📌

알고리즘 복잡도 분석

일반적으로 알고리즘의 복잡도를 분석하고자 할 때, 시간 복잡도를 많이 보기 때문에, 이에 대해서 살펴보자!

그에 앞서서, 복잡도에는 효율성 면에서 분류될 수 있는 복잡도에 대해서 알아보자

I. 효율성 면에서 복잡도의 분류

1. **최악의 복잡도 Worst Case Complexity**
2. **평균적인 복잡도 Average Case Complexity**
3. **최선의 복잡도 Best Case Complexity**

각각에 대해서 조금 더 이해해보자!

🌟최악의 복잡도, Worst Case Complexity🌟

- 입력크기 n에 대한 문제를 풀 때, **최악의 성능**을 내는 복잡도
- 알고리즘 실행 시간의 상한을 계산(➡️ Big-O, Theta 표기법이 관련있을 것으로 생각됨!)
- **가장 많이 사용하는 분석방법!**

평균적인 복잡도, Average Case Complexity

- 평균적인 성능을 내는 복잡도

- 가능한 모든 입력에 대한 실행 시간을 계산하여 그 평균을 계산

최선의 복잡도, Best Case Complexity

- 최선의 성능을 내는 복잡도

이번에는 시간 복잡도에 대해서 살펴보도록 하자

🌟II. 시간 복잡도

- 시간 복잡도의 종류(아래로 내려갈 수록, 시간이 증가)

시간 복잡도의 종류

Aa 이름	표기법
상수(constant) 시간	O(1)
로그(Logarithmic) 시간	O(logn)
선형(linear) 시간	O(n)
N LogN(N 로그 N)시간	O(nlogn)
이차(quadratic) 시간	O(n^2)
다항(polynomial) 시간	O(n^c)(c > 1, c=const)
지수(exponential) 시간	O(c^m) (c > 1, c= const)
계승(factorial) 또는 n의 n승(N-power-N) 시간	O(n!) 또는 O(n^n)

🌟정리🌟

상수 < 로그 < **선형 < nlogn** < **이차 < 다항** < **지수 < 계승 또는 n의 n승**

색상을 다르게 하여 표시한 이유는, 색상이 바뀌는 시점에서 하나의 chunk 처럼 구획되어 복잡해지는 것으로 볼 수 있기 때문이다!

처음에는 n의 0 승 등 복잡도가 없었지만, 선형을 기준으로 하나의 chunk가 nlogn과 함께 형성될 수 있는 등 특징으로 묶여질 수 있기 때문에 위와 같이 끊어서 보았다

다음은 시간 복잡도의 증가율을 살펴보자

https://hy6219.github.io/TIL-Today-I-Learned-/Algorithm/Asymptotic%20Analysis/TimeComplexity/Time%20Complexity%20Inc%20Ratio.html

위는 참고만 하도록 하고, 데이터가 커질수록 각 시간 복잡도가 어떻게 변화하는지만 확인해 보자

III-II. 시간 복잡도의 종류

1. 상수시간 O(1)

- 입력 크기와 상관없이 결과를 고정된(상수) 시간에 계산

(예시)

- 배열의 N번째 요소에 접근
- 스택에 넣고 빼기
- 큐에 삽입하고 삭제하기
- 해시 테이블의 원소에 접근하기

2. 선형시간 O(n)

- 알고리즘 실행 시간 ∝ 입력크기 n

(예시)

- 배열에서 원소 검색, 최솟값 찾기, 최댓값 찾기 등의 연산
- 연결 리스트에서 순회(traversal), 최솟값 찾기, 최댓값 찾기(단, 모든 노드를 방문하게 되는 경우 ≥ O(n)

3. 로그시간O(logn)

- 알고리즘 실행 시간 ∝ log(입력크기 n)
- 알고리즘의 각 단계에서 입력의 상당 부분(절반)을 방문하지 않고 지나감

(예시)

- 이진 검색(binary search) 알고리즘

4. N 로그 N 시간 O(nlogn)

- 알고리즘 실행 시간 ∝ 입력크기 n * log(입력크기 n)
- 입력을 절반 혹은 일부 비율로 나눌 때마다 각 부분을 독립적으로 처리

➡️ ∝분할정복

(예시)

- 병합정렬
- 퀵정렬의 평균적인 성능(최악: $O(n^2)$)
- 힙 정렬

5. 이차시간 $O(n^2)$

- 알고리즘 실행 시간 ∝ $입력크기^2$

- 각 원소를 다른 모든 원소와 비교

(예시)

- 버블정렬
- 선택정렬
- 삽입정렬

6. 지수 시간 $O(2^n)$

- 입력 데이터의 원소들로 만들 수 있는 모든 부분 집합 생성

7. 계승 시간 $O(n!)$

- 입력 데이터의 원소들로 만들 수 있는 모든 순열을 생성

🌟IV. 알고리즘의 함수 실행 시간 도출

1. 상수 : 시간복잡도 O(1)
2. 반복문 : 반복문 실행 시간 = 반복문 내 구문의 실행 시간 * 반복 횟수!
   각 반복문의 시간복잡도는 O(n)
3. 중첩 반복문: 시간복잡도 $O(n^c)$ (c는 반복문의 갯수)
4. 연속 구문 : 연속된 구문의 실행시간을 모두 합하기
5. if-else문 : if나 else 중 실행시간이 더 많이 걸리는 블록을 선택하고, 나머지는 무시
6. 로그 구문 : 각 반복마다 입력크기가 일정하게 감소

V. 시간 복잡도 예제

(1)

```
int fun1(int n)
{
  int m = 0 ;
  for(int i = 0 ; i < n; i++)
  {
    m+=1;
  }
  return m;
}
```

이 경우, for 반복문이 있기 때문에 시간 복잡도는 O(n)이 됩니다!

(2)

```
int fun2(int n)
{
  int i = 0, j = 0, m = 0 ;

  for(i = 0 ; i < n; i++)
  {
    for(j = 0; j < n; j++)
    {
        m+=1;
    }
  }
  return m;
}
```

이 경우, 중첩된 반복문을 사용하였고, (0,...,n-1)이 두번 반복되었기 때문에, 시간복잡도는 $O(n^c) = O(n^2)$ (c = 2= 반복문 갯수)가 된다

(3)

```
int fun3(int n)
{
  int i = 0 , j = 0, m = 0;
  for(i = 0 ; i < n; i++)
  {
    for(j = 0 ; j < i; j++)
    {
```

```
        m+=1;
      }
    }
    return m;
  }
```

이 경우, 무조건 두 개의 반복문이 있다고 시간 복잡도를 접근하지 말고, 천천히 생각해보자

바깥쪽의 반복문은 0, 1, ..., n-1 번째까지 수행하고, 내부의 반복문은

예제 3번 설명

i	j
0	-
1	0
2	0, 1
3	0, 1, 2
4	0, 1, 2, 3
...	
n-1	0, 1, 2, ..., n-2 → 총 (n-1)번

와 같이 $T = 0 + 1 + 2 + ... + (n-2) + (n-1) + n = (n(n+1))/2$ 번,

즉, $O((n(n+1))/2)$ 의 시간 복잡도를 갖는다.

이 때, c =3을 가정했을 때

$f(n) = 0.5n^2 + 0.5n$, $g(n) = n^2$ 에 대해서

$0.5n^2 + 0.5n$ ≤ $3n^2$

↔ 0 ≤ $2.5n^2 - 0.5n$

↔ 0 ≤ $5n^2 - n$

↔ 0 ≤ $n(5n-1)$

↔ n ≥ 0.2 에 대해서 f(n) ≤ cg(n) 성립

∴ $O(n^2)$ 의 시간 복잡도를 가짐

(4)

```
int func4(int n)
{
  int i = 0, int m = 0;
  i = 1;
  while(i < n)
  {
    m +=1;
   i *=2 ;
  }
  return m;
}
```

while을 통해서

- i= 0 → i =2
- i = 2 → i =4
- ...
- i = (n-1)/2 →i = (n-1)

와 같이 문제 공간을 절반으로 줄였기 때문에 시간복잡도는 $O(logn)$ 이다

(5)

```
int fun5(int n)
{
  int i = 0 , m = 0;
  i = n;
  while(i > 0)
  {
    m +=1;
    i/= 2;
  }
```

```
  return m;
}
```

위의 경우

- i = n ➡️ i = n/2
- i = n/2 ➡️ i = n/4
- i = n/4 ➡️ i = n/8

...

와 같이 점차 문제 공간을 절반씩 줄여나가고 있기 때문에 시간복잡도는 $O(logn)$ 이다

(6)

```
int fun6(int n)
{
  int i = 0, j = 0 , k = 0 , m = 0;
  for(i = 0 ; i < n; i++)
  {
    for(j = 0; j < n; j++)
    {
        for(k = 0; k < n; k++)
        {
          m+=1;
        }
    }
  }
  return m;
}
```

위의 중첩된 반복문에서는 3개의 반복문이 n 의 복잡도를 갖고 있기 때문에

n * n * n= n^3 에 해당되는 $O(n^3)$ 의 시간복잡도를 지닌다

(7)

```
int fun7(int n)
{
  int i = 0, j = 0 , k = 0 , m = 0;
  for(i = 0 ; i < n; i++)
  {
```

```
    for(j = 0 ; j < n; j++)
    {
      m+=1;
    }
  }
  for(i = 0 ; i < n; i++)
  {
    for(k = 0 ; k < n; k++)
    {
      m+=1;
    }
  }
  return m;
}
```

이 경우, 위의 중첩된 반복문과 아래쪽의 중첩된 반복문 모두 동일한 방법으로 구성되었다.

두 반복문 모두, 바깥의 반복문과 내부의 반복문 모두 n번씩 수행하기 때문에 시간복잡도는

$O(g(n)) = O(n * n) = O(n^2)$ 이 된다

따라서 $O(n^2) + O(n^2) = O(n^2)$ 이 된다

(8)

```
int fun8(int n)
{
  int i = 0, j = 0, m = 0;
  for(i = 0 ; i < n; i++)
  {
    for(j = 0 ; j < sqrt(n); j++)
    {
      m+=1;
    }
  }
  return m;
}
```

이 경우, n을 16으로 가정해보자

예제 8번 접근

n	i	j
16 → 4	0	0, 1, 2, 3
4 → 2	1	0, 1
…		
제목 없음	n-1	0, …, sqrt(value)-1

➡️ O(n * n^(1/2))=O(n^(3/2)) 이 시간복잡도가 된다!

(9)

```
int fun9(int n)
{
  int i = 0, j = 0, m = 0;
  for(i = n; i >0; i/=2)
  {
    for(j = 0; j < i; j++)
    {
      m+=1;
    }
  }
  return m;
}
```

위의 경우를 표로 정리, 생각해보면 아래와 같다

n = 8로 가정해보자

9번 접근

i	j
8	0, 1, 2, …,7
4	0, 1,2,3
2	0, 1

Aa i	j
1	0

중첩반복문의 내부 반복문을 먼저 보면, 1/2 씩 횟수가 줄어드는 것을 볼 수 있고, 이를 나타내보면

$n + n/2 + n/4 +$ ...의 형태로 나타내어 지는 것을 볼 수 있다. 즉, f(n)=n/k(k는 양의 상수) 라고 볼 수 있는데,

c = 3, g(n) = n, k = 64이면

↔ n/64 ≤ 3n

↔ n ≥ 0 이면 cg(n)이 f(n)의 상한이 될 수 있기 때문에

시간 복잡도는 O(n)이다

(10)

```
int fun10(int n)
{
  int i = 0, j = 0, m = 0;
  for(i = 0 ; i < n; i++)
  {
    for( j = i ; j > 0; j--)
    {
        m+=1;
    }
  }
  return m;
}
```

위의 경우,

10번 접근

Aa i	j
0	-
1	1

i	j
2	2, 1
3	3, 2, 1
...	
n-1	n-1, n-2, ...,1

으로 인해서 $T = 0 + 1 + 2 + ... + (n-1) = ((n-1)n)/2$

즉, 시간복잡도는 $O(n^2)$ 가 된다

(11) 🌟

```
int fun11(int n)
{
  int i = 0, j = 0 , k = 0 , m = 0;
  for(i = 0 ; i < n; i++)
  {
    for(j = i ; j < n; j++)
    {
      for(k = j + 1; k < n; k++)
      {
        m+=1;
      }
    }

  return m;
}
```

표를 통해 접근해보자

11번 접근방식

i	j	k
0	0	1,2,3,..., (n-1)
0	1	2,3,4,...,(n-1)
0	...	
0	n-1	-
1	1	2,3,4,...,(n-1)
1	...	

Aa i	j	k
1	n-1	-

$\therefore T_0 = i\text{가0일때} = (n-1) + (n-2) + ... + 1 = n(n-1)/2$

$T_1 = i\text{가1일때} = (n-2) + ... + 1 = (n-1)(n-2)/2$

∴ 내부 복잡도가 T_i=[n(n-1)+(n-1)(n-2)+...+2*1]/2, 외부 복잡도는 (0,...,(n-2)) 이기 때문에

전체 복잡도는 $T_t = n^3$

즉 $O(n^3)$ 이 된다

(12)

```
int fun12(int n)
{
  int i = 0, j = 0 , m = 0;
  for(i = 0 ; i < n; i++)
  {
    for(; j < n; j++)
    {
      m+=1;
    }
  }

  return m;
}
```

위의 경우에는

- i=0 ➡️ j= 0, 1,...,n-1
- 그런데 j를 초기화하지 않으므로 이로써 끝!

따라서 시간 복잡도는 O(n)!!

(13)

```
int fun13(int n)
{
  int i = 0, j = 0, m = 0;
  for(i = 1; i <= n; i *=2)
  {
    for(j = 0 ; j <= i; j++)
    {
      m+=1;
    }
  }
  return m;
}
```

표를 그려서 생각을 정리해보자

13번 접근

Aa i	j
1	0,1
2	0,1,2
4	0,1,2,3,4
8	0,1,2,3,…,8
..	
n	0,…,n : n+1번

따라서 내부에서 i에 대해서 n+k 번 형태로 돌아가기 때문에 시간복잡도는 O(n)이다!