

# 파이썬으로 배우는 데이터 구조



한동대학교  
전산전자공학부  
김영섭 교수



# 학습 목표

---

프로그램의 성능을 비교하기 위한 방법을 배우고  
예시를 통해 적용할 수 있다

# **Data Structures in Python**

## **Chapter 2 - 2**

- **Performance Analysis**
- Big-O Notation
- Big-O Properties
- Growth Rates
- Growth Rates Examples

그러므로 나의 사랑하는 자들아 너희가 나 있을 때 뿐 아니라 더욱 지금 나 없을 때에도 항상 복종하여 두렵고 떨림으로 너희 구원을 이루라 (Continue to work out your salvation with fear and trembling.) 빌2:12

나는 인애를 원하고 제사를 원하지 아니하며 번제보다 하나님을 아는 것을 원하노라 (호6:6)  
하나님은 모든 사람이 구원을 받으며 진리를 아는데에 이르기를 원하시느니라 (딤후2:4)

그런즉 너희가 먹든지 마시든지 무엇을 하든지 다 하나님의 영광을 위하여 하라 (고전10:31)

# Agenda & Reading

---

- Performance Analysis
  - Introduction
  - Step Counts - Counting Operations
- References:
  - Textbook: Problem Solving with Algorithms and Data Structures
    - Chapter 3. [Analysis](#)
  - Textbook: [www.github.idebtor/DSpy](http://www.github.idebtor/DSpy)
    - Chapter 2.1 ~ 3

# 1 Introduction - What Is Performance Analysis?

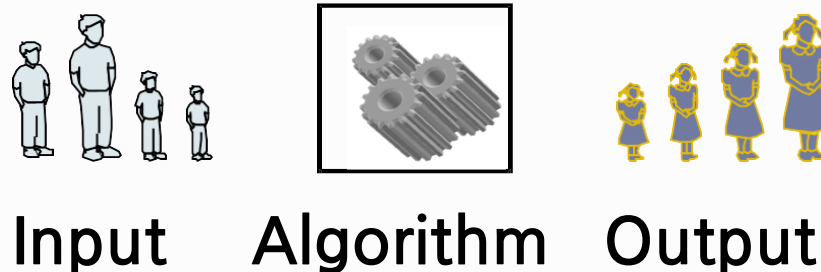
---

- How to compare programs with one another?
- When two programs solve the same problem but look different, is one program better than the other?
- What criteria are we using to compare them?
  - Readability?
  - Efficiency? Time vs. Memory
- Why do we need Performance Analysis or Complexity Analysis?
  - Writing a working program is not good enough.
  - The program may be inefficient!
  - If the program runs on a large data set, then the running time may become an issue.

# 1 Introduction - Data Structures & Algorithm

---

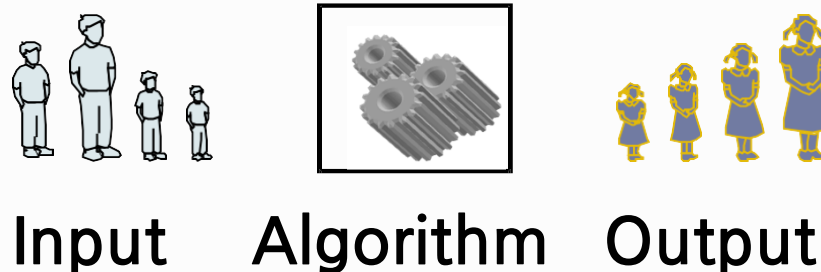
- Data Structures:
  - A systematic way of **organizing** and **accessing** data.
  - No single data structure works well for **ALL** purposes.
- Algorithm
  - An algorithm is **a step-by-step procedure** for solving a problem in a finite amount of time.
- Program
  - A program is an algorithm that has been encoded into some programming language.
- **Program = data structures + algorithms**



# 1 Introduction - Performance Analysis/Complexity

---

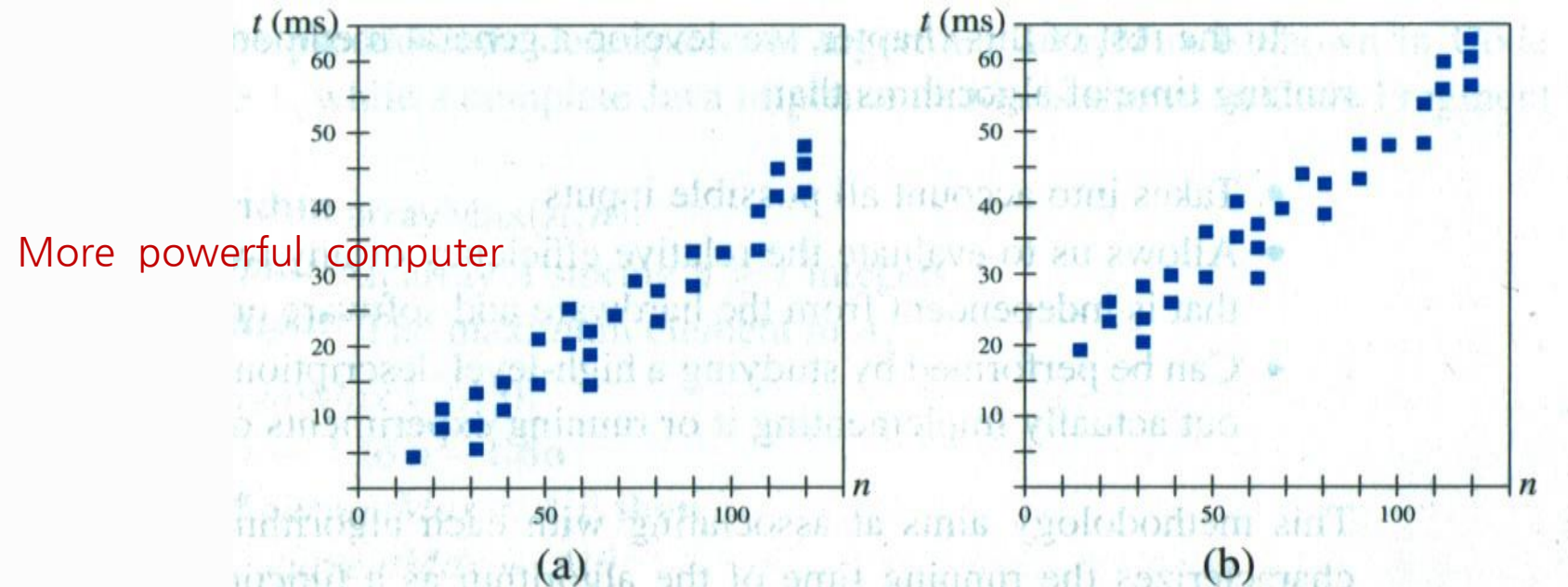
- When we analyze the **performance** of an algorithm, we are interested in how much of a given resource the algorithm uses to solve a problem.
- The most common resources are **time** (how many **steps** it takes to solve a problem) and **space** (how much memory it takes).
- We are going to be mainly interested in **how long** our programs take to run, as time is generally more precious resource than **space**.





# 1 Introduction - Efficiency of Algorithms

- For example, the following graphs show the execution time, in milliseconds, against sample size,  $n$  of a given problem in **different computers**



- The actual running time of a program depends **not only** on the efficiency of the algorithm, **but** on many other variables such as Processor speed & type, Operating system,  $\cdots$  etc.

# 1 Introduction - Running-time of Algorithms

---

- In order to compare algorithm speeds experimentally
  - All other variables must be kept constant, i.e.
    - independent of specific implementations (C, C++ or Java),
    - independent of computers used, and,
    - independent of the data on which the program runs
  - Involved a lot of work (better to have some theoretical means of predicting algorithm speed)

# 1 Introduction - Example 1

---

- Task:
  - Complete the `sum_of_n()` function which calculates the sum of the first `n` natural numbers.
    - **Arguments:** an integer
    - **Returns:** the sum of the first `n` natural numbers
- Cases:

`sum_of_n(5)`

15

`sum_of_n(100_000)`

5000050000

# 1 Introduction - Algorithm 1

- `sum_of_n`

```
time_start = time.time()

sum = 0
for i in range(1,n+1):
    sum = sum + i

time_end = time.time()
time_taken = time_end - time_start
```

The timing calls embedded before and after the summation to calculate the time required for the calculation.

Set sum = 0

Add each value to sum  
using a for loop

Return sum

# 1 Introduction - Algorithm 2

- `sum_of_n_2`

```
time_start = time.time()

sum = 0
sum = n * (n + 1) / 2

time_end = time.time()
time_taken = time_end - time_start
```

The timing calls embedded before and after the summation to calculate the time required for the calculation.

Set `sum = 0`

Use the equation  $(n(n + 1))/2$ ,  
to calculate the total

Return `sum`

# 1 Introduction - Experimental Result

- Using 4 different values for n: [10000, 100000, 1000000, 10000000]

| n        | sum_of_n<br>(for loop) | sum_of_n_2<br>(equation) |
|----------|------------------------|--------------------------|
| 10000    | 0.0033                 | 0.00000181               |
| 100000   | 0.0291                 | 0.00000131               |
| 1000000  | 0.3045                 | 0.00000107               |
| 10000000 | 2.7145                 | 0.00000123               |

Time Consuming Process!



Time increase as we increase the value of n.

NO impacted by the number of integers being added.

- We shall **count** the number of basic operations of an algorithm and **generalize** the count.

## 2 Counting Operations - Example 2A

- Example: Calculating the sum of the first 10 elements in the list

```
def count1(numbers):  
    sum = 0  
    index = 0  
    while index < 10:  
        sum = sum + numbers[index]  
        index += 1  
    return sum
```

1 assignment  
1 assignment  
11 comparisons  
10 plus/assignments  
10 plus/assignments  
1 return

- Total: 34 operations (**steps**)

## 2 Counting Operations - Example 2B

- Example: Calculating **the sum of n elements** in the list.

```
def count2(numbers):  
    n = len(numbers)  
    sum = 0  
    index = 0  
    while index < n:  
        sum = sum + numbers[index]  
        index += 1  
    return sum
```

```
1 assignment  
1 assignment  
1 assignment  
n + 1 comparisons  
n plus/assignments  
n plus/assignments  
1 return
```

- Total:  **$3n + 5$**  operations (steps)
- We need to measure an algorithm's time requirement as **a function of the problem size**, e.g., in the example above the problem size is the number of elements in the list.



## 2 Counting Operations - Problem size

---

- Performance is usually measured by the **rate** at which the running time increases as the problem size gets bigger,
  - i.e., we are interested in the relationship between the **running time** and the **problem size**.
  - It is very important that **we identify what the problem size is**.
    - For example, if we are analyzing an algorithm that processes a list, the problem size is the **size** of the list.
- In many cases, the problem size will be the **value** of a variable, where the running time of the program depends on how big that value is.

## 2 Counting Operations - Exercise 1

---

- How many operations are required to do the following tasks?
  - Adding an element to the end of a list
  - Printing each element of a list containing  $n$  elements

## 2 Counting Operations - Exercise 1 solution

---

- How many operations are required to do the following tasks?
  - Adding an element to the end of a list - one operation, a constant time, or  $O(1)$
  - Printing each element of a list containing  $n$  elements -  $n$  operation, or  $O(n)$

## 2 Counting Operations - Example 3

- Consider the following two algorithms:

- Algorithm A:

- Outer Loop:  $n$  operations
- Inner Loop:  $\frac{n}{5}$  operations
- Total:  $(n * \frac{n}{5}) = \frac{n^2}{5}$  operations

```
for i in range(0, n):  
    for j in range(0, n, 5):  
        print(i, j)
```

- Algorithm B:

- Outer Loop:  $n$  operations
- Inner Loop: 5 operations
- Total:  $(n * 5) = 5 * n$  operations

```
for i in range(0, n):  
    for j in range(0, 5):  
        print(i, j)
```

## 2 Counting Operations - Growth Rate Function - A or B?

- Consider the following two algorithms:

- Algorithm A:  $\frac{n^2}{5}$
- Algorithm B:  $5 * n$

| n | 5  | 10 | 15 | 20  | 24  | 25  | 26  | 30  |
|---|----|----|----|-----|-----|-----|-----|-----|
| A | 5  | 20 | 45 | 80  | 115 | 125 | 135 | 180 |
| B | 25 | 50 | 75 | 100 | 120 | 125 | 130 | 150 |

- If  $n$  is  $10^6$ ,

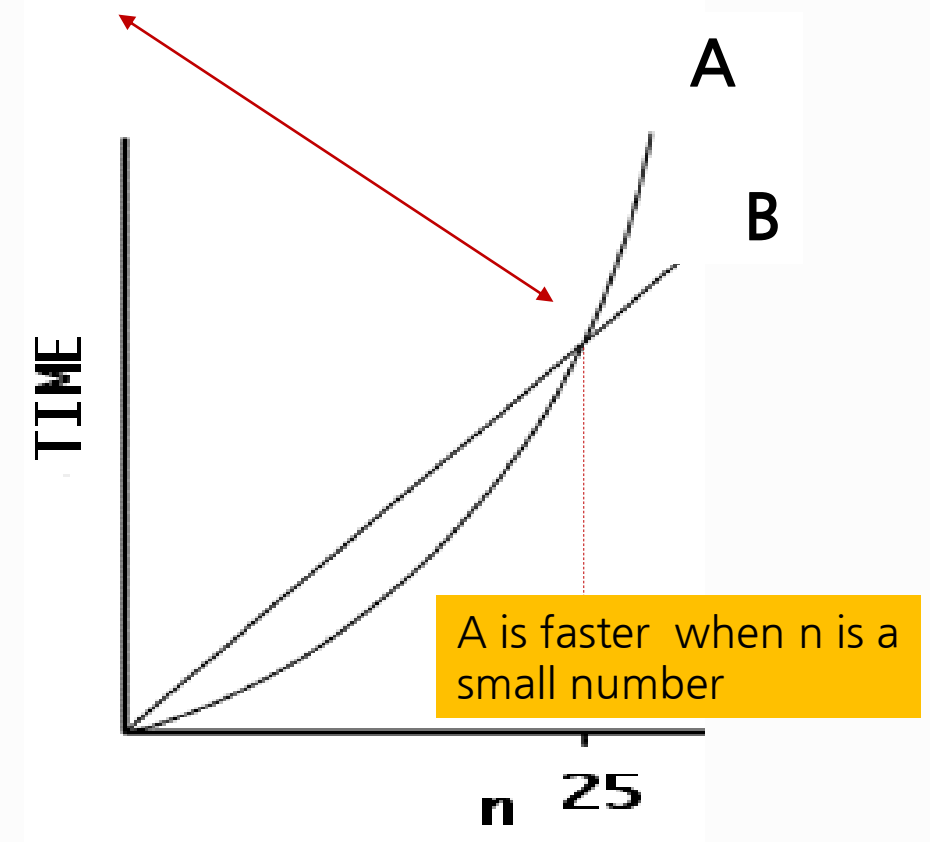
- Algorithm A's time requirement is

- $\frac{n^2}{5} = \frac{10^{12}}{5} = 2 * 10^{11}$

- Algorithm B's time requirement is

- $5 * n = 5 * 10^6$

- What does the **growth rate** tell us about the running time of the program?



## 2 Counting Operations - Growth Rate Function - A or B?

---

- **For smaller values of  $n$** , the differences between algorithm A ( $n^2/5$ ) and algorithm B ( $5n$ ) are not very big. But the differences are very evident for larger problem sizes such as for  $n > 1,000,000$
- $2 * 10^{11}$  vs.  $2 * 10^6$
- **Bigger** problem size produces **bigger** differences
- **Algorithm efficiency is a concern for large problem sizes**

# Step Count Exercise Warming-up:

---

◆ What is **the exact number of times** `sum += 1` executed?

```
def step(n):  
    sum = 0  
    for i in range(n*n):  
        sum += 1  
    return sum  
  
if __name__ == '__main__':  
    print(step(10))
```

## Useful formulas:

$$1 + 2 + 3 + \dots + N = N(N+1)/2$$

$$1 + 2 + 4 + 8 + \dots + 2^n = 2^{n+1} - 1$$

# Step Count Exercise 1:

---

◆ What is **the exact number of times** `sum += 1` executed?

```
def step(n):  
    sum = 0  
    for i in range(n*n):  
        for j in range(i+1):  
            sum += 1  
    return sum  
  
if __name__ == '__main__':  
    print(step(10))
```

## Useful formulas:

$$1 + 2 + 3 + \dots + N = N(N+1)/2$$

$$1 + 2 + 4 + 8 + \dots + 2^n = 2^{n+1} - 1$$



## Step Count Exercise 2:

---

◆ What is **the exact number of times** `sum += 1` executed?

```
def step(n):  
    sum = 0  
    while n > 1:  
        sum += 1  
        n /= 2  
    return sum  
  
if __name__ == '__main__':  
    print(step(128))
```

## Step Count Exercise 2:

◆ What is **the exact number of times** `sum += 1` executed?

```
def step(n):  
    sum = 0  
    while n > 1:  
        sum += 1  
        n /= 2  
    return sum  
  
if __name__ == '__main__':  
    print(step(128))
```

We have to find the smallest  $k$  such that  $n / 2^k = 1$

$$\begin{aligned}n / 2^k &= 1 \\n &= 2^k \\ \log(n) &= \log(2^k) \\ \log(n) &= k\end{aligned}$$

## Step Count Exercise 3:

◇ Compute the following series:

a)  $1 + 2 + 3 + \dots + 9 + 10 =$

b)  $1 + 2 + 3 + \dots + (N - 1) + N =$

c)  $1 + 2 + 4 + \dots + 16 =$

Compute the following series and express the result in term of N but without log expression.

(Hint:  $N = 2^{\log N}$ )

Then use the result and to compute the series shown above in c):

d)  $1 + 2 + 4 + \dots + N =$

**Useful formulas:**

$$1 + 2 + 3 + \dots + N = N(N+1)/2$$

$$1 + 2 + 4 + 8 + \dots + 2^n = 2^{n+1} - 1$$

## Summary

---

- Performance Analysis measure an algorithm's time requirement **as a function of the problem size  $n$**  by **using a growth-rate** function.
- It is an **implementation-independent** (including hardware and coded language) way of measuring an algorithm.
- Performance(Complexity) analysis focuses on **large** problems.

# 학습 정리

- 1) 프로그램 및 알고리즘의 성능 평가는 시간(time)과 공간(memory, storage)의 측면에서 성능을 비교할 수 있다
- 2) 문제 크기에 따른 변화율(growth rate)로 알고리즘의 성능을 비교할 수 있다

# 파이썬으로 배우는 데이터 구조

수고했습니다  
곧 다음 시간에  
다시 뵙겠습니다

