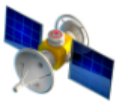

복잡도(시간, 공간)

주차	21주차
스터디 일자	@2024/03/22

👾 복잡도

> 알고리즘 성능을 평가하기 위해 '복잡도(Complexity)'의 척도를 사용
> 동일한 기능을 수행하는 알고리즘이 있을 때
> `복잡도가 낮을 수록` 좋은 알고리즘이라 말한다.
>
> - **시간 복잡도** : 특정한 크기의 입력에 대하여 알고리즘의 **수행 시간 분석**
> - **공간 복잡도** : 특정한 크기의 입력에 대하여 알고리즘의 **메모리 사용량 분석**

🌌 공간 복잡도

공간 복잡도는 **작성한 프로그램이 얼마나 많은 공간(메모리)을 차지하느냐를 분석하는 방법**이다.

but! 예전에 비해 컴퓨터 성능의 발달로 인해 메모리 공간이 넘쳐나다 보니 **중요도는 떨어졌다고 한다!**

반면, 시간 복잡도의 경우 알고리즘을 잘못 구성하면 결괏값이 나오지 않거나 너무 느린 속도로 나와서 최근에는 시간 복잡도를 더 우선시하여 프로그래밍을 작성한다.

→ 시간과 공간은 **반비례적 경향이 있음**

→ 최근 대용량 시스템이 보편화되면서, 공간 복잡도보다는 시간 복잡도가 우선

→ 알고리즘은 **시간 복잡도**가 중심

⏳ 시간 복잡도

시간 복잡도는 **특정 알고리즘이 어떤 문제를 해결하는데 걸리는 시간**을 의미한다.

같은 결과를 갖는 프로그래밍 소스도 작성 방법에 따라 걸리는 시간이 달라지며, 같은 결과를 갖는 소스라면 시간이 적게 걸리는 것이 좋은 소스이다.

시간 복잡도를 표기하는 방법 3가지

Big-O(빅-오) ⇒ 상한 점근 (최악)

Big-Ω(빅-오메가) ⇒ 하한 점근 (최선)

Big-θ(빅-세타) ⇒ 그 둘의 평균 (중간)

위 세 가지 표기법은 시간 복잡도를 각각 **최악**, **최선**, **중간(평균)**의 경우에 대하여 나타내는 방법이다.

가장 자주 사용되는 표기법은?

[빅-오 표기법!]

빅오 표기법은 최악의 경우를 고려하므로, 프로그램이 실행되는 과정에서 소요되는 **최악의 시간까지 고려할 수 있기 때문이다.**

"최소한 특정 시간 이상이 걸린다" 혹은 "이 정도 시간이 걸린다"를 고려하는 것보다,

"이 정도 시간까지 걸릴 수 있다"를 고려해야 그에 맞는 대응이 가능하다.

✨ 빅-오 표기법

시간 복잡도 계산에 사용하는 개념

> 예를 들어, 동전을 튕겨 뒷면이 나올 확률을 이야기 할 때 운이 좋으면 1번에 뒷면이 나오지만 운이 안 좋다면 n번 만큼 동전을 튕겨야 하는 경우가 발생한다.

이 **최악의 경우를 계산하는 방식을 빅-오(Big-O) 표기법**이라 부른다.

> **시간 복잡도 계산하는 방법**
>
> - 코드를 보고 계산
> - 코드 작성전 문제의 크기를 보고 구현 방법을 생각하여 계산
>
> 사실 일일히 연산을 세는것은 큰 의미가 없다.
> 반복문 밖에 있는 단순 연산들은 결국
> **상수항이라 무시되고**,
> 반복문 안에 있는 연산이라도 가장 큰 차수 외에는
> **각 항의 계수를 포함한 모든 것들이 무시**되기 때문이다.
> 결국
> **반복문과 재귀로 반복되는 횟수**만 확인하면 된다.
>
> 예를 들어, 반복횟수가 `n`인 `for` 반복문이 이중으로 있다면 `n`에 대한 1차식 두개를 곱하므로 연산의 개수에 대한 식은 n^2이 되는 것이고 시간복잡도는 $O(n^2)$이다.
>
> 결국 빅오 표기법은 알고리즘 내에서 반복의 차수와 직결된다고 볼 수 있다.

👉 `O(1)`

입력 데이터의 크기에 상관없이 **언제나 일정한 시간이 걸리는 알고리즘.**
데이터가 얼마나 증가하든 성능에 영향을 거의 미치지 않음.

`O(log n)`

입력 데이터의 크기가 커질수록 처리 **시간이 로그(log: 지수 함수의 역함수) 만큼 짧아지는 알고리즘.**
예를 들어 데이터가 10배가 되면, 처리 시간은 2배가 된다.
예시) 이진 탐색, 재귀가 순기능으로 이루어지는 경우

`O(n)`

입력 데이터의 **크기에 비례해 처리 시간이 증가하는 알고리즘.**
예를 들어 데이터가 10배가 되면, 처리 시간도 10배가 된다.
예시) 1차원 for문

`O(n log n)`

데이터가 많아질수록 처리시간이 **로그(log) 배만큼 더 늘어나는 알고리즘.**
예를 들어 데이터가 10배가 되면, 처리 시간은 약 20배가 된다.
예시) 병합 정렬, 퀵 정렬

`O(n²)`

데이터가 많아질수록 처리시간이 급수적으로 늘어나는 알고리즘.
예를 들어 데이터가 10배가 되면, 처리 시간은 최대 100배가 된다.
예시) 이중 루프

`O(2ⁿ)`

데이터량이 많아질수록 처리시간이 **기하급수적으로 늘어나는 알고리즘.**
예시) 피보나치 수열, 재귀가 역기능을 할 경우

💡 **faster** `O(1)` < `O(log n)` < `O(nlog n)` < `O(n²)` < `O(2ⁿ)` **slower**

빅-오 표기법 구하는 법?

1) 연산의 개수를 세어본다.

2) 가장 높은 차수만 남긴다.

- ex) $O(n^2+n) \Rightarrow O(n^2)$

3) 계수 및 상수는 과감하게 버린다.

- ex) $O(2n+3) \Rightarrow O(n)$

🌈 예시

```
int sum = 0;
for(int i = 0; i < n; i++){
        sum += i;
}
```

▼ 뭘까요~?

`sum = 0` 한 번, `int i = 0` 한 번, `i++` n번, `sum+=i` n번
합쳐서 총 2n+2번의 연산이 수행된다.

답: O(N)

```
int sum = (x + 1) * x / 2;
```

▼ 뭘까요~?

`(x+1)` 한 번, `*x`가 한 번, `/2`가 한 번, 계산된 값을 sum에 대입할 때 한 번
이렇게 총 네 번이므로 4가 되지만, Big-O 표기법으로는 O(1)이 된다.

답: O(1)

```
int sum = 0;
for(int i = 0; i < n; i++){
        for(int j = 0; j < i;j++){
            sum += j;
      }
}
```

▼ 뭘까요~?

바깥쪽 반복문은 n번, 안쪽 반복문은 i의 값에 따라 반복한다.
i는 0부터 n-1까지 변하고, 안쪽 반복문은 해당하는 i만큼 반복하므로
0 + 1 + 2 +...+ (n-1) ⇒
`n * (n - 1) / 2` 번 반복한다.

답: $O(n^2)$

```
int sum = 0;
for(int i = n; i > 0; i/=2){
        for(int j = 0; j < i; j++){
            sum += j;
    }
}
```

▼ 뭘까요~?

바깥쪽 반복문이 log N번 반복, 안쪽 반복문은 i값에 따라 반복 횟수가 달라진다.
→ logN번 반복인 이유는 i/=2이기 때문

i는 n부터 1까지 변하고, 안쪽 반복문은 해당하는 i만큼 반복하므로
n + (n / 2) + (n / 4) +...+ 1 =
`2n` 번 반복한다.

답: O(N)

정렬 알고리즘 복잡도 비교

정렬 종류	시간 복잡도			공간 복잡도
	평균(Average)	최선(Best)	최악(Worst)	
선택 정렬	O(n2)	O(n2)	O(n2)	O(n2)
버블 정렬	O(n2)	O(n2)	O(n2)	O(n)
삽입 정렬	O(n2)	O(n)	O(n2)	O(n2)
합병 정렬	O(n×log n)	O(n×log n)	O(n×log n)	O(n×log n)
퀵 정렬	O(n×log n)	O(n×log n)	O(n2)	O(n×log n)
힙 정렬	O(n×log n)	O(n×log n)	O(n×log n)	O(n×log n)
쉘 정렬	O(N^1.25)	O(N^1.25)	O(N^1.25)	O(n)
기수 정렬	O(dn)	O(dn)	O(dn)	