시간복잡도의 이해

> 메가스터디IT아카데미 이광호 강사

#01. 시간복잡도

알고리즘이 주어진 문제를 해결하기 위해 수행하는 연산 횟수.

일반적으로 파이썬은 초당 2000만 ~ 1억번의 연산을 수행함.

1. 시간 복잡도 유형

유형	표기	설명
빅-오메가	$\Omega(n)$	최선일 때(best case) 연산 횟수를 나타낸 표기법
빅-세타	$\Theta(n)$	보통일 때(average case) 연산 횟수를 나타낸 표기법
빅-오	$O(n)$	최악일(worst case) 연산 횟수를 나타낸 표기법

2. 시간복잡도 예제 코드

Ex01

```
import random

def find(x):
    for i in range(1, 101):
        print(i, end=" ")
        if i == x:
            return i

findNumber = random.randrange(1, 100)
result = find(findNumber)
print(f'\n숫자를 찾았습니다. 찾은 숫자는 {result}입니다.')
```

```
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
숫자를 찾았습니다. 찾은 숫자는 73입니다.
```

예시 설명

> N은 데이터의 크기

유형	설명	시간복잡도
최선일 때	findNumber가 1이 되어 단 한번만에 반복문이 종료됨	1
보통일 때	총 100회의 반복 중에서 평균치인 50회 전후로 반복문이 종료됨	$N/2$
최악일 때	findNumber가 100이 되어 반복문의 마지막 회차에서 값을 찾음	N

3. 알고리즘에서 사용되는 시간 복잡도

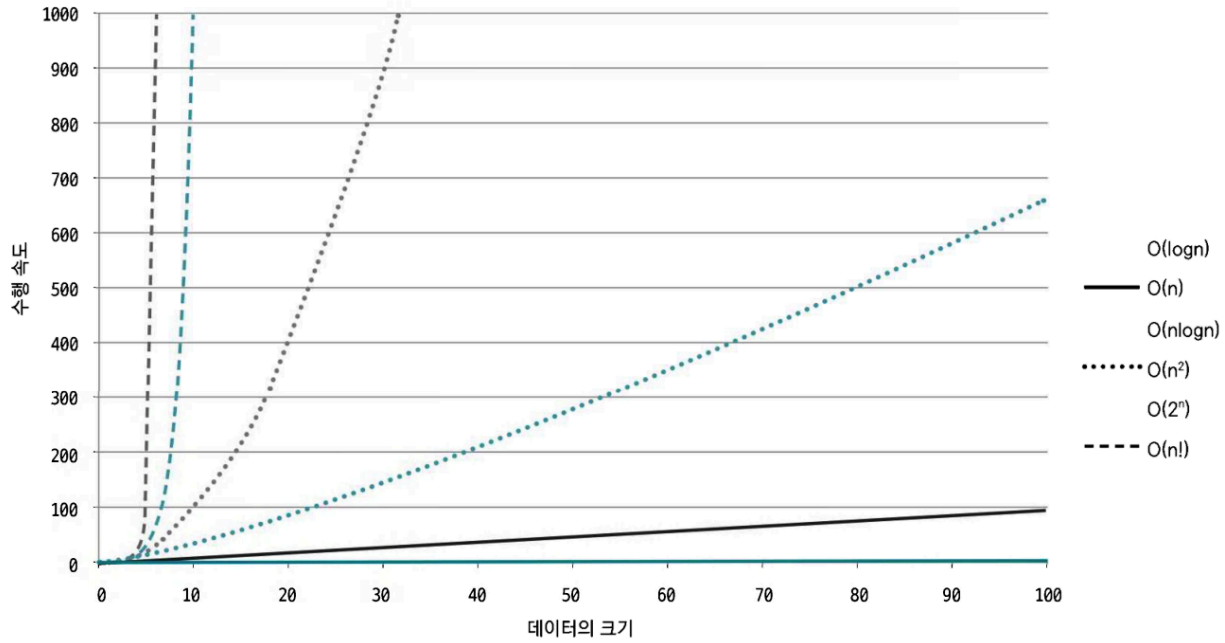
시간복잡도	설명
$O(1)$	문제를 해결하는데 오직 한 단계만 처리함
$O(\log n)$	문제를 해결하는데 필요한 단계들이 연산마다 특정 요인에 의해 줄어듬
$O(n)$	문제를 해결하기 위한 단계의 수와 입력값 n이 `1:1` 관계를 가짐
$O(n \log n)$	선형로그형 - 문제를 해결하기 위한 단계의 수가 $N \times \log_2 n$번 만큼의 수행시간을 갖는다
$O(n^2)$	2차 시간 - 문제를 해결하기 위한 단계의 수는 입력값 n의 제곱

벅-오 표기법을 기준으로 수행 시간을 계산하는 것이 좋다.

실제 코딩 테스트에서는 1개의 테스트 케이스로 합격 , 불합격을 결정하지 않는다.

응시자가 작성한 프로그램으로 다양한 테스트 케이스를 수행해 모든 케이스를 통과해야만 합격으로 판단 하므로 시간 복잡도를 판단할 때는 최악일 때 염두에 둬야 한다.

빅오 표기법으로 표현한 시간 복잡도 그래프



빅-오 표기법(O(n))의 시간 복잡도

> 각각의 시간 복잡도는 데이터 크기(N)의 증가에 따라 성능(수행 시간)이 다르다는 것을 알 수 있다.

#02. 시간 복잡도 도출하기

1. 시간 복잡도 도출 기준

1. 상수는 시간 복잡도 계산에서 제외한다.
2. 가장 큰 차수의 항만 남긴다
    - 가장 많이 중첩된 반복문의 수행 횟수가 시간 복잡도의 기준이 된다.

2. 연산 횟수가 N인 경우

Ex02-1

```
def foo(N, cnt):
    for i in range(N):
        print("연산횟수: ", cnt)
        cnt += 1

foo(100000, 1)
```

주어진 데이터를 기준으로 n번 수행하므로 시간 복잡도는 $O(n)$

3. 연산 횟수가 $3N$ 인 경우

Ex02-2

```
def foo(N, cnt):
    for i in range(N):
        print("연산횟수: ", cnt)
        cnt += 1

    for i in range(N):
        print("연산횟수: ", cnt)
        cnt += 1

    for i in range(N):
        print("연산횟수: ", cnt)
        cnt += 1

foo(100000, 1)
```

주어진 데이터를 기준으로 n번씩 3셋트를 수행하기 때문에 $O(3n)$이지만 시간복잡도에서는 상수를 모두 무시하므로 시간 복잡도는 $O(n)$

4. 상수항을 제거하는 또 다른 경우

Ex02-3

```
def foo(N, cnt):
    print("시작~!!!")                # 1회

    for i in range(N/2):
        print("연산횟수: ", cnt)     # N/2회
        cnt += 1

    for i in range(100):
        print("연산횟수: ", cnt)     # 100회
        cnt += 1

foo(100000, 1)
```

위의 코드에서 시간 복잡도는 $O(1+\frac{n}{2}+100)$이지만 이 역시 상수를 제거하여 $O(n)$이 된다.

5. 연산 횟수가 N^2 인 경우

Ex02-4

```
def foo(N, cnt):
    for i in range(N):
        for j in range(N)
            print("연산횟수: ", cnt)
            cnt += 1

foo(100000, 1)
```

6. 가장 큰 차수의 항만 남기기

Ex02-5

```
def foo(N, cnt):
    for i in range(N):              # N회
        print("연산횟수: ", cnt)
        cnt += 1

    for i in range(N):              # N^2회
        for j in range(N)
            print("연산횟수: ", cnt)
            cnt += 1

foo(100000, 1)
```

위 경우에서 시간복잡도는 $O(n+n^2)$로 계산된다.

N이 커질 수록 더 영향력이 큰 항, 즉 가장 큰 차수의 항만 남겨두고 삭제해 준다.

따라서 위의 코드에 대한 시간 복잡도는 $O(n^2)$이다.

#03. 시간 복잡도 활용 - 수 정렬하기

N개의 수가 주어졌을 때 이를 오름차순으로 정렬하는 프로그램을 작성하시오.

> 백준 온라인 저지 2750번을 각색함

◎ 입력

입력값은 N개의 원소를 갖는 리스트이다. 리스트의 각 원소는 `N(1 ⩽ N ← 1,000,000)`으로 구성된다.

이 수는 절댓값이 `1000,000`보다 작거나 같은 정수다. 리스트의 원소는 중복되지 않는다.

◎ 출력

정렬이 완료된 리스트를 출력한다.

입력	출력
[5, 2, 3, 4, 1]	[1, 2, 3, 4, 5]

◎ 연산 횟수 계산 방법

$$\text{연산횟수} = \text{알고리즘 시간복잡도} \, n\text{값에 데이터의 최대 크기를 대입하여 도출}$$

◎ 버블정렬 풀이

버블 정렬은 인접한 두 요소를 비교하여 크기 순서에 맞지 않으면 교환하는 방식으로 리스트를 정렬하는 알고리즘

Ex03

```
def solution(lst):
    # 리스트의 길이를 구합니다.
    n = len(lst)

    # 리스트의 모든 요소를 비교하는 반복문
    for i in range(n):
        # 내부 루프는 리스트의 끝까지 비교하는데,
        # 이미 정렬된 마지막 i개 요소는 비교하지 않아도 된다.
        for j in range(0, n-i-1):
            # 인접한 두 요소를 비교해서 앞의 요소가 더 크면 서로 교환
            if lst[j] > lst[j+1]:
                # 요소를 교환합니다.
                lst[j], lst[j+1] = lst[j+1], lst[j]

    # 정렬된 리스트를 반환합니다.
    return lst

# 예시 사용
lst = [5, 2, 3, 4, 1]
sorted_lst = solution(lst)
print(sorted_lst)
```

```
[1, 2, 3, 4, 5]
```

◎ 시간복잡도 계산 코드 추가

시간 복잡도를 계산하는 방법은 코드 내에서 특정 연산이 몇 번 수행되는지를 확인하는 것.

버블 정렬의 경우, 두 요소를 비교하는 연산과 교환 연산이 시간 복잡도에 영향을 미친다.

비교 연산이 얼마나 자주 발생하는지를 추적하는 방식으로 시간 복잡도를 분석할 수 있다.

```
def solution(lst):
    # 리스트의 길이를 구합니다.
    n = len(lst)

    # 비교 연산의 횟수를 저장할 변수
    comparison_count = 0

    # 리스트의 모든 요소를 비교하는 반복문
    for i in range(n):
        # 내부 루프는 리스트의 끝까지 비교하는데,
        # 이미 정렬된 마지막 i개 요소는 비교하지 않아도 된다.
        for j in range(0, n-i-1):
            # 비교 연산 횟수를 증가시킴
            comparison_count += 1
            # 인접한 두 요소를 비교해서 앞의 요소가 더 크면 서로 교환
            if lst[j] > lst[j+1]:
                # 요소를 교환합니다.
                lst[j], lst[j+1] = lst[j+1], lst[j]
```

```
    # 정렬된 리스트와 비교 연산 횟수를 함께 반환
    return lst, comparison_count

# 예시 사용
lst = [64, 34, 25, 12, 22, 11, 90]
sorted_lst, comparison_count = solution(lst)
print(f"정렬된 리스트: {sorted_lst}")
print(f"비교 연산 횟수: {comparison_count}")
```

```
정렬된 리스트: [11, 12, 22, 25, 34, 64, 90]
비교 연산 횟수: 21
```

시간 복잡도는 알고리즘이 실행되는 동안 수행되는 기본 연산(예: 비교, 교환, 접근 등)의 횟수를 기반으로 계산된다.

데이터의 크기(리스트의 길이 n)가 커질수록 얼마나 더 많은 연산이 필요해지는지를 분석하는 것이 목표이다.

버블 정렬의 경우, 두 요소를 비교하고 필요하면 자리를 바꾸는 연산을 반복하게 되는데, 이 과정에서 시간 복잡도를 계산할 수 있다.

◎ 버블 정렬의 시간 복잡도를 분석하는 과정

버블 정렬은 인접한 두 요소를 비교하고, 만약 정렬 순서가 맞지 않으면 교환하는 작업을 반복해서 리스트를 정렬한다.

이 과정에서 여러 번의 비교와 교환이 일어나게 된다.

1. 외부 반복문 (i 루프)

```
for i in range(n):
```

이 루프는 리스트의 길이 n만큼 반복.

n은 리스트에 있는 요소의 개수이다.

즉, 외부 반복문은 총 n번 반복됩니다.

2. 내부 반복문 (j 루프)

```
for j in range(0, n-i-1):
```

내부 반복문은 인접한 두 요소를 비교

처음에는 $n-1$번 비교하고, 그 다음에는 $n-2$번 비교하는 방식으로 점점 비교하는 범위가 줄어든다.

내부 반복문에서 비교할 때마다 비교 횟수가 $n-i-1$번씩 줄어듭니다. 여기서 i는 외부 반복문의 반복 횟수.

3. 비교 연산의 총 횟수

각 반복에서 비교 연산이 몇 번 일어나는지 계산할 수 있다.

예를 들어, 리스트에 $n=5$개의 요소가 있다고 가정하면:

- 첫 번째 반복 ($i=0$): 4번 비교 ($n-1$번)
- 두 번째 반복 ($i=1$): 3번 비교 ($n-2$번)
- 세 번째 반복 ($i=2$): 2번 비교 ($n-3$번)
- 네 번째 반복 ($i=3$): 1번 비교 ($n-4$번)
- 다섯 번째 반복 ($i=4$): 0번 비교 ($n-5$번)

즉, n개의 요소가 있을 때 비교 연산의 총 횟수는 다음과 같다.

$$(n-1)+(n-2)+(n-3)+\cdots+1=\frac{n(n-1)}{2}$$

4. 시간 복잡도 계산

이제 시간 복잡도를 계산하기 위해 리스트의 길이를 기준으로 위 수식을 생각해본다면,

$\frac{n(n-1)}{2}$ 는 큰 수일 때 근사적으로 n^2 에 비례한다. 수학적으로는 $\frac{n^2}{2}$ 와 유사하므로, 상수 계수(2)는 무시할 수 있다.

따라서 버블 정렬의 시간 복잡도는 최악의 경우와 평균적으로 $O(n^2)$으로 계산된다.

◎ 실제 풀이 접근

실제 문제에서는 시간 제한이 2초이므로 이 조건을 만족하려면 `4,000`만 번 이하의 연산 횟수로 문제를 해결해야 한다.

따라서 문제에서 주어진 시간 제한과 데이터 크기를 바탕으로 어떤 정렬 알고리즘을 사용해야 할 것인지를 판단할 수 있다.

- 연산 횟수는 1초에 2,000만 번 연산하는 것을 기준으로 생각한다.
- 시간 복잡도는 항상 최악일 때 , 즉 데이터의 크기가 가장 클 때를 기준으로 한다.

$$\text{버블정렬}=(1000000)^2=1000000000000>40000000$$

$$\text{병합정렬}=1000000\log_2(1000000)=\text{약}20000000<40000000$$

그러므로 실제 코딩 테스트에서는 병합정렬을 사용해야 한다.