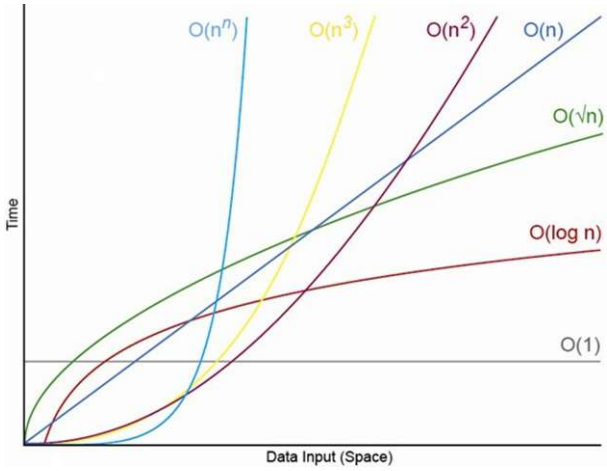
22. Big-O 표기법 (Big-O notation)

- 시간복잡도를 측정하는 방법
- 동일한 문제를 여러가지 알고리즘으로 구현할 수 있으나 각 알고리즘이 얼마나 빠른지 비교할 수 있는 기준으로 Big-O 표기법 사용.



- Big-O 표기법은 algorithm 에 어떤 input 이 들어가서 결과가 나올 때까지 몇단계의 실행문을 거치는지 표시해 준다. 예를 들어, n 번의 실행문을 거치면 O(n), n^2 번이면 O(n^2) 으로 표시한다.
- Jupyter notebook 의 경우 %timeit magic command 로 두가지 알고리즘의 수행시간을 비교할 수 있으나, CPU 속도에 dependent 하므로 알고리즘의 복잡성을 측정하는 기준으로 사용하기 어렵다.
- 일반적인 Big-O 기준

```
insertion / removel / access / push / pop - O(1)
Linear Searching - O(n)
List Push/Pop - O(1)
List slice / concatenate - O(n)
Bubble Sort : O(n^2)
Sort - O(nlogn) : merge sort, quick sort
for / map / filter / reduce - O(n)
```

In [1]:

```
def factorial_1(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print (factorial_1(5))
```

```
120
```

In [2]:

```
def factorial_2(n):
    if n == 0:
        return 1
    else:
        return n * factorial_2(n-1)

print (factorial_2(5))
```

120

In [3]:

```
%timeit factorial_1(50)
```

4.91 µs ± 617 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [4]:

```
%timeit factorial_2(50)
```

7.64 µs ± 726 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

O(1) : input 크기에 무관하게 항상 한번의 연산만 수행

In [5]:

```
def constant_algo(items):
    result = items[0] * items[0]
    print (result)

constant_algo([4, 5, 6, 8])
```

16

O(n) : 선형복잡도 (Linear Complexity)

입력자료의 갯수에 따라 수행 시간이 선형으로 증가 한다.

In [6]:

```
lst = []
def linear_algo(items):
    for item in items:
        lst.append(item)
```

In [7]:

```
%timeit linear_algo(list(range(100)))
```

10.8 µs ± 817 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [8]:

```
%timeit linear_algo(list(range(1000)))
```

115 µs ± 20 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

O(n^2) : 제곱증가 복잡도 (Quadratic Complexity)

In [9]:

```
def quadratic_algo(items):
    n2items = []
    for item in items:
        for item2 in items:
            n2items.append(item2)
```

In [10]:

```
%timeit quadratic_algo(list(range(100)))
```

535 µs ± 58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [11]:

```
%timeit quadratic_algo(list(range(200)))
```

2.02 ms ± 210 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

O(log n)

big-O notation 에서 사용하는 log 는 일반적으로 밑이 2 인 log 이다. 즉,

$$\begin{aligned} 2^0 &= 1 & log_2 1 &= 0 \\ 2^1 &= 2 & log_2 2 &= 1 \\ 2^2 &= 4 & log_2 4 &= 2 \\ 2^3 &= 8 & log_2 8 &= 3 \\ 2^{10} &= 1024 & log_2 1024 &= 10 \end{aligned}$$

log 안의 숫자가 2 배로 늘어도 log 값은 선형으로 증가한다.
대표적인 O(log N) 알고리즘은 이진탐색(binary search) 이다. 이진탐색은 target value 를 찾을 때까지 input data 를 절반씩 잘라내어 탐색하므로 data 의 log 값 만큼만 계산 시간이 증가한다.

In [15]:

```
n = 100
lst = list(range(1, n+1))
print(lst)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

Brute-force algorithm : 모든 가능한 경우를 순차적으로 시도

O(n)

In [18]:

```
def findNumber_Brute(n, lst):
    search_count = 1

    for x in lst:
        if x == n:
            return search_count
        else:
            search_count += 1

findNumber_Brute(100, lst)
```

Out[18]:

100

이진 탐색 (Binary Search)

O(log n)

In [19]:

```
def findnumber_Binary(n, lst):
    start = 0
    end  = len(lst)
    numbers = lst
    search_count = 0

    while (start < end):
        search_count += 1
        middle = len(numbers) // 2                  # 가운데의 위치

        if numbers[middle] == n:
            return search_count
        elif numbers[middle] > n:                   # 찾으려는 단어가 아래쪽 절반에 위치
            start, end = 0, middle
        else:
            start, end = middle+1, len(numbers)    # 찾으려는 단어가 위쪽 절반에 위치

        numbers =  numbers[start : end]
```

In [20]:

```
print(findnumber_Binary(50, lst))
```

6

In [21]:

```
n = 100000
lst2 = list(range(1, n+1))
print(findnumber_Binary(50, lst2))
```

17

O(log n)

In [22]:

```
import math

print(math.log(100, 2))
print(math.log(100000, 2))
```

6.643856189774725
16.609640474436812

Big-O 연습문제

1) 다음 code 의 Big-O 수행시간은 ?

```
test = 0
for i in range(n):
    for j in range(n):
        test = test + i * j
```

2) 다음 code 의 Big-O 수행시간은 ?

```
test = 0
for i in range(n):
    test = test + 1
for j in range(n)
    test = test - 1
```

3) 다음 code 의 Big-O 수행시간은 ?

```
i = n
while i > 0:
    k = 2  + 2
    i = i // 2
```