘

- 1)  $a$ 가 0보다 크거나 같은지 확인  $\rightarrow$  그렇다면  $a$ 를 결과로 돌려줌
- 2) 1의 경우가 아니라면( $a$ 가 0보다 작으면)  $-a$ 를 결과로 돌려줌

$$|a| = \begin{cases} a, & a \geq 0 \\ -a, & a < 0 \end{cases}$$



### 실수의 절댓값을 구하는 알고리즘

- $a = 5$ 일 때  $a$ 가 0보다 크거나 같음( $a \geq 0$ )  $\rightarrow$  1의 경우  
 $a$  값인 5가 결과
- $a = -3$ 일 때  $a$ 가 0보다 작음( $a < 0$ )  $\rightarrow$  2의 경우  
 $-a$  값인  $-(-3) = 3$ 이므로 3이 결과

# 1 알고리즘

- 1)  $a$ 가 0보다 크거나 같은지 확인 → 그렇다면  $a$ 를 결과로 돌려줌
- 2) 1의 경우가 아니라면( $a$ 가 0보다 작으면)  $-a$ 를 결과로 돌려줌

$$|a| = \begin{cases} a, & a \geq 0 \\ -a, & a < 0 \end{cases}$$



- 이처럼 컴퓨터 프로그램을 만들기 위한 알고리즘은 계산 과정을 최대한 구체적이고 명료하게 적어야 함
- 컴퓨터는 주어진 명령에 따라 계산을 수행하는 기계이므로 알고리즘이 구체적이지 않으면 올바르게 계산할 수 없음

## 2 알고리즘 분석

- 어떤 문제를 푸는 방법이 꼭 한 가지만 있는 것은 아님

### 절댓값을 구하는 방법들

- 0보다 큰지 작은지를 비교해 부호를 확인하는 방법
- 주어진  $a$ 를 제곱한 다음 그 값의 제곱근을 취하는 방법

$$|a| = \sqrt{a^2}$$

$$\textcircled{1} a = -3$$

$$(-3)^2 = 9$$

$$|-3| = \sqrt{(-3)^2} = \sqrt{9} = 3$$

$$\textcircled{2} a = 2$$

$$|2| = \sqrt{2^2} = \sqrt{4} = 2$$



## 알고리즘 분석

- 한 가지 문제를 푸는 여러 가지 방법, 즉 여러 가지 알고리즘 중에 상황에 맞는 적당한 알고리즘을 골라 쓰려면 어떤 알고리즘이 어떤 특징을 지니고 있으며 얼마나 계산이 빠르고 편한지 등을 알아야 함



알고리즘 분석 필요

big O Notation

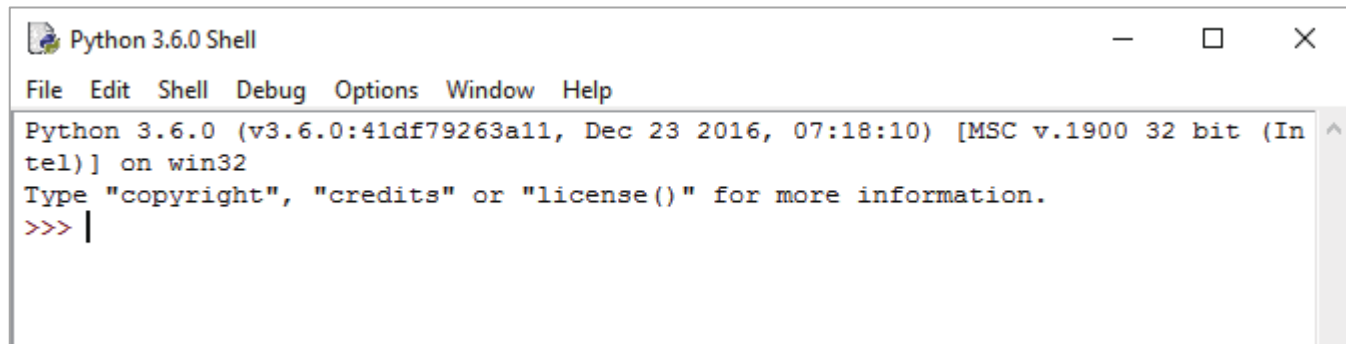
- 알고리즘 분석은 알고리즘의 성능이나 특징을 분석하는 것

- 시간 : time complexity

- 메모리 : space complexity

## 3 파이썬 프로그래밍 언어

- 파이썬 3로 준비하기
- 파이썬을 실행해서 버전 확인하기



```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> |
```

그림 0-3 파이썬을 실행해서 버전을 확인하기

### 3 파이썬 프로그래밍 언어

- 절댓값 구하기 알고리즘

1) a의 절댓값 구하기 알고리즘 ①: 부호 판단 → `abs_sign(a)`

- a가 0보다 크거나 같은지 확인 → 그렇다면 a를 결과로 돌려줌
- 위의 경우가 아니라면(a가 0보다 작다면) -a를 결과로 돌려줌

$$|a| = \begin{cases} a, & a \geq 0 \\ -a, & a < 0 \end{cases}$$

2) a의 절댓값 구하기 알고리즘 ②: 제곱 후 제곱근 → `abs_square(a)`

- a를 제곱하여 변수 b에 저장
- b의 제곱근을 구해 결과로 돌려줌

$$b = a^2$$

$$|a| = \sqrt{b}$$

## 3 파이썬 프로그래밍 언어

### ■ 프로그램 0-1 절댓값 구하기 알고리즘

▼ 예제 소스 p00-1-abs.py

```
import math # 수학 모듈 사용

# 절댓값 알고리즘 1(부호 판단)
# 입력: 실수 a
# 출력: a의 절댓값

def abs_sign(a):
    if a >= 0:
        return a
    else:
        return -a
```



## 3 파이썬 프로그래밍 언어



```
# 절댓값 알고리즘 2(제곱-제곱근)
# 입력: 실수 a
# 출력: a의 절댓값

def abs_square(a):
    b = a * a
    return math.sqrt(b) # 수학 모듈의 제곱근 함수

print(abs_sign(5))
print(abs_sign(-3))
print()
print(abs_square(5))
print(abs_square(-3))
```

## 3 파이썬 프로그래밍 언어

### ■ 실행 결과

```
5  
3  
  
5.0  
3.0
```

- 파이썬의 제곱근 함수인 `math.sqrt(b)`는 소수점이 붙은 값을 돌려줌
- 5는 5.0과 같고 3은 3.0과 같은 결과

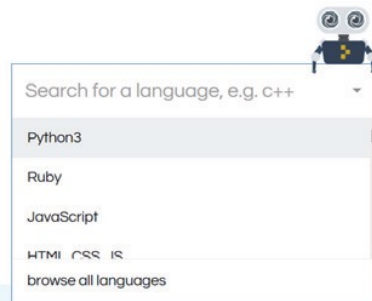
## 3 파이썬 프로그래밍 언어

### 웹에서 파이썬 사용하기

- <https://repl.it>에 접속 해서 Python 3을 선택
- 주소 창에 <https://repl.it/languages/python3>을 입력

repl.it is a cloud coding environment for Jest

join a community of  
*engineers, teachers, and students*



## 3 파이썬 프로그래밍 언어

- 회원 가입을 하면 입력한 프로그램을 저장할 수도 있음



The screenshot shows the repl.it web interface. The top bar includes the repl.it logo, a file name 'Untitled', and a 'Log in' button. Below the bar are buttons for 'share', 'save', 'run', and a play icon. The main editor area contains Python code for a function that calculates the sum of numbers from 1 to n. The code is as follows:

```
1 # 연속한 숫자의 합 구하기 알고리즘
2 # 입력: n
3 # 출력: 1부터 n까지 연속한 숫자를 더한 합
4
5 def sum_n(n):
6     s = 0 # 합을 계산할 변수
7     for i in range(1, n + 1): # 1부터 n까지 반복 (n+1 제외)
8         s = s + i
9     return s
10
11 print(sum_n(10)) # 1부터 10까지 합 (입력:10, 출력:55)
12 print(sum_n(100)) # 1부터 100까지 합 (입력:100, 출력:5050)
13
```

On the right side, there is a terminal window with 'input' and 'clear' buttons. It shows the Python version (3.5.2) and GCC version (4.8.2) on Linux. The output of the code execution is displayed as follows:

```
Python 3.5.2 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
55
5050
```



# 첫째 마당 알고리즘 기초

---

문제01 1부터 n까지의 합 구하기

문제02 최댓값 찾기

문제03 동명이인 찾기①

## 문제 01 1부터 $n$ 까지의 합 구하기

---

## 문제01 1부터 $n$ 까지의 합 구하기

0

**1부터  $n$ 까지 연속한 정수의 합을 구하는 알고리즘을 만들어 보세요.**

- 1부터 10까지의 수를 모두 더하면? 55
- 1부터 100까지의 수를 모두 더하면? 5050

## 1 알고리즘의 중요 포인트

### 문제

- 알고리즘은 주어진 문제를 풀기 위한 절차나 방법
- 알고리즘이 있으려면 반드시 문제가 필요
- 여기서는 '1부터  $n$ 까지 연속한 숫자의 합 구하기'가 바로 문제

### 입력

- 알고리즘은 주어진 '입력'을 '출력'으로 만드는 과정
- 이 문제에서 입력은 ' $n$ 까지'에 해당하는  $n$
- $n$ 을 입력으로 하는 문제를 만들면 만들어진 알고리즘으로 다양한 입력에 대한 결과를 얻을 수 있음

## 1 알고리즘의 중요 포인트

### 출력

- $n = 10$ 이면 1부터 10까지의 합은 55
- $n = 100$ 이면 1부터 100까지의 합은 5050
- 55와 5050이 각각의 입력에 대한 출력

## 2 구체적이고 명료한 계산 과정

- 우리는 다음과 같은 방식으로 1부터 10까지의 합을 구하고 있음
- 1) 1 더하기 2를 계산한 결과인 3을 머릿속에 기억
- 2) 기억해 둔 3에 다음 숫자 3을 더해 6을 기억
- 3) 기억해 둔 6에 다음 숫자 4를 더해 10을 기억
- 4) 기억해 둔 10에 다음 숫자 5를 더해 15를 기억
- 5~8) 같은 과정 반복
- 9) 기억해 둔 45에 다음 숫자인 10을 더해 55를 기억
- 10) 10까지 다 더했으므로 마지막에 기억된 55를 답으로 제시

## 2 구체적이고 명료한 계산 과정

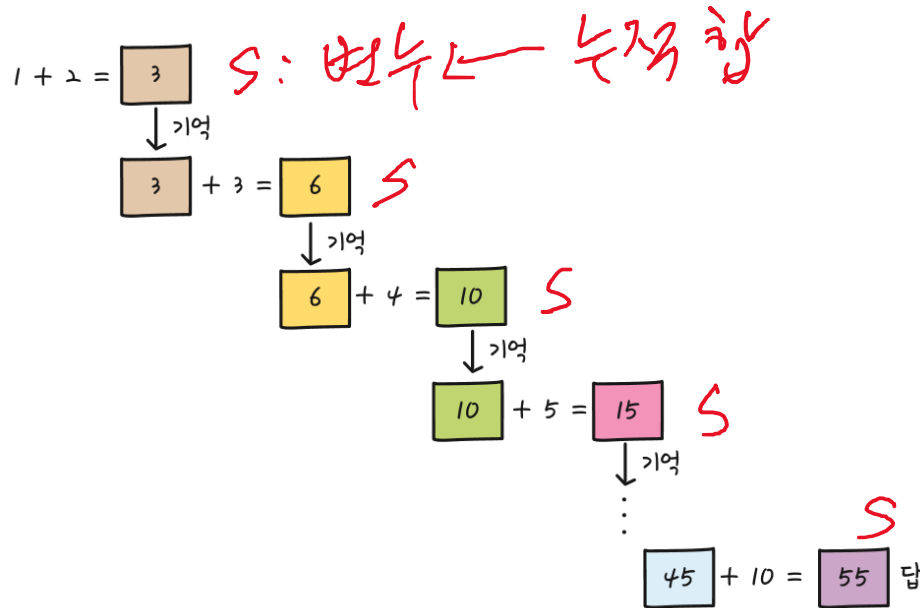


그림 1-1 1부터 10까지의 합을 구하는 과정(사람의 생각)

- 사람과 달리 컴퓨터는 주어진 명령을 기계적으로 수행하는 장치이므로 기계가 알아들을 수 있는 명료하고 구체적인 알고리즘이 있어야 함

3 1부터 n까지의 합을 구하는 알고리즘

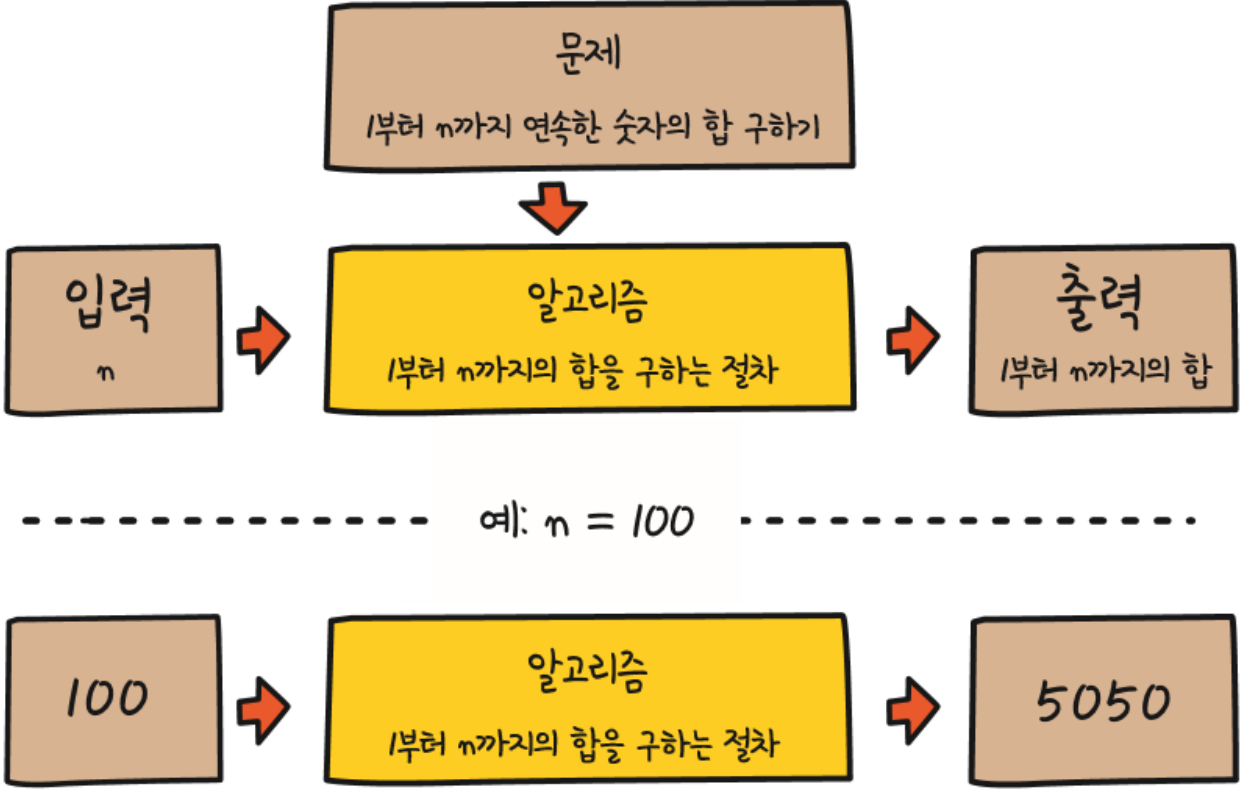


그림 1-2 1부터 n까지의 합을 구하는 알고리즘의 구조와 예



## 1부터 n까지의 합을 구하는 알고리즘

- 1부터 n까지 연속한 숫자의 합을 구하는 문제를 풀기 위한 알고리즘
  - 1) 합을 기록할 변수  $s$ 를 만들고 0을 저장
  - 2) 변수  $i$ 를 만들어 1부터 n까지의 숫자를 1씩 증가시키며 반복
  - 3) [반복 블록] 기존의  $s$ 에  $i$ 를 더하여 얻은 값을 다시  $s$ 에 저장
  - 4) 반복이 끝났을 때  $s$ 에 저장된 값이 결과값
- 알고리즘을 하나의 함수로 만들어 입력은 인자로 전달
- 출력은 함수의 결과값(return 값)
- 알고리즘은 '입력 → 알고리즘 → 출력'을 수행하는 과정

## 1부터 n까지의 합을 구하는 알고리즘

- 프로그램 1-1 1부터 n까지의 연속한 숫자의 합을 구하는 알고리즘①

▼ 예제 소스 p01-1-sum.py

```
# 1부터 n까지 연속한 숫자의 합을 구하는 알고리즘 1
# 입력: n
# 출력: 1부터 n까지의 숫자를 더한 값

def sum_n(n):
    s = 0 # 합을 계산할 변수
    for i in range(1, n + 1): # 1부터 n까지 반복 (n+1은 제외)
        s = s + i
    return s

print(sum_n(10)) # 1부터 10까지 합(입력:10, 출력:55)
print(sum_n(100)) # 1부터 100까지 합(입력:100, 출력:5050)
```

## 3 1부터 n까지의 합을 구하는 알고리즘

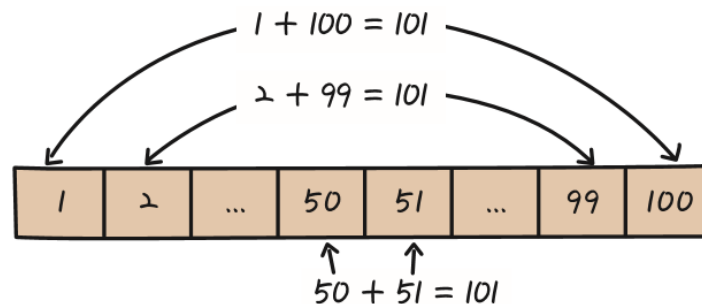
- 실행 결과

```
55
5050
```

- 프로그램 1-1의 알고리즘은 합을 계산할 변수 s에 0을 넣고 첫 번째 수인 1을 더하는 것으로 계산을 시작
- 즉,  $1 + 2 = 3$ 이 아니라  $0 + 1 = 1$ 이 첫 덧셈

## 4 알고리즘 분석

- 한 문제를 푸는 방법은 보통 여러 가지
  - 1부터 100까지의 합을 구하는 문제만 해도 최소 두 가지 방법이 있음
- 앞에서 만든 프로그램처럼 1부터 100까지의 숫자를 차례로 더하는 방법
  - 수학 천재 가우스의 방법



101이 50번 나오므로  $101 \times 50 = 5050$

그림 1-3 가우스의 방법

$$\begin{array}{r} 100 \times 101 \\ \hline 2 \\ \hline \end{array} \rightarrow \begin{array}{r} n \times (n+1) \\ \hline 2 \end{array}$$

## 알고리즘 분석

- 이러한 발견을 일반화해서 만든 1부터 n까지의 합 공식

$$\frac{n(n+1)}{2}$$

- 이 공식을 이용해 '1부터 n까지 연속한 숫자의 합 구하기' 문제를 푸는 파이썬 프로그램을 만들어 보자

## 알고리즘 분석

- 프로그램 1-2 1부터 n까지의 연속한 숫자의 합을 구하는 알고리즘②

▼ 예제 소스 p01-2-sum.py

```
# 1부터 n까지 연속한 숫자의 합을 구하는 알고리즘 2
# 입력: n
# 출력: 1부터 n까지의 숫자를 더한 값

def sum_n(n):
    return n * (n + 1) // 2 # 슬래시 두 개(//)는 정수 나눗셈을 의미

print(sum_n(10)) # 1부터 10까지 합(입력:10, 출력:55)
print(sum_n(100)) # 1부터 100까지 합(입력:100, 출력:5050)
```

## 알고리즘 분석

- 실행 결과

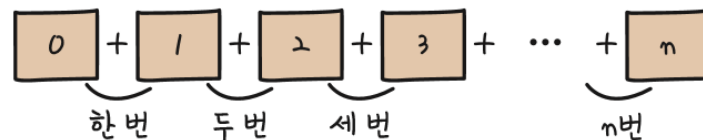
```
55
5050
```

- 첫 번째 방법은 입력 값  $n$ 이 커질수록 덧셈을 훨씬 더 많이 반복해야 함
- 두 번째 방법은  $n$  값의 크기와 관계없이 덧셈, 곱셈, 나눗셈을 각각 한 번씩만 하면 답을 얻을 수 있음

## 문제01 1부터 n까지의 합 구하기

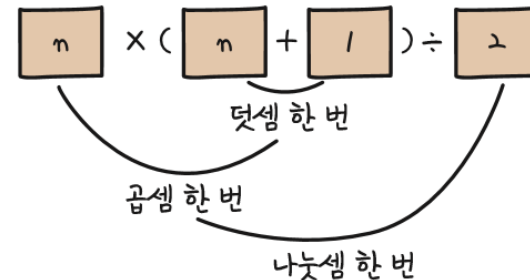
### 4 알고리즘 분석

첫 번째 방법



$O(n)$

두 번째 방법



$O(1)$

그림 1-4 1부터 n까지의 연속한 숫자의 합을 구하는 두 가지 방법

- 주어진 문제를 푸는 여러 가지 방법 중 어떤 방법이 더 좋은 것인지 판단할 때 필요한 것이 '알고리즘 분석'



## 5 입력 크기와 계산 횟수

- 알고리즘에는 입력이 필요
- 입력 크기가 알고리즘의 수행 성능에 영향을 미칠 때가 많음
- 입력 크기가 커지면 보통 알고리즘의 계산도 복잡해짐

### 입력 크기와 계산 횟수의 관계 생각해 보기

- 1부터  $n$ 까지의 합을 구하는 문제에서는  $n$ 의 크기가 바로 입력 크기
- 첫 번째 알고리즘(프로그램 1-1): 덧셈  $n$ 번  
→ 10까지의 합을 구하려면 덧셈을 열 번, 100까지의 합을 구하려면 백 번
- 두 번째 알고리즘(프로그램 1-2): 덧셈, 곱셈, 나눗셈 각 한 번(총 세 번)  
→ 입력 크기  $n$ 이 아무리 큰 수라도 덧셈, 곱셈, 나눗셈 각 한 번

## 5 입력 크기와 계산 횟수

- $n$ 이 커지면 커질수록 계산 횟수의 차이가 커짐
- $n = 1000$ 인 경우 첫 번째 알고리즘은 덧셈을 천 번 해야 함  
→ 세 번만 계산해도 되는 두 번째 알고리즘과 엄청나게 차이가 남
- 보통 컴퓨터를 이용해서 계산할 때는 입력 크기  $n$ 이 매우 큰 경우가 많음
- 알고리즘 분석에서는 입력 크기가 매우 큰  $n$ 에 대해서 따져 보는 것이 중요
- 예) 우리나라 전 국민의 평균 나이를 계산하는 문제  
→ 입력 크기  $n$ 은 우리나라 인구수에 해당하는 51,736,224 (2017년 6월)

## 대문자 O 표기법: 계산 복잡도 표현

- 계산 복잡도(complexity): 어떤 알고리즘이 문제를 풀기 위해 해야 하는 계산이 얼마나 복잡한지를 나타낸 정도
- 계산 복잡도를 표현하는 방법에는 여러 가지가 있음
- 그 중 대문자 O 표기법을 가장 많이 사용  
(대문자 O 표기법은 '빅 오' 표기법이라고도 부름)

## 대문자 O 표기법: 계산 복잡도 표현

예제 프로그램의 계산 복잡도를 대문자 O로 표기하는 방법①

- 첫 번째 알고리즘은 입력 크기  $n$ 에 대해 사칙 연산(덧셈)을  $n$ 번 해야 함
- 이때 이 알고리즘의 계산 복잡도를  $O(n)$ 이라고 표현
- 필요한 계산 횟수가 입력 크기에 '정비례'할 때는  $O(n)$ 이라고 표현
- 대문자 O 표기법은 알고리즘에서 필요한 계산 횟수를 정확한 숫자로 표현하는 것이 아니라 입력 크기와의 관계로 표현
- 입력 크기  $n$ 에 따라 덧셈을 두 번씩 하는 알고리즘도  $O(2n)$ 이 아니라 마찬가지로 그냥  $O(n)$ 으로 표현
- $n$ 이 10에서 20으로 '2배'가 될 때  $2n$ 도 20에서 40으로 '2배'가 됨(정비례)

## 대문자 O 표기법: 계산 복잡도 표현

예제 프로그램의 계산 복잡도를 대문자 O로 표기하는 방법②

- 두 번째 알고리즘은 입력 크기  $n$ 과 무관하게 사칙연산을 세 번 해야 함
- 이때 알고리즘의 계산 복잡도는  $O(1)$ 로 표현
- 입력 크기  $n$ 과 필요한 계산의 횟수가 무관하다면, 즉 입력 크기가 커져도 계산 시간이 더 늘어나지 않는다면  $O(1)$ 로 표기
- 사칙연산을 세 번 한다고  $O(3)$ 으로 표현하지 않음
- 앞에서  $O(2n)$ 을  $O(n)$ 으로 표현하는 것과 같은 원리

## 대문자 O 표기법: 계산 복잡도 표현

- 대문자 O 표기법은 알고리즘의 대략적인 성능을 표시하는 방법
- 세밀한 계산 횟수나 소요 시간을 표현하는 것이 아님
- 입력 크기  $n$ 과 필요한 계산 횟수와의 '관계'에 더 주목하는 표현
- 입력 크기가 큰 문제를 풀 때는 보통  $O(1)$ 인 알고리즘이 훨씬 더 빠름



- $O(n)$ : 필요한 계산 횟수가 입력 크기  $n$ 과 비례할 때
- $O(1)$ : 필요한 계산 횟수가 입력 크기  $n$ 과 무관할 때

## 대문자 O 표기법: 계산 복잡도 표현

### 시간 복잡도

- 어떤 알고리즘을 수행하는 데 얼마나 오랜 시간이 걸리는지 분석한 것
- 사칙연산 횟수로 계산 복잡도를 생각해 본 것은 시간 복잡도에 해당
- 어떤 알고리즘을 수행하는 데 필요한 사칙연산의 횟수가 많아지면 결국 알고리즘 전체를 수행하는 시간이 늘어나기 때문

### 공간 복잡도

- 어떤 알고리즘을 수행하는 데 얼마나 많은 공간(메모리/기억 장소)이 필요한지 분석한 것

### 연습 문제



1-1 1부터 n까지 연속한 숫자의 제곱의 합을 구하는 프로그램을 for 반복문으로 만들어 보세요(예를 들어  $n = 10$ 이라면  $1^2 + 2^2 + 3^2 + \dots + 10^2 = 385$ 를 계산하는 프로그램입니다).

$$1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$$



1-2 연습 문제 1-1 프로그램의 계산 복잡도는  $O(1)$ 과  $O(n)$  중 무엇일까요?

- 1-3 1부터 n까지 연속한 숫자의 제곱의 합을 구하는 공식은  $\frac{n(n+1)(2n+1)}{6}$ 로 알려져 있습니다. for 반복문 대신 이 공식을 이용하면 알고리즘의 계산 복잡도는  $O(1)$ 과  $O(n)$  중 무엇이 될까요?



## 문제01 1부터 n까지의 합 구하기

# 연습 문제 풀이

부록 A

### ■ 1-1 1부터 n까지 제곱의 합을 구하는 프로그램

#### ▼ 예제 소스 e01-1-sumsq.py

```
# 연속한 숫자 제곱의 합을 구하는 알고리즘  
# 입력: n  
# 출력: 1부터 n까지 연속한 숫자의 제곱을 더한 합
```

```
def sum_sq(n):
```

```
    s = 0
```

```
    for i in range(1, n + 1):
```

```
        s = s + i * i
```

```
    return s
```

```
print(sum_sq(10)) # 1부터 10까지 제곱의 합(입력:10, 출력:385)
```

```
print(sum_sq(100)) # 1부터 100까지 제곱의 합(입력:100, 출력:338350)
```

연습 문제 풀이

부록 A

■ 실행 결과

385  
338350

## 연습 문제 풀이

부록 A

- 1-2 계산 복잡도

$O(n)$

- 곱셈  $n$ 번, 덧셈  $n$ 번으로 사칙연산이 총  $2n$ 번 필요하지만,  $O(n)$ 으로 표현

## 연습 문제 풀이

부록 A

- 1-3 계산 복잡도(공식 이용)

$O(1)$

- 덧셈 두 번, 곱셈 세 번, 나눗셈 한 번으로 사칙연산이 총 여섯 번 필요하지만, 이 값은  $n$ 의 크기와 상관없이 일정한 값이므로  $O(1)$ 로 표현

# 연습 문제 풀이

부록 A

### ■ 1-3 계산 복잡도(공식 이용)

#### ▼ 예제 소스 e01-3-sumsq.py

```
# 연속한 숫자 제공의 합 구하기 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자의 제곱을 더한 합

def sum_sq(n):
    return n * (n + 1) * (2 * n + 1) // 6

print(sum_sq(10)) # 1부터 10까지 제공의 합(입력:10, 출력:385)
print(sum_sq(100)) # 1부터 100까지 제공의 합(입력:100, 출력:338350)
```

- 참고 : <https://lifeignite.tistory.com/9>

연습 문제 풀이

부록 A

■ 실행 결과

385  
338350

$$1 + 2 + 3 + 4 + \cdots + n = \sum_{i=1}^n i$$

다음과 같이 표기한다. 보통 시그마라고 읽는다. 보통 우리들은 이 공식들을 외우고 있다.

$$1) 1 + 2 + 3 + 4 + \cdots + n = \sum_{i=1}^n i$$

$$2) n + \cdots + 4 + 3 + 2 + 1 = \sum_{i=1}^n i$$

$$3) (n + 1) + (n + 1) + \cdots (n + 1) = 2 \sum_{i=1}^n i$$

$$\frac{n * (n + 1)}{2} = \sum_{i=1}^n i$$

$$\sum_{i=1}^n i = 1+2+3+4+\cdots+n = \frac{n(n+1)}{2} \quad \sum_{i=1}^n i^2 = 1^2+2^2+3^2+4^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^n i^3 = 1^3+2^3+3^3+4^3+\cdots+n^3 = \left( \frac{n(n+1)}{2} \right)^2$$



## 문제 02 최댓값 찾기

---

주어진 숫자  $n$ 개 중 가장 큰 숫자를 찾는 알고리즘을 만들어 보세요.

- 주어진 숫자  $n$ 개 중에서 가장 큰 숫자(최댓값)를 찾는 문제
- 17, 92, 18, 33, 58, 7, 33, 42와 같이 숫자가 여덟 개가 있을 때 최댓값은? 92

# 1 리스트

- 리스트(list): 정보 여러 개를 하나로 묶어 저장하고 관리할 수 있는 기능
- 숫자 여러 개는 파이썬의 리스트 기능을 이용하면 쉽게 관리할 수 있음
- 리스트는 대괄호([ ]) 안에 정보 여러 개를 쉼표(,)로 구분하여 적음

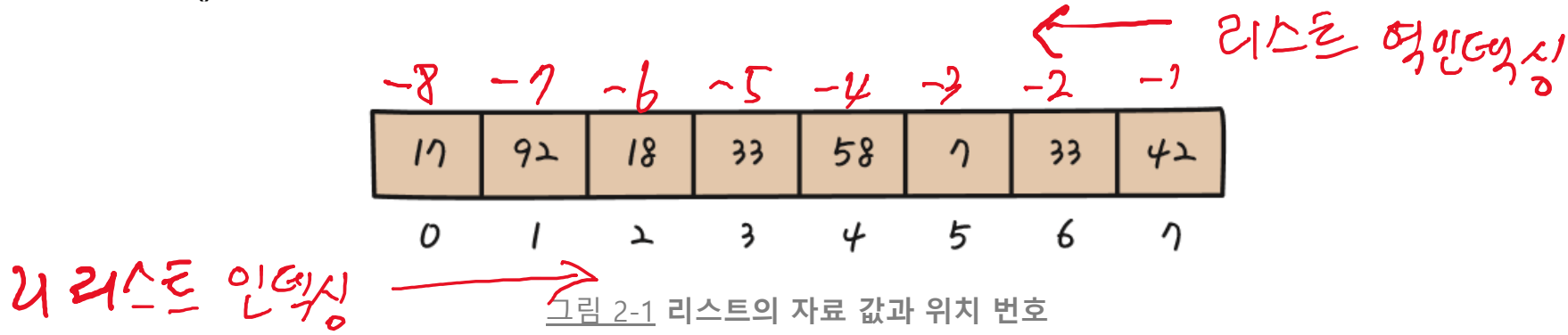
```
>>> a = [5, 7, 9]
>>> a
[5, 7, 9]
>>> a[0]
5
>>> a[2]
9
>>> a[-1]
9
>>> len(a)
3
```

### 1 리스트

- $a = [5, 7, 9]$   
5, 7, 9라는 정수 세 개를 묶은 리스트를 만들어  $a$ 에 저장
- $a$  입력  $\rightarrow [5, 7, 9]$  표시  
 $a$ 가 5, 7, 9라는 정보 세 개를 묶어 놓은 리스트라고 알려주는 것
- **주의!** 리스트는 자료 위치를 1이 아닌 0부터 센다
- $a[0] \rightarrow 5$  표시  
리스트  $a$ 의 0번 위치 값을 의미, 5, 7, 9 중 맨 앞에 있는 값인 5 표시
- $a[2] \rightarrow 9$  표시  
리스트의 마지막 값인 9 표시

1 리스트

- `a[-1]` → 9 표시  
리스트에서 위치 번호 -1은 리스트의 끝에서 첫 번째 값, 즉 마지막 값
- `len(a)` → 3 표시  
리스트 a에는 세 개의 값인 [5, 7, 9]가 들어 있으므로 3 표시
- `len()` 함수를 사용하면 어떤 리스트 안에 들어 있는 자료 개수를 알 수 있음



1 리스트

함수	설명	사용 예
len(a)	리스트 길이(자료 개수)를 구합니다.	<pre>a = [] len(a)           # 빈 리스트이므로 0 len( [1, 2, 3] ) # 자료 개수가 세 개이므로 3</pre>
append(x)	자료 x를 리스트의 맨 뒤에 추가합니다.	<pre>a = [1, 2, 3] a.append(4)      # a는 [1, 2, 3, 4]가 됩니다.</pre>
insert(i, x)	리스트의 i번 위치에 x를 추가합니다.	<pre>a = [1, 2, 3] a.insert(0, 5)   # 0번 위치(맨 앞)에 5를 추가합니다.                  # a = [5, 1, 2, 3]이 됩니다.</pre>
pop(i)	i번 위치에 있는 자료를 리스트에서 빼내면서 그 값을 함수의 결과값으로 돌려줍니다. i를 지정하지 않으면 맨 마지막 값을 빼내서 돌려줍니다.	<pre>a = [1, 2, 3] print(a.pop())  # 3이 출력되고 a = [1, 2]가 됩니다.</pre>
clear()	리스트의 모든 자료를 지웁니다.	<pre>a = [1, 2, 3] a.clear()       # a = [], 즉 빈 리스트가 됩니다.</pre>
x in a	어떤 자료 x가 리스트 a 안에 있는지 확인합니다(x not in a는 반대 결과).	<pre>a = [1, 2, 3] 2 in a          # 2가 리스트 a 안에 있으므로 True 5 in a          # 5가 리스트 a 안에 없으므로 False 5 not in a      # 5가 리스트 a 안에 없으므로 True</pre>

표 2-1 자주 쓰는 리스트 기능

## 2 최댓값을 찾는 알고리즘

- 17, 92, 18, 33, 58, 7, 33, 42 중 최댓값을 찾는 알고리즘(사람의 생각)
  - 1) 첫 번째 숫자 17을 최댓값으로 기억(최댓값: 17)
  - 2) 두 번째 숫자 92를 현재 최댓값 17과 비교  
92는 17보다 크므로 최댓값을 92로 바꿔 기억(최댓값: 92)
  - 3) 세 번째 숫자 18을 현재 최댓값 92와 비교, 작으므로 지나감(최댓값: 92)
  - 4~7) 네 번째 숫자부터 일곱 번째 숫자까지 같은 과정 반복
  - 8) 마지막 숫자 42를 현재 최댓값 92와 비교, 작으므로 지나감(최댓값: 92)
  - 9) 마지막으로 기억된 92가 주어진 숫자 중 최댓값

## 2 최댓값을 찾는 알고리즘

### ■ 프로그램 2-1 최댓값을 구하는 알고리즘

▼ 예제 소스 p02-1-findmax.py

```
# 최댓값 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중 최댓값

def find_max(a):
    n = len(a) # 입력 크기 n
    max_v = a[0] # 리스트의 첫 번째 값을 최댓값으로 기억
    for i in range(1, n): # 1부터 n-1까지 반복
        if a[i] > max_v: # 이번 값이 현재까지 기억된 최댓값보다 크면
            max_v = a[i] # 최댓값을 변경
    return max_v
```





## 2 최댓값을 찾는 알고리즘



```
v = [17, 92, 18, 33, 58, 7, 33, 42]  
print(find_max(v))
```

### ■ 실행 결과

```
92
```

### 3 알고리즘 분석

#### 최댓값 구하기 프로그램의 계산 복잡도(시간 복잡도)

- 최댓값을 구하는 데 컴퓨터가 해야 하는 가장 중요한 계산은 두 값 중 어느 것이 더 큰지를 판단하는 '비교'
- 프로그램 2-1의 `for i in range(1, n):` 반복문 안의(`if a[i] > max_v:`)이 있어 자료  $n$ 개 중에서 최댓값을 찾으려면 비교를  $n-1$ 번 해야 함
- 계산 복잡도는  $O(n-1)$ 이 아닌  $O(n)$
- $n$ 이 굉장히 커질 때는  $n$ 번과  $n-1$ 번의 차이가 무의미
- 계산 복잡도  $O(n)$ 의 가장 중요한 특징은 입력 크기와 계산 시간이 대체로 비례한다는 것

리스트에 숫자가  $n$ 개 있을 때 가장 큰 값이 있는 위치 번호를 돌려주는 알고리즘을 만들어 보세요.

- 최댓값이 리스트의 어느 위치에 있는지 묻는 문제

### 4 응용하기

- 리스트 [17, 92, 18, 33, 58, 7, 33, 42]에서는 두 번째 나오는 값인 92가 최댓값  
→ 0번 위치가 17, 1번 위치가 92이므로 이 문제의 답은 1
- 파이썬에서 리스트의 위치 번호는 0부터 시작

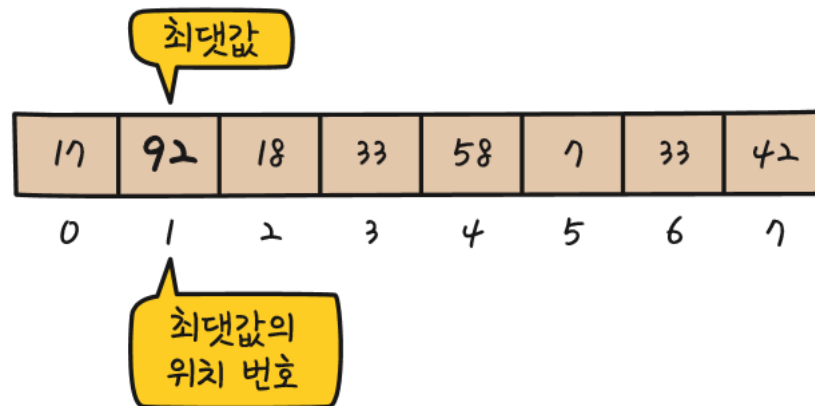


그림 2-2 최댓값과 최댓값의 위치 번호

### 4 응용하기

#### ■ 프로그램 2-2 최댓값의 위치를 구하는 알고리즘

▼ 예제 소스 p02-2-findmaxidx.py

```
# 최댓값의 위치 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중에서 최댓값이 있는 위치(0부터 n-1까지의 값)

def find_max_idx(a):
    n = len(a) # 입력 크기 n
    max_idx = 0 # 리스트 중 0번 위치를 일단 최댓값 위치로 기억
    for i in range(1, n):
        if a[i] > a[max_idx]: # 이번 값이 현재까지 기억된 최댓값보다 크면
            max_idx = i      # 최댓값의 위치를 변경
    return max_idx
```





```
v = [17, 92, 18, 33, 58, 7, 33, 42]  
print(find_max_idx(v))
```

### ■ 실행 결과

1

- $a[\text{최댓값의 위치 번호}] = \text{최댓값}$
- 예시:  $a[1] = 92$

## 연습 문제

- 2-1 숫자  $n$ 개를 리스트로 입력받아 최솟값을 구하는 프로그램을 만들어 보세요.

### ■ 2-1 최솟값 구하기 프로그램

▼ 예제 소스 e02-1-findmin.py

```
# 최솟값 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중 최솟값

def find_min(a):
    n = len(a) # 입력 크기 n
    min_v = a[0] # 리스트 중 첫 번째 값을 일단 최솟값으로 기억
    for i in range(1, n): # 1부터 n-1까지 반복
        if a[i] < min_v: # 이번 값이 현재까지 기억된 최솟값보다 작으면
            min_v = a[i] # 최솟값을 변경
    return min_v
```





## 연습 문제 풀이

부록 A



```
v = [17, 92, 18, 33, 58, 7, 33, 42]  
print(find_min(v))
```

### ■ 실행 결과

7

## 문제 03 동명이인 찾기①

---

## 문제03 동명이인 찾기①

0

n명의 사람 이름 중에서 같은 이름을 찾아 집합으로 만들어 돌려주는 알고리즘을 만들어 보세요.

- 여러 사람의 이름 중에서 같은 이름이 있는지 확인
- 같은 이름이 있다면 같은 이름들을 새로 만든 결과 집합에 넣어 돌려주면 됨
- 이 문제의 입력은 n명의 이름이 들어 있는 리스트
- 결과는 같은 이름들이 들어 있는 집합(set)
- 사람 이름으로 구성된 리스트 ["Tom", "Jerry", "Mike", "Tom"]이 입력되면 Tom이란 이름이 두 번 나오므로 결과는 집합 {"Tom"}이 됨

### 1 집합

- 집합: 리스트와 같이 정보를 여러 개 넣어서 보관할 수 있는 파이썬의 기능
- 집합 하나에는 같은 자료가 중복되어 들어가지 않고, 자료의 순서도 의미가 없다는 점이 리스트와 다름

```
>>> s = set()
>>> s.add(1)
>>> s.add(2)
>>> s.add(2) # 이미 2가 집합에 있으므로 중복해서 들어가지 않습니다.
>>> s
{1, 2}
>>> len(s) # 집합 s에는 자료가 두 개 들어 있습니다.
2
>>> {1, 2} == {2, 1} # 자료의 순서는 무관하므로 {1, 2}와 {2, 1}은 같은 집합입니다.
True
```

1 집합

- 빈 집합을 만들려면 set()를 이용
- 집합에 자료를 추가하려면 add() 함수 이용

함수	설명	사용 예
len(s)	집합의 길이(자료 개수)를 구합니다.	s = set() len(s) # 빈 집합이므로 0 len({1, 2, 3}) # 자료 개수가 세 개이므로 3
add(x)	집합에 자료 x를 추가합니다.	s = {1, 2, 3} s.add(4) # s는 {1, 2, 3, 4}가 됩니다(순서는 다를 수 있음).
discard(x)	집합에 자료 x가 들어 있다면 삭제 합니다(없으면 변화 없음).	s = {1, 2, 3} s.discard(2) # s는 {1, 3}이 됩니다.
clear()	집합의 모든 자료를 지웁니다.	s = {1, 2, 3} s.clear() # s = set(), 즉 빈 집합이 됩니다.
x in s	어떤 자료 x가 집합 s에 들어 있는지 확인합니다(x not in s는 반대 결과).	s = {1, 2, 3} 2 in s # 2가 집합 s 안에 있으므로 True 5 in s # 5가 집합 s 안에 없으므로 False 5 not in s # 5가 집합 s 안에 없으므로 True

표 3-1 자주 쓰는 집합 기능

## 2 동명이인을 찾는 알고리즘

- 동명이인을 찾는 알고리즘(사람의 생각)
  - 1) 첫 번째 Tom을 뒤에 있는 Jerry, Mike, Tom과 차례로 비교
  - 2) 첫 번째 Tom과 마지막 Tom이 같으므로 동명이인(동명이인: Tom)
  - 3) 두 번째 Jerry를 뒤에 있는 Mike, Tom과 비교(앞의 Tom과는 이미 비교함)
  - 4) 세 번째 Mike를 뒤에 있는 Tom과 비교
  - 5) 마지막 Tom은 비교하지 않아도 됨(이미 앞에서 비교함)
  - 6) 같은 이름은 Tom 하나뿐

2 동명이인을 찾는 알고리즘

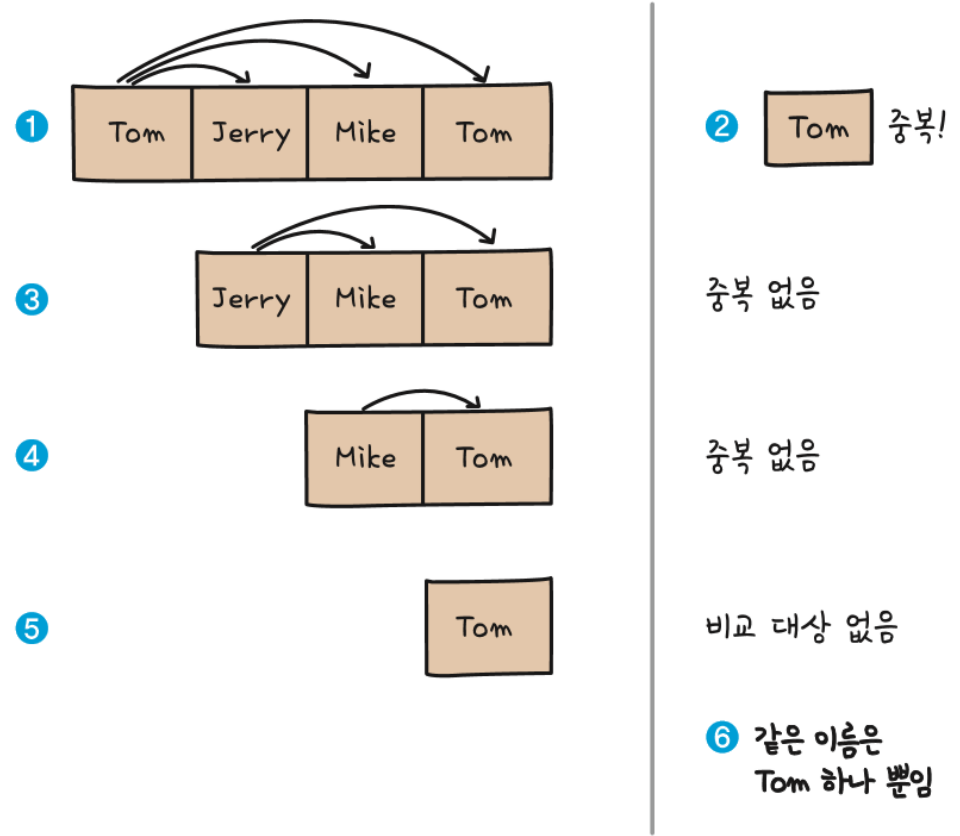


그림 3-1 동명이인을 찾는 과정

## 2 동명이인을 찾는 알고리즘

### 이 알고리즘에서 주의할 점

1. 이번에 비교할 이름을 뽑은 다음에는 뽑은 이름보다 순서상 뒤에 있는 이름하고만 비교하면 됨  
→ 자기 자신과 비교하는 것은 무의미, 앞의 이름과는 이미 비교가 끝났음
2. 리스트의 마지막 이름을 기준으로 비교하지 않아도 됨  
→ 자신의 뒤에는 비교할 이름이 없고, 앞과는 이미 비교가 끝났음
3. 같은 이름을 찾으면 결과 집합에 그 이름을 추가



## 2 동명이인을 찾는 알고리즘

### ■ 프로그램 3-1 동명이인을 찾는 알고리즘

▼ 예제 소스 p03-1-samename.py

```
# 두 번 이상 나온 이름 찾기
# 입력: 이름이 n개 들어 있는 리스트
# 출력: 이름 n개 중 반복되는 이름의 집합

def find_same_name(a):
    n = len(a) # 리스트의 자료 개수를 n에 저장
    result = set() # 결과를 저장할 빈 집합
    for i in range(0, n - 1): # 0부터 n-2까지 반복
        for j in range(i + 1, n): # i+1부터 n-1까지 반복
            if a[i] == a[j]: # 이름이 같으면
                result.add(a[i]) # 찾은 이름을 result에 추가
    return result
```



## 2 동명이인을 찾는 알고리즘



```
name = ["Tom", "Jerry", "Mike", "Tom"] # 대소문자 유의: 파이썬은 대소문자를 구분함
print(find_same_name(name))

name2 = ["Tom", "Jerry", "Mike", "Tom", "Mike"]
print(find_same_name(name2))
```

### ■ 실행 결과

```
{'Tom'}
{'Mike', 'Tom'}
```

- 자료의 출력 순서는 중요하지 않음 → {'Tom', 'Mike'} 출력도 틀린 것 아님

## 2 동명이인을 찾는 알고리즘

### 중첩된 반복문

- 리스트 안의 자료를 서로 빠짐없이 비교하되 중복해서 비교하지 않게 함
- 첫 번째 반복문 for i in range(0, n - 1):  
→ i를 0부터 n-2까지 반복
- 두 번째 반복문 for j in range(i + 1, n):  
→ 비교 기준으로 정해진 i번째 위치에 1을 더한 위치의 값부터 끝까지 비교

2 동명이인을 찾는 알고리즘

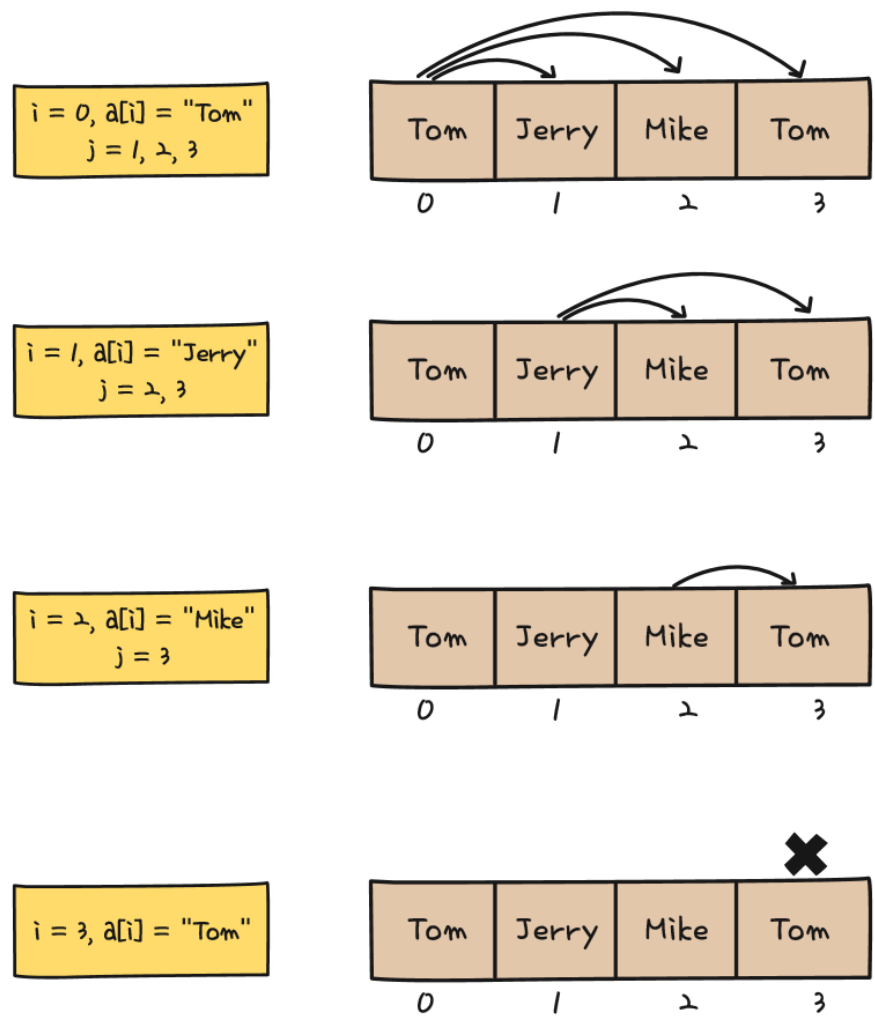


그림 3-2  
리스트에 있는 자료를  
모두 비교하는 방법

3 알고리즘 분석

계산 복잡도 분석

- 두 이름이 같은지 '비교'하는 횟수를 따져 보면 됨

위치	이름	비교 횟수	비교 대상
0	Tom	3	Jerry, Mike, Tom
1	Jerry	2	Mike, Tom
2	Mike	1	Tom
3	Tom	0	비교 안 함
전체 비교 횟수 = 3 + 2 + 1 + 0 = 6			

표 3-2 n = 4일 때 비교 횟수

## 알고리즘 분석

- 일반적인 입력 크기인  $n$ 에 대한 비교 횟수
- 0번 위치 이름:  $n-1$ 번 비교(자기를 제외한 모든 이름과 비교)
- 1번 위치 이름:  $n-2$ 번 비교
- 2번 위치 이름:  $n-3$ 번 비교
- ...
- $n-2$ 번 위치 이름: 1번 비교
- $n-1$ 번 위치 이름: 0번 비교
- 전체 비교 횟수는  $0 + 1 + 2 + 3 + 4 + \dots + (n-1)$ 번, 즉 1부터  $n-1$ 까지의 합

## 3 알고리즘 분석

- 문제 1에서 배운 1부터  $n$ 까지의 합을 구하는 공식에  $n$  대신  $n-1$ 을 대입

$$1 + 2 + 3 + \dots + (n-1) = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

- $\frac{1}{2}n^2 - \frac{1}{2}n$ 번 비교해야 함
- 대문자  $O$  표기법으로는  $O(n^2)$ 이라고 표현( $n^2$  앞 계수  $\frac{1}{2}$ 이나  $-\frac{1}{2}n$ 은 무시)  
→  $n$ 의 제곱에 비례해서 계산 시간이 변하는 것이 핵심
- 계산 복잡도가  $O(n^2)$ 인 알고리즘은 입력 크기  $n$ 이 커지면 계산 시간은  
그 제곱에 비례하므로 엄청난 차이로 증가  
→ 입력 크기가 커질 때 계산 시간이 얼마나 증가할지 가늠해 볼 수 있음

## 연습 문제

■ 3-1 n명 중 두 명을 뽑아 짝을 짓는다고 할 때 짝을 지을 수 있는 모든 조합을 출력하는 알고리즘을 만들어 보세요.

예를 들어 입력이 리스트 ["Tom", "Jerry", "Mike"]라면 다음과 같은 세 가지 경우를 출력합니다.

Tom - Jerry

Tom - Mike

Jerry - Mike



## 연습 문제

■ 3-2 다음 식을 각각 대문자 O 표기법으로 표현해 보세요.

A. 65536

B.  $n - 1$

C.  $\frac{2n^2}{3} + 10000n$

D.  $3n^4 - 4n^3 + 5n^2 - 6n - 7$

### ■ 3-1 두 명을 뽑아 짝으로 만드는 프로그램

▼ 예제 소스 e03-1-pairing.py

```
# n명에서 두 명을 뽑아 짝을 만드는 모든 경우를 찾는 알고리즘  
# 입력: n명의 이름이 들어 있는 리스트  
# 출력: 두 명을 뽑아 만들 수 있는 모든 짝
```

```
def print_pairs(a):  
    n = len(a) # 리스트의 자료 개수를 n에 저장  
    for i in range(0, n - 1): # 0부터 n-2까지 반복  
        for j in range(i + 1, n): # i+1부터 n-1까지 반복  
            print(a[i], "-", a[j])
```



## 문제03 동명이인 찾기①

# 연습 문제 풀이

부록 A



```
name = ["Tom", "Jerry", "Mike"]  
print_pairs(name)  
print()  
name2 = ["Tom", "Jerry", "Mike", "John"]  
print_pairs(name2)
```

## 연습 문제 풀이

부록 A

### ■ 실행 결과

Tom - Jerry

Tom - Mike

Jerry - Mike

Tom - Jerry

Tom - Mike

Tom - John

Jerry - Mike

Jerry - John

Mike - John

- $n$ 명에서 두 명을 뽑아 짝으로 만들면 짝 조합이  $\frac{n(n-1)}{2}$  가지 출력  
→ 이 경우의 수를  ${}_nC_2$ 라고도 표현

## 연습 문제 풀이

부록 A

- 3-2 대문자 O 표기법

A. 65536

$O(1)$

- 65536은 n 값의 변화와 관계가 없음

## 연습 문제 풀이

부록 A

- 3-2 대문자 O 표기법

B.  $n - 1$

$O(n)$

- $n$ 이 굉장히 커지면  $-1$ 은 거의 영향이 없어짐

## 연습 문제 풀이

부록 A

- 3-2 대문자 O 표기법

C.  $\frac{2n^2}{3} + 10000n$

$O(n^2)$

- $n$ 이 굉장히 커지면  $\frac{2n^2}{3}$  과 비교했을 때  $10000n$ 의 영향은 작아짐
- 제곱에 비례한다는 관계가 핵심이므로 계수  $\frac{2}{3}$ 도 생략

## 연습 문제 풀이

부록 A

- 3-2 대문자 O 표기법

D.  $3n^4 - 4n^3 + 5n^2 - 6n - 7$

$O(n^4)$

- n의 변화에 따라 가장 크게 변하는 항을 계수를 생략하여 표현