1. Complexity Predictions & Analysis

2017010698
수학과 오서영

Time Complexity (시간 복잡도) -> Big O notation

빠른 알고리즘을 만들려면? -> 알고리즘의 속도를 어떻게 측정할지 정하기

두 알고리즘의 속도를 비교하는 가장 직관적인 방법
-> 프로그램 수행 시간 측정
But 부적합한 기준 -> 사용한 언어, 하드웨어, 운영체제, 컴파일러 등의 요소에 의해 바뀔 수 있기 때문

알고리즘의 수행 시간을 지배(dominate) 하는것은? -> 반복문

Ex) N번 수행되는 반복문이 두 개 겹쳐져 있으면,
알고리즘의 수행시간은 N^2

시간 복잡도

- 가장 널리 사용되는 알고리즘 수행시간 기준
- 알고리즘이 실행되는 동안 수행하는 **기본적인 연산**의 수를 입력 크기에 대한 함수로 표현한 것
- **시간 복잡도가 높다** = 입력크기가 증가할 때 알고리즘 수행 시간이 더 빠르게 증가한다

기본적인 연산

1. 두 32비트 정수의 사칙연산
2. 두 실수형 변수의 대소 비교
3. 변수 대입하기

입력의 종류에 따른 수행시간의 변화

- 반복문이 실행되는 횟수는 찾는 원소의 위치에 따라 달라짐
-> 최선/최악의 경우, 평균적인 경우에 대한 수행시간을 각각 따로 계산

```
For i in range(len(array)) :
    If array[i] == element :
        print(i)
```

최선의 수행시간 : 1

최악의 수행시간 : N

평균적인 수행시간 : N/2
(모든 입력의 등장 확률이
모두 같다고 가정)

Big O notation

- 주어진 함수에서 가장 빨리 증가하는 항만을 남긴 채
나머지를 다 버리는 표기법
- 함수의 상한을 나타낸다는 의미

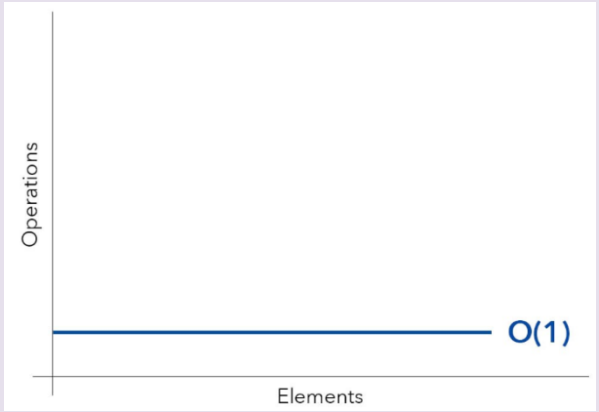
EX) 1. f(N) = 3N^2 + 16N – 7 -> **O(N^2)**
2. f(N) = N^M + NlogM + NM^ -> **O(N^M + NM^)**
3. f(N) = 42 -> **O(1)**

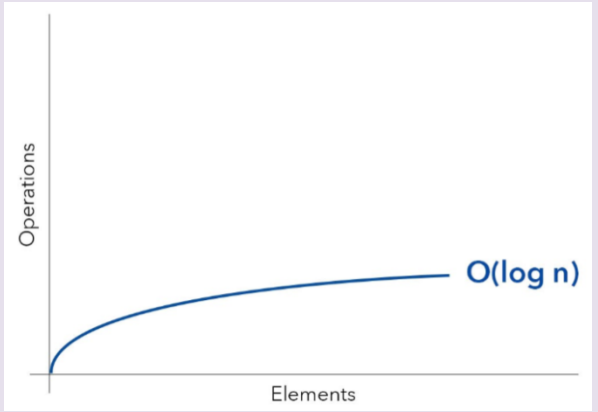
```
a=5 ; b=6 ; c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

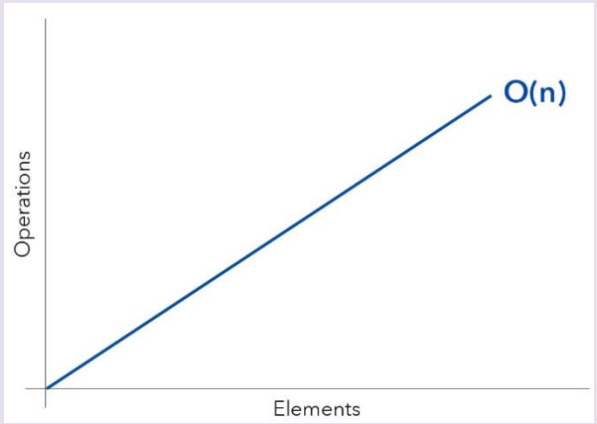
T(n) = 3n^2+2n+4
-> O(n^2)

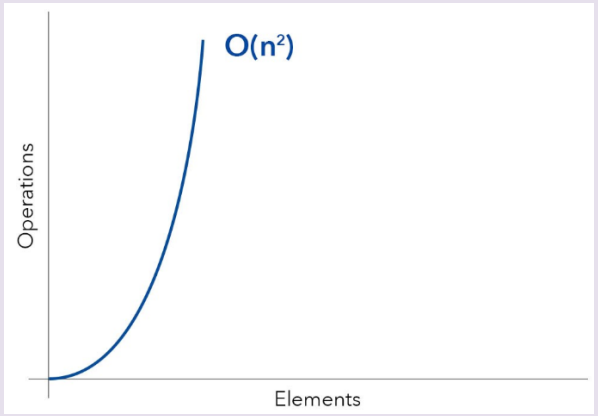
Common varieties of Big O Notation

1. O(1) - Constant time complexity
2. O(log n) - Logarithmic time complexity
3. O(n) - Linear time complexity
4. O(n^2) - Quadratic time complexity









Big O Notation examples

O(1)

```
void printFirstItem(const vector<int>& items)
{   cout << items[0] << endl;   }
```

the input array could be 1 item or 1,000 items, but this function would still just require one "step."

O(n)

```
void printAllItems(const vector<int>& items)
{   for (int item : items) {
      cout << item << endl;    }}
```

O(n^2)

```
void printAllPossibleOrderedPairs(const vector<int>& items)
{   for (int firstItem : items) {
      for (int secondItem : items) {
    cout << firstItem << ", " << secondItem << endl;
}}}
```

two loops. If the vector has 10 items, we have to print 100 times.

Space Complexity (공간 복잡도)

-프로그램을 실행시킨 후 완료하는 데 필요로 하는 자원 공간의 양

-총 공간 요구 = 고정 공간 요구 + 가변 공간 요구

- **고정 공간** : 입력과 출력의 횟수나 크기와 관계없는 공간의 요구
(코드 저장 공간, 단순 변수, 고정 크기의 구조 변수, 상수)
- **가변 공간** : 해결하려는 문제의 특정 인스턴스에 의존하는 크기를 가진
-구조화 변수들을 위해서 필요로 하는 공간, 함수가 순환 호출을 할 경우 요구되는 추가 공간 -> 동적으로 필요한 공간

```
int factorial(int n) { if(n > 1)
return n * factorial(n - 1);
else return 1; }
```

n이 1 이하일 때까지 함수가 재귀적으로 호출되므로 스택에는 n부터 1까지 모두 쌓이게 됨.
-> O(n)

Time Complexity vs Space Complexity

시간 복잡도 : "얼마나 빠르게 실행되느냐"
공간 복잡도 : "얼마나 많은 자원이 필요한가?"

시간과 공간은 반비례적인 경향
->알고리즘의 척도는 시간 복잡도를 위주로 판단합니다.
-> 시간 복잡도만 괜찮다면 공간 복잡도는 어느 정도 이해

Reference

[1] 구종만,『 프로그래밍 대회에서 배우는 알고리즘 문제 해결 전략 』, 인사이트(2012)