Data Structures in Python
Chapter 2

1. Abstract Data Type(ADT)
2. Performance Analysis
3. **Big-O Notation**
4. Growth Rates

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 (딤전2:4)

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

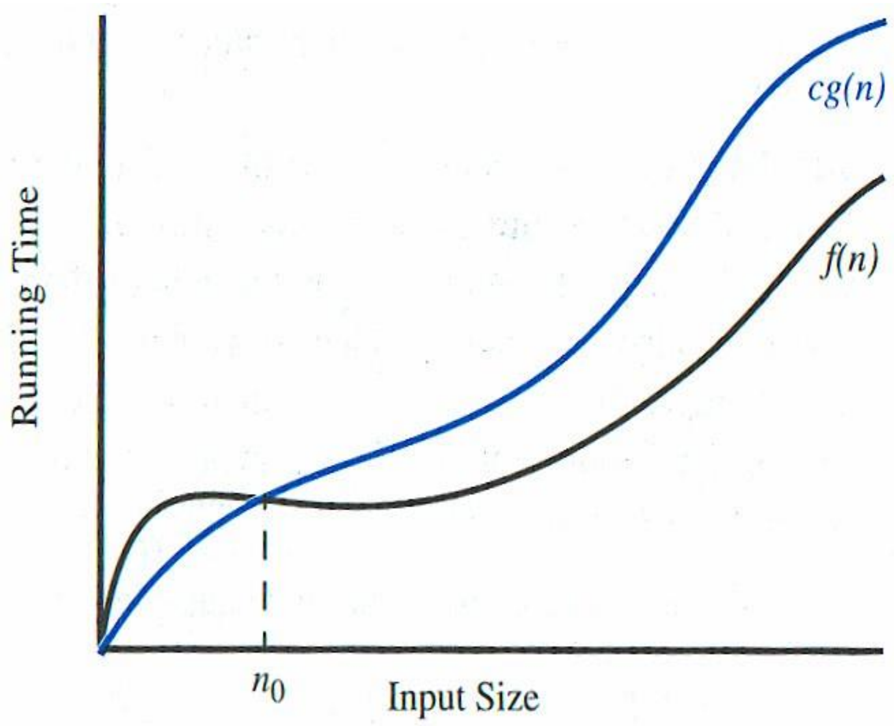
Agenda & Reading

- Performance Analysis
  - Introduction
  - Step Counts - Counting Operations
- **Big-O Notation -** Asymptotic Analysis
  - **Properties of Big-O**
  - **Calculating Big-O**
- Growth Rates
  - Comparison of Growth Rates
  - Big-O Performance of Python Lists
  - Big-O Performance of Python Dictionaries
- References:
  - Textbook: Problem Solving with Algorithms and Data Structures
    - Chapter 3. Analysis
  - Textbook: www.github.idebtor/DSpy
    - Chapter 2.1 ~ 3

3 Big-O Definition

- Let $f(n)$ and $g(n)$ be functions that map non-negative integers to real numbers. We say that $\boldsymbol{f(n)}$ **is** $\boldsymbol{O(g(n))}$ if there is a real constant c, where $c > 0$ and an integer constant n, where $n_0 \geq 1$ such that $f(n) \leq c * g(n)$ for every integer $n \geq n_0$.

  - $f(n)$ describe the actual time of the program
  - $g(n)$ is a much simpler function than $f(n)$
  - With assumptions and approximations, we can use $g(n)$ to describe the complexity i.e., $\boldsymbol{O(g(n))}$



Big-O Notation is a mathematical formula that best describes an algorithm's performance.

3 Big-O Notation

- We use Big-O notation (capital letter O) to specify the order of complexity of an algorithm.

  - $e.g., \ O(n^2), \ O(n^3), \ O(n)$

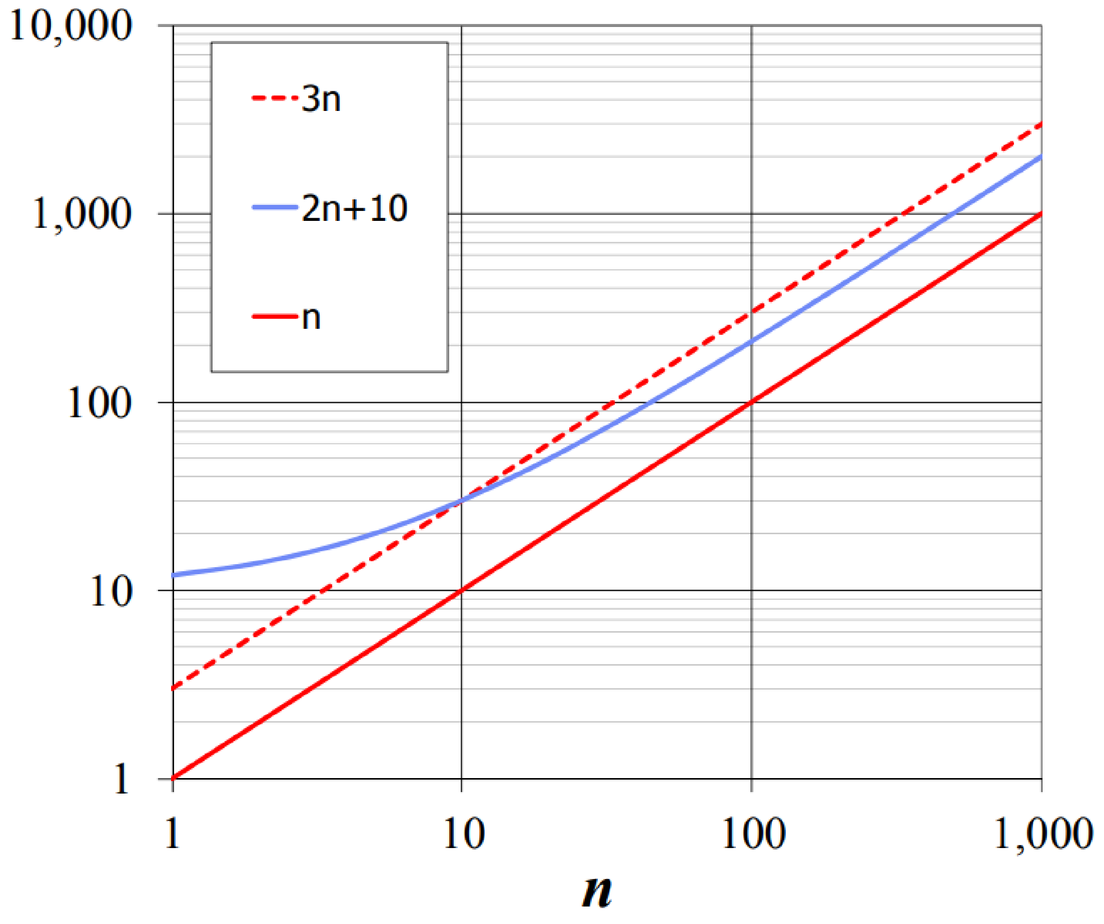
  - If a problem of size n requires time that is directly proportional to n, the problem is $O(n)$ - that is, order n.
  - If the time requirement is directly proportional to n^2, the problem is $O(n^2)$, etc.

3 Big-O Examples

- Given functions $f(n)$ and $g(n)$, we say that **$f(n)$ is $O(g(n))$** if there are positive constants, c, and n_0 such that $f(n) \leq c * g(n)$ for every integer $n \geq n_0$.

- Example:
  T(n) = 2n + 10
  T(n) is O(n)

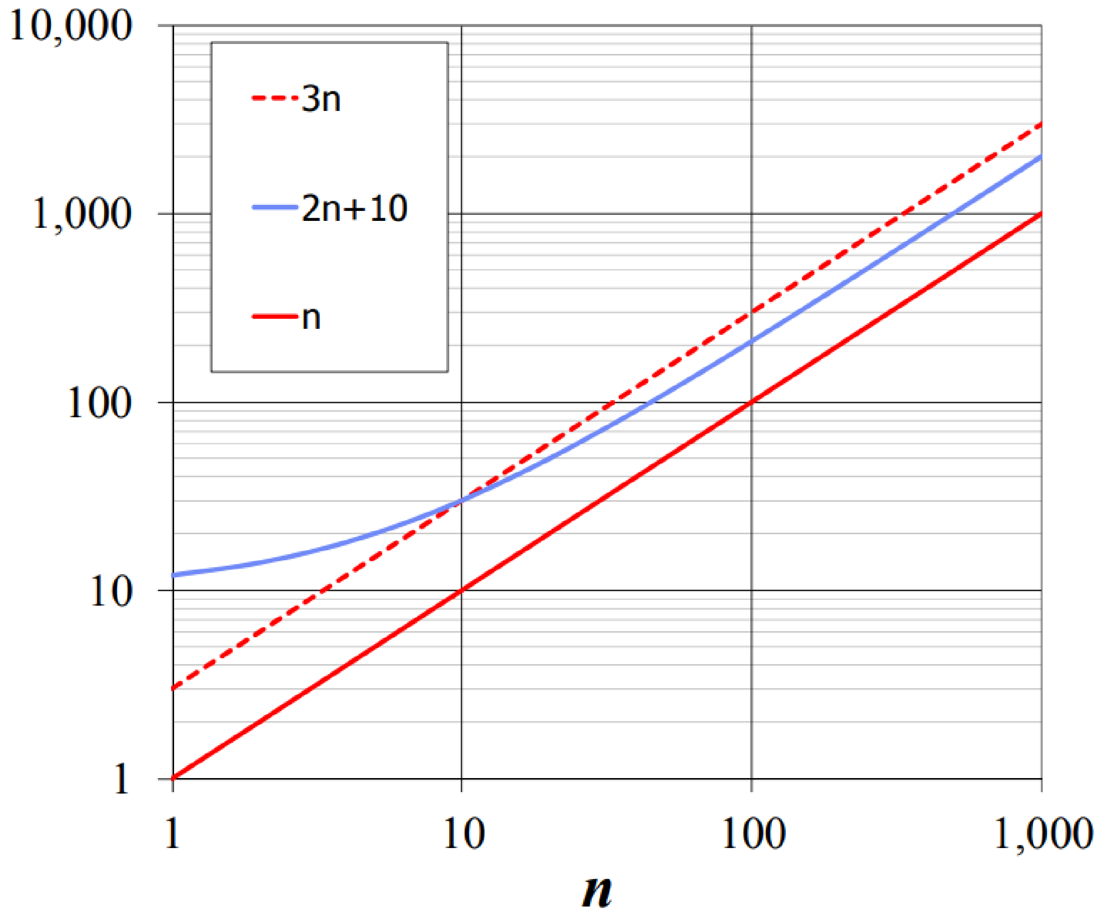
- Question:



3 Big-O Examples

- Given functions $f(n)$ and $g(n)$, we say that $\boldsymbol{f(n)}$ **is** $\boldsymbol{O(g(n))}$ if there are positive constants, c, and n_0 such that $f(n) \leq c * g(n)$ for every integer $n \geq n_0$.

- Example:
  T(n) = 2n + 10
  T(n) is O(n)



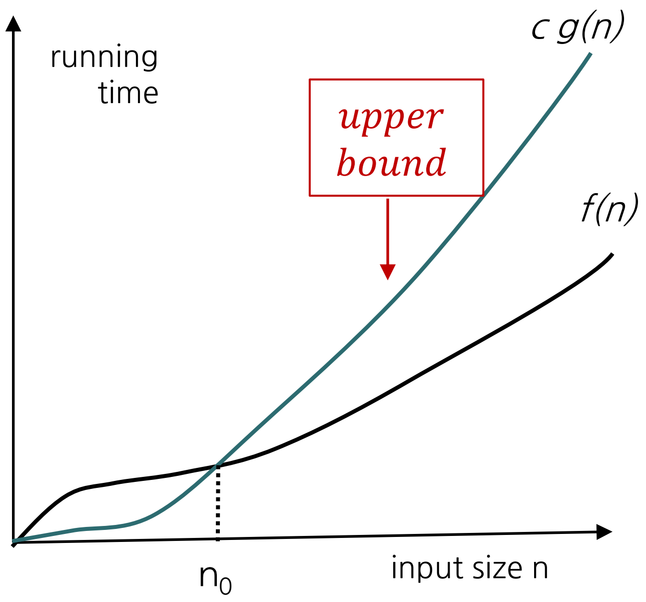
- Question:
  - n_0
  - c
  - g(n)
  - $f(n) \leq c * g(n)$
  - $\boldsymbol{f(n)}$ **is** $\boldsymbol{O(g(n))}$

3 Big-O Examples

- Find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ to justify that the function $7\boldsymbol{n} + 5$ **is** $\boldsymbol{O(n)}$.

We must find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$7n + 5 \leq c\,n \qquad for\ n \geq \boldsymbol{n_0}$$



3 Big-O Examples

- Find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ to justify that the function $7\boldsymbol{n} + 5$ is $\boldsymbol{O}(\boldsymbol{n})$.

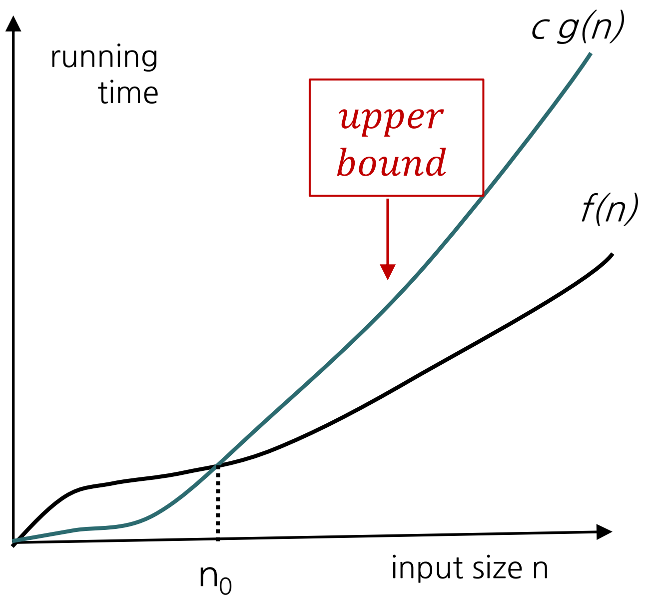
We must find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$7n + 5 \le c\,n \qquad for\ n \ge \boldsymbol{n_0}$$

$$7n + 5 \le 7\,n + n$$

$$7n + 5 \le 8\,n \qquad for\ n \ge \boldsymbol{n_0} = 5$$

Therefore, $7n + 5 \le c\,n$ for $c = 8$ and $\boldsymbol{n_0} = 5$, $g(n) = n$ and $O(n)$



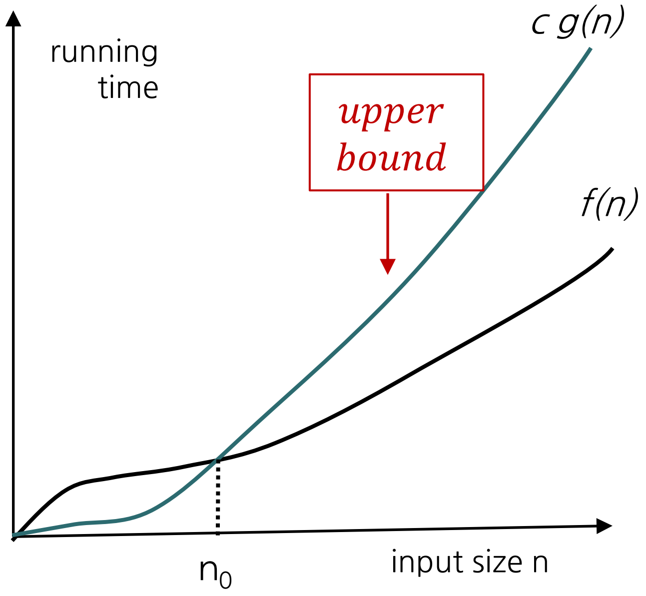
3 Big-O Examples

- Find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ to justify that the function $7\boldsymbol{n} + 5$ is $\boldsymbol{O}(\boldsymbol{n})$.

We must find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$7n + 5 \le c\,n \qquad for\ n \ge \boldsymbol{n_0}$$
$$7n + 5 \le 7\,n + n$$
$$7n + 5 \le 8\,n \qquad for\ n \ge \boldsymbol{n_0} = 5$$

Therefore, $7n + 5 \le c\,n$ for $c = 8$ and $\boldsymbol{n_0} = 5$, $f(n)$ is $O(n)$



$7n + 5 \le c\,n \qquad for\ n \ge \boldsymbol{n_0}$

$7n + 5 \le 12\,n \qquad for\ n \ge \boldsymbol{n_0} = 1$

Therefore, $7n + 5 \le c\,n$ for $c = 12$ and $\boldsymbol{n_0} = 1$

$g(n) = n$, $f(n)$ is $O(n)$

3 Big-O Examples

- Find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ to justify that the function $\boldsymbol{f(n) = 27n^2 + 16n}$ is $\boldsymbol{O(n^2)}$.

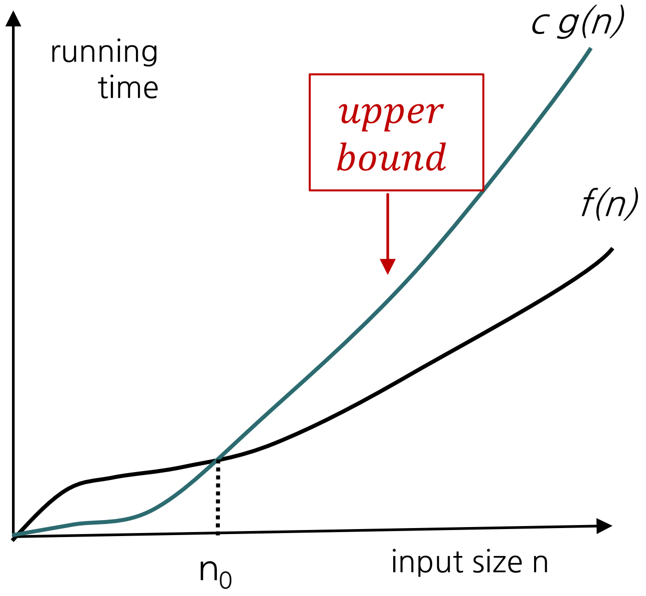
We must find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

For $\boldsymbol{16n \le n^2}$

$$\boldsymbol{27n^2 + 16n \le 27n^2 + n^2}$$

$$\boldsymbol{27n^2 + 16n \le 28n^2} \qquad for\ n \ge \boldsymbol{n_0} = 16$$

Hence, c = 28 and $\boldsymbol{n_0}$ = 16, Therefore, $\boldsymbol{g(n) = n^2, f(n)\ is\ O(n^2).}$



$\boldsymbol{27n^2 + 16n}$ is $\boldsymbol{O(n^2)}$, we must find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$\boldsymbol{27n^2 + 16n \le 43n^2}$$

$$\boldsymbol{27n^2 + 16n \le 43n^2} \qquad for\ n \ge \boldsymbol{n_0} = 1$$

Hence, c = 43 and $\boldsymbol{n_0}$ = 1, Therefore, $\boldsymbol{g(n) = n^2, f(n)\ is\ O(n^2).}$

3 Big-O Examples

$f(n) \leq c * g(n)$ for every integer $n \geq n_0$

- Suppose an algorithm requires
  - T(n) = 7n-2 operations to solve a problem of size n

```
7n-2 ≤ 7 * n for all n0 ≥ 1
i.e., c = 7, n0 = 1
```

O(n)

  - T(n) = n^2- 3 $*$ n + 10 operations to solve a problem of size n

```
n² - 3 * n + 10 < 3 * n² for all n0 ≥ 2
i.e., c = 3, n0 = 2
```

$O(n^2)$

  - T(n) = $3n^3 + 20n^2 + 5$ operations to solve a problem of size n

```
3n³ + 20n² + 5 < 4 * n³ for all n0 ≥ 21
i.e., c = 4, n0 = 21
```

$O(n^3)$

3 Big-O Examples

1) $3n+2 =$
2) $3n+3 =$
3) $100n+6 =$
4) $10n^2+4n+2 =$
5) $6*2^n+n^2 =$
6) $3n+3 =$
7) $10n^2+4n+2 =$
8) $3n+2 \neq O(1)$ *as* $3n + 2$ *is* ***not*** $\leq c$ *for any* c *and all* $n, n \geq n_0$.
9) $10n^2+4n+2 \neq O(n)$

4 Properties of Big-O

- There are three properties of Big-O

  - Ignore low order terms in the function (smaller terms)
    - $O(f(n)) + O(g(n)) = O(\max of\ f(n) and\ g(n))$

  - Ignore any constants in the high-order term of the function
    - $C * O(f(n)) = O(f(n))$

  - Combine growth-rate functions
    - $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
    - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

4 Properties of Big-O - Ignore low order terms

- Consider the function: $f(n) = n^2 + 100n + \log 10n + 1000$
  - For small values of n the last term, 1000, dominates.
  - When n is around 10, the terms 100n + 1000 dominate.
  - When n is around 100, the terms n^2 and 100n dominate.
  - When n gets much larger than 100, the n^2 dominates all others.
- So, it would be safe to say that this function is O(n^2) for values of $n > 100$

- Consider another function: $f(n) = n^3 + n^2 + n + 5000$
  - Big-O is O(n^3)

- And consider another function: $f(n) = n + n^2 + 5000$
  - Big-O is O(n^2)

4 Properties of Big-O - Ignore any Constant Multiplications

- Consider the function:
  - `f(n) = 254 * n² + n`
  - Big-O is $O(n^2)$

- Consider the function:
  - `f(n) = n / 30`
  - Big-O is O(n)

- And consider another function:
  - `f(n) = 3n + 1000`
  - Big-O is O(n)

4 Properties of Big-O - Combine growth-rate functions

- Consider the function:
  - `f(n) = n * log n`
  - Big-O is O(n log n)

- Consider another function:
  - **f(n) = n² * n** → $f(n) = n^2 * n$
  - Big-O is $O(n^3)$

4 Properties of Big-O - Exercise 2

- What is the Big-O performance of the following growth functions?
  - **T(n) = n + log(n)**
  - **T(n) =** n^4 **+ n*log(n) + 300** n^3
  - **T(n) = 300n + 60 * n * log(n) + 342**

4 Properties of Big-O - Exercise 2

- What is the Big-O performance of the following growth functions?
  - `T(n) = n + log(n)` O(n)
  - `T(n) =` n^4 `+ n*log(n) + 300` n^3 O(n^4)
  - `T(n) = 300n + 60 * n * log(n) + 342` O(n log n)

5 Calculating Big-O

- We will investigate rules for finding out the time complexity of a piece of code
  - Straight-line code
  - Loops
  - Nested Loops
  - Consecutive statements
  - If-then-else statements
  - Logarithmic complexity

5 Calculating Big-O - Rules

- Rule 1: Straight-line code
  - Big-O = Constant time O(1)
  - Does not vary with the size of the input
  - Example:
    - Assigning a value to a variable
    - Performing an arithmetic operation.
    - Indexing a list element

```
x = a + b
i = y[2]
```

- Rule 2: Loops
  - The running time of the statements inside the loop (including tests) times the number of iterations
  - Example:
    - Constant time * n = c * n = O(n)

```
for i in range(n):      ← executed n times
    print(i)            ← constant time
```

5 Calculating Big-O - Rules (con't)

- Rule 3: Nested Loop
  - Analyze inside out. Total running time is the product of the sizes of all the loops.
  - Example:
    - constant ∗ (inner loop: n)∗(outer loop: n)
    - Total time = c ∗ n ∗ n = c∗n^2 = O(n^2)

executed n times

```
for i in range(n):
    for j in range(n):
        k = i + j
```

- Rule 4: : Consecutive statements
  - Add the time complexities of each statement
  - Example:
    - Constant time + n times ∗ constant time
    - $c_0 + c_1 n$
    - Big-O = O(f(n) + g(n))
= O(max(f(n) + g(n)))
= O(n)

```
x = x + 1
for i in range(n):
    m = m + 2;
```

constant time

5 Calculating Big-O - Rules (cont.)

- Rule 5: if-else statement
  - Worst-case running time: the test, plus either the if part or the else part (whichever is the larger).
  - Example:
    - c_0 + Max(c_1, (n * (c_0 + c_0)))
    - Total time = c_0 * n(c_1 + c_2) = O(n)
  - Assumption:
    - The condition can be evaluated in constant time. If it is not, we need to add the time to evaluate the expression.

```
if len(a) != len(b):
    return False
else:
    for index in range(len(a)):
        if a[index] != b[index]:
            return False
```

Test: constant time c_0
True Case: constant time c_1

False Case: executed n times

Another if: constant c_2 + constant c_3

5 Calculating Big-O - Rules (cont.)

- Rule 6: Logarithmic
  - An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by ½)
  - Example:
    - Finding a word in a dictionary of n pages
      - Look at the center point in the dictionary
      - Is word to left or right of center?
      - Repeat process with left or right part of dictionary until the word is found
  - Example:
    - Size: n, n/2, n/4, n/8, n/16, . . . 2, 1
    - If $n = 2^K$, it would be approximately k steps.
      The loop will execute log k in the worst case ($\log_2 n = k$).
      Big-O = O(log n)
    - Note: we don't need to indicate the base.
      The logarithms to different bases differ only by a constant factor.

```
size = n
while size > 1:
    // O(1) stuff
    size = size / 2
```

Exercise

- Example: Running time estimates - empirical analysis
  - Personal computer executes 10^9 compares/second
  - Super-computer executes 10^{13} compares/second

	Selection sort (N^2)			Merge sort **(N $\log_2$ N)**		
N	Million	10 million	Billion	Million	10 million	Billion
PC	16.7 min			instant	0.2 sec	
Super Com	0.1 sec			Instant	Instant	Instant

$log_{10}2 \cong 0.3$
$86{,}400\text{sec}/day$
instant 〈 0.1 sec

Use a reasonable or understandable time units.
Do not say, for example, "3660 days" nor "1220 seconds", but 10.0 years or 20.3 min, respectively.

※ **Bottom line**: Good algorithms are better than supercomputers.

Summary

- Big-O Notation is a mathematical formula that best describes an algorithm's performance.
- Big-O notation is often called the asymptotic notation **(점근적 표기법)** since it uses so-called the **asymptotic analysis (점근적 분석)** approach.
- Normally **we assume worst-case analysis**, unless told otherwise.
- In some cases, it may need to consider the best, worst and/or average performance of an algorithm