Data Structures in Python
Chapter 2

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

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

Agenda & Reading

- Performance Analysis
  - Introduction
  - Step Counts - Counting Operations
- Big-O Notation - Asymptotic Analysis
  - Properties of Big-O
  - Calculating Big-O
- **Growth Rate**
  - **Comparison**
  - **Profiling and Prediction**
- Growth Rate Examples
  - Python List & Dictionary
- References:
  - Textbook: Problem Solving with Algorithms and Data Structures
    - Chapter 3. Analysis
  - Textbook: www.github.idebtor/DSpy
    - Chapter 2.1 ~ 3

1 Growth Rate Comparison - Hypothetical Running Time

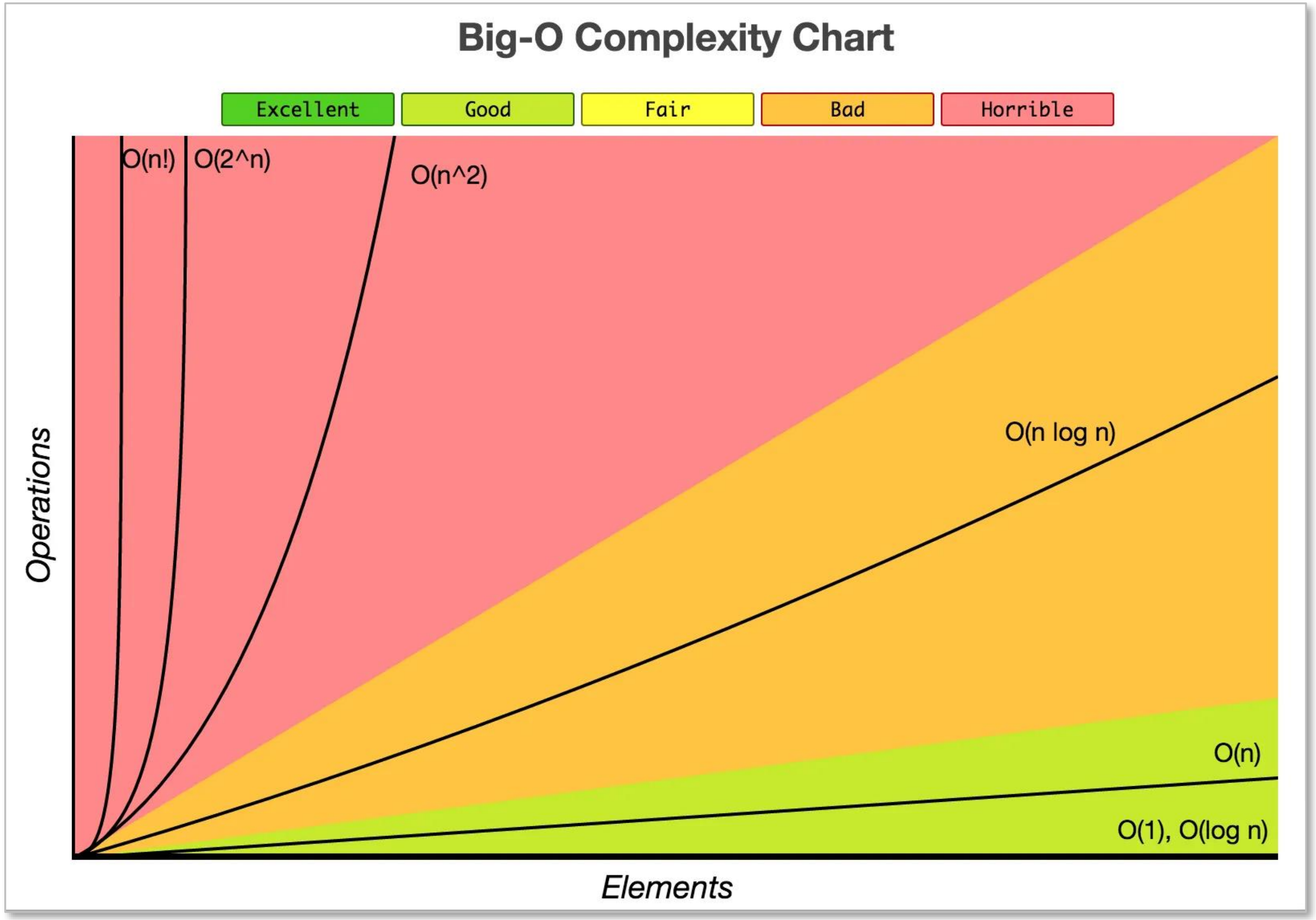
- The running time on a hypothetical computer that computes 10^6operations per second for varies problem sizes

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Notation			**n =** 10	**n =** 10^2	**n =** 10^3	**n =** 10^4	**n =** 10^5	**n =** 10^6
O(1)	Constant	상수	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
O(log(n))	Logarithmic	대수 함수	3 μsec	7 μsec	10 μsec	13 μsec	17 μsec	20 μsec
O(n)	Linear	선형 함수	10 μsec	100 μsec	1 msec	10 msec	100 msec	1 sec
O(nlog(n))	N log N	선형 대수 함수	33 μsec	664 μsec	10 msec	13.3 msec	1.6 sec	20 sec
O(n^2)	Quadratic	2차 함수	100 μsec	10 msec	1 sec	1.7 min	16.7 min	11.6 days
O(n^3)	Cubic	3차 함수	1 msec	1 sec	16.7 min	11.6 days	31.7 years	31709 years
O(2^n)	Exponential	지수 함수	10 msec	3e17 years				

1 Growth Rate Comparison

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

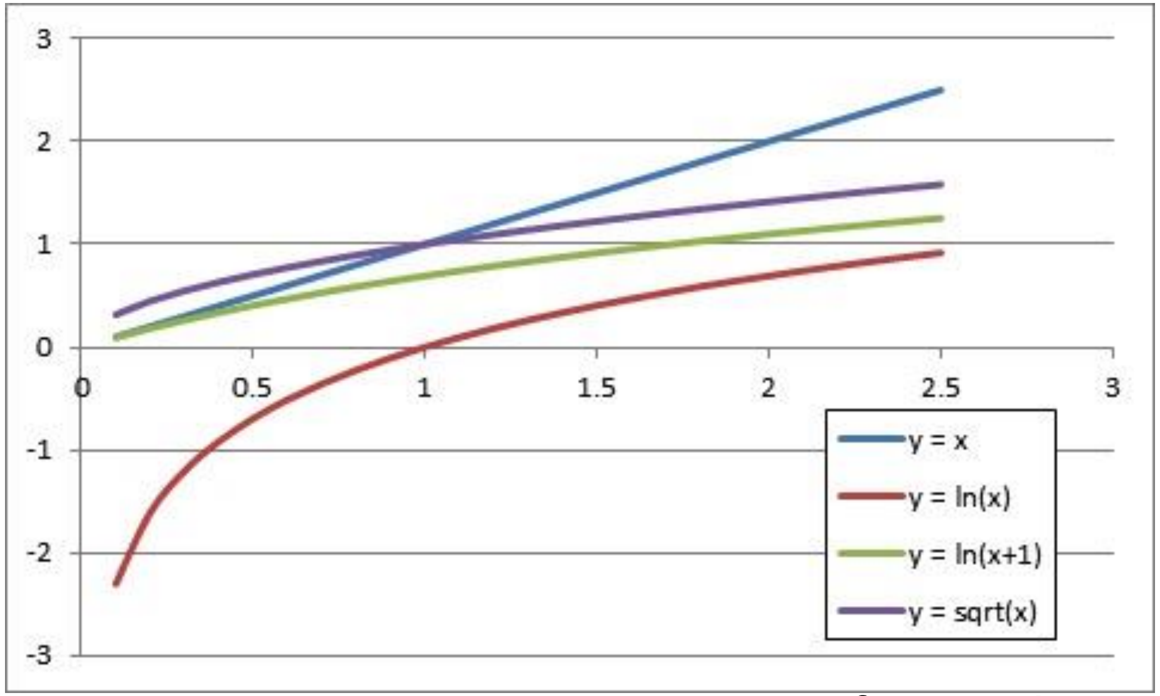


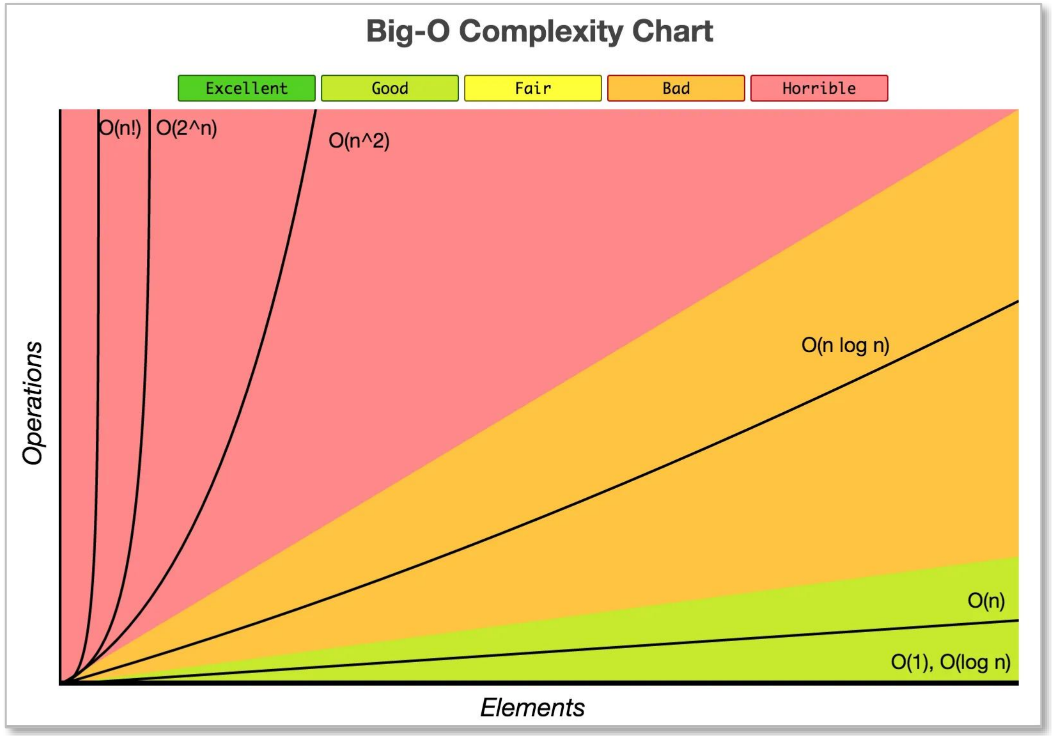
A comparison of growth-rate functions in graphical form

1 Growth Rate Comparison

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$$T(N) \approx aN^b$$



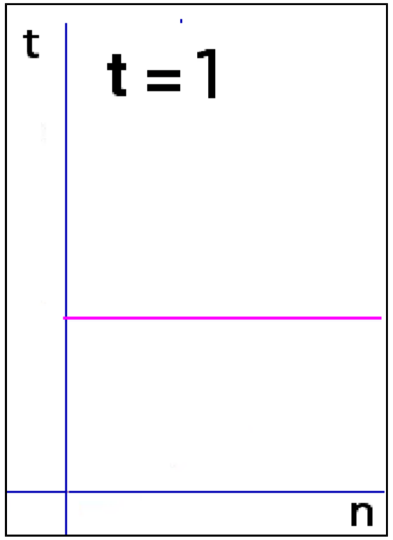


A comparison of growth-rate functions in graphical form

2 Growth Rate Examples - Constant Growth Rate - O(1)

- Time requirement is constant and, therefore, **independent of the problem's size n.**

```
def rate1(n):
    s = "SWEAR"
    for i in range(25):
        print("I must not ", s)
```

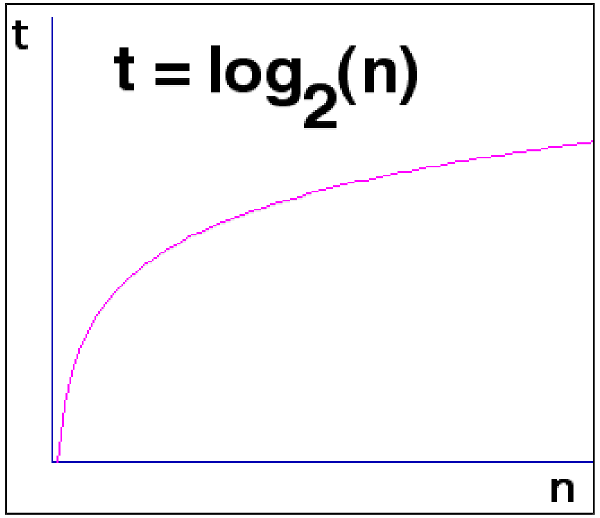


n	10^1	10^2	10^3	10^4	10^5	10^6
O(1)	1	1	1	1	1	1

2 Growth Rate Examples - Logarithmic Growth Rate - O(log n)

- Increase **slowly** as the problem size increases.
- If you square the problem size, you only double its time requirement.
- The base of the log does not affect a log growth rate, so you can omit it.

```
def rate2(n):
    s = "YELL"
    i = 1
    while i < n:
        print("I must not ", s)
        i = i * 2
```

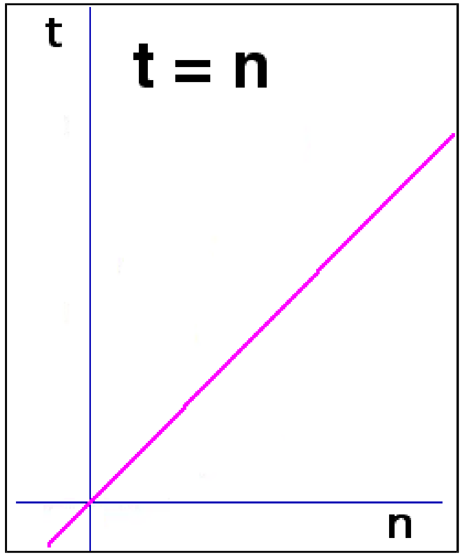


n	10^1	10^2	10^3	10^4	10^5	10^6
$O(\log_2 n)$	3	6	9	13	16	19

2 Growth Rate Examples - Linear Growth Rate - O(n)

- The time increases directly with the sizes of the problem.
- If you square the problem size, you also square its time requirement.

```
def rate3(n):
    s = "FIGHT"
    for i in range(n):
        print("I must not ", s)
```

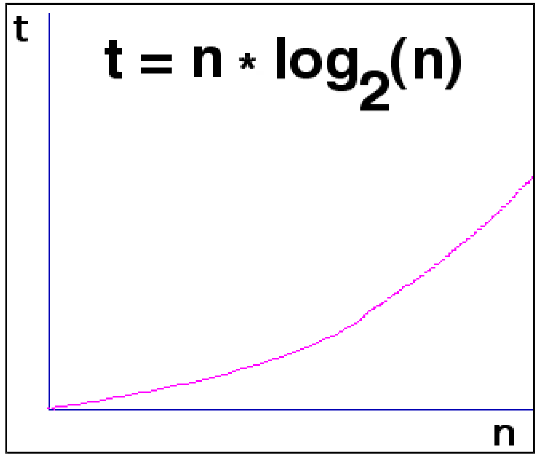


n	10^1	10^2	10^3	10^4	10^5	10^6
O(n)	10	10^2	10^3	10^4	10^5	10^6

2 Growth Rate Examples - n* log n Growth Rate - O(n log(n))

- The time requirement increases more rapidly than a linear algorithm.
- Such algorithms usually divide a problem into smaller problem that are each solved separately.

```
def rate4(n):
    s = "HIT"
    for i in range(n):
        j = n
        while j > 1:
            print("I must not ", s)
            j = j // 2
```

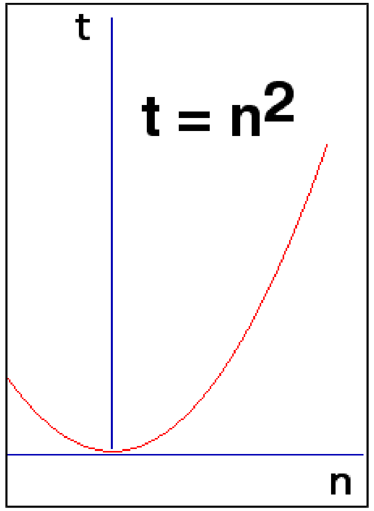


n	10^1	10^2	10^3	10^4	10^5	10^6
O(nlog(n))	30	664	9965	10^5	10^6	10^7

2 Growth Rate Examples - Quadratic Growth Rate - $O(n^2)$

- The time requirement increases rapidly with the size of the problem.
- Algorithms that use two nested loops are often quadratic.

```
def rate5(n):
    s = "LIE"
    for i in range(n):
        for j in range(n):
            print("I must not ", s)
```

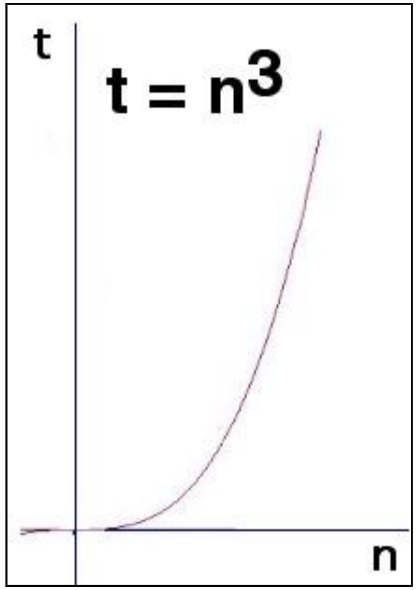


n	10^1	10^2	10^3	10^4	10^5	10^6
$O(n^2)$	10^2	10^4	10^6	10^8	10^{10}	10^{12}

2 Growth Rate Examples - Cubic Growth Rate - $O(n^3)$

- The time requirement increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.
- Algorithms that use three nested loops are often quadratic and are practical only for small problems.

```
def rate6(n):
    s = "SPACE OUT IN CLASS"
    for i in range(n):
        for j in range(n):
            for k in range(n):
                print("I must not ", s)
```

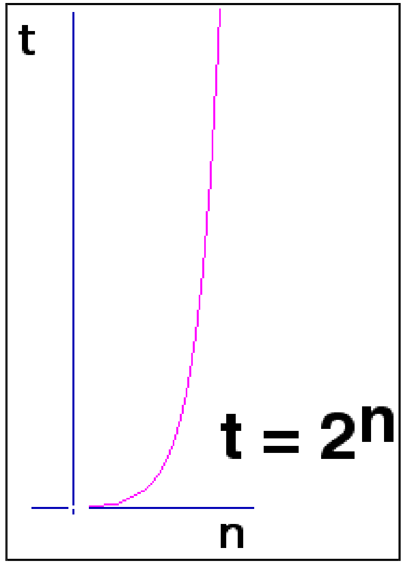


n	10^1	10^2	10^3	10^4	10^5	10^6
$O(n^3)$	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}

2 Growth Rate Examples - Exponential Growth Rate - $O(2^n)$

- As the size of a problem increases, the time requirement usually increases too rapidly to be practical.

```
def rate7(n):
    s = "ZONE OUT IN CLASS"
    for i in range(2 ** n):
        print("I must not ", s)
```



n	10^1	10^2	10^3	10^4	10^5	10^6
$O(2^n)$	10^3	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}

Exercise

- What is the Big-O of the following statements?

```
for i in range(n):              executed n times
    for j in range(10):         executed 10 times
        print(i, j)             constant time
```

  - Running time, $T(n) = n * 10 * 1 = 10n$, Big-O =

- What is the Big-O of the following statements? Big-O =

```
for i in range(n):              executed n times
    for j in range(n):          executed n times
        print(i, j)
for k in range(n):              executed n times
    print(k)
```

- The first set of nested loops is $O(n^2)$ and the second loop is $O(n)$. This is $O(max(n^2,n))$ Big-O =

Exercise

- What is the Big-O of the following statements?

```
for i in range(n):              executed n times
    for j in range(10):         executed 10 times
        print(i, j)             constant time
```

  - Running time, $T(n) = n * 10 * 1 = 10n$, Big-O = $O(n)$

- What is the Big-O of the following statements? Big-O =

```
for i in range(n):              executed n times
    for j in range(n):          executed n times
        print(i, j)
for k in range(n):              executed n times
    print(k)
```

- The first set of nested loops is $O(n^2)$ and the second loop is $O(n)$. This is $O(\max(n^2,n))$ Big-O = $O(n^2)$

Quiz

- What is the Big-O of the following statements?

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

  - When i is 0, the inner loop executes (n - 1) times.
    When i is 1, the inner loop executes (n - 2) times.
    …
    When i is (n - 2), the inner loop executes once.
  - The number of times the inner loop statements execute:
    - Running time, T(n) =
    - Big-O =

3 Profiling: Measuring Growth Rate

Problem: Predict the running time of a big data set (i.e., n = 1 million or 1 billion).

- Most algorithms approximately have the order of growth of the running time:

$$T(N) \approx aN^b$$

- For example, we may compute the constant "a" and the growth rate "b" from data we got from profiling (i.e., performance analysis) as shown below.

N	sec
1000	0.000949
2000	0.001706
3000	0.002773
4000	0.004145
5000	0.004781
6000	0.005814
7000	0.006897
8000	0.008350
9000	0.009346
10000	0.009941

3 Profiling: Measuring Growth Rate

Problem: Predict the running time of a big data set (i.e., n = 1 million or 1 billion).

- Most algorithms approximately have the order of growth of the running time:

$$T(N) \approx aN^b$$

- For example, we may compute the constant "a" and the growth rate "b" from data we got from profiling (i.e., performance analysis) as shown below.

- **Since $T(N) \approx a N^b$, $T(2N) = a\,(2N)^b$, then**

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = \frac{2^b(N)^b}{N^b} = 2^b$$

Take log both sides

$$log\frac{T(2N)}{T(N)} = \log 2^b$$

$$b = log\frac{T(2N)}{T(N)}$$

3 Profiling: Measuring Growth Rate - Example

- **Example:** let us choose N = 4000 or 2N = 8000, an average case of the insertion sort shown above. Recall that log we use here is **log base 2.**

$$b = log\frac{T(2N)}{T(N)} = log\frac{t2(8000)}{t1(4000)} = log\frac{0.036643}{0.011144} = 1.717$$

- Now, we use this $\mathrm{b} = 1.717$ to solve for a when $\mathrm{N} = 4000,\ \mathrm{T(N)} = 0.011144$ in the following:

$$\mathrm{T(N)} = a\,N^{1.717}$$
$$0.011144 = a\,(4000)^{1.717}$$
$$a = \frac{0.011144}{(4000)^{1.717}}$$
$$a = 7.27\mathrm{x}10^{-9}$$

- Therefore, we have the growth rate $\mathrm{b} = 1.717$, **the constant** $a = 7.27\mathrm{x}10^{-9}$ for the insertion sort average case.

Summary

- Performance analysis or profiling measures an algorithm's time requirement as a function of the problem size **n** by **using a growth-rate** function.

- The **growth rates** shown below are commonly used :

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

- Generally, **growth rates** can be measured in a form of the following:

$$T(N) \approx aN^b$$