파이썬으로 배우는
데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

Big-O의 특성들을 알고 실제 코드에서

Big-O를 적용하는 규칙들을 배운다

Data Structures in Python
Chapter 2 - 2

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

Agenda & Reading

- Big-O Notation
  - Asymptotic Analysis
- **Big-O Properties**
  - **Calculating Big-O**

- References:
  - Textbook: Problem Solving with Algorithms and Data Structures
    - Chapter 3. Analysis
  - Textbook: www.github.idebtor/DSpy
    - Chapter 2.1 ~ 3

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:
  - $f(n) = n^2 * n$
  - 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):
    print(i)
```

executed n times
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

$log_{10}2 \cong 0.3$
$86{,}400 sec/day$
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

학습 정리

1) Big-O는 점근적 분석법(asymptotic analysis)을 사용한다

2) 코드에서는 straight line, loop, logarithm 등을 기준으로 Big-O를 설정한다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다