시간복잡도

물론입니다! 아래는 **시간 복잡도**(Time Complexity)에 대한 간략한 정리와 요약입니다. 실무 및 알고리즘 학습을 위한 핵심 내용을 중심으로 정리했습니다.

📚 시간 복잡도 (Time Complexity) 요약

🔍 정의

시간 복잡도는 알고리즘이 입력 크기 `n` 가 증가할 때, **실행 시간이 얼마나 증가하는지**를 나타내는 지표입니다.
대부분 `O(n^k)` 형태로 표현되며, `n` 이 커질수록 성능이 얼마나 나빠지는지를 비교할 수 있습니다.

*이와 함께 언급되는게 공간복잡도 도 있습니다.

📊 주요 시간 복잡도 종류 (낮음 → 높음)

복잡도	의미	예시	
O(1)	상수 시간	배열의 첫 원소 접근 (`arr[0]`)	
O(log n)	로그 시간	이진 탐색, 힙 정렬	
O(n)	선형 시간	배열 순회, 단순 검색	
O(n log n)	선형 로그	병합 정렬, 힙 정렬	
$O(n^2)$	제곱 시간	두 반복문 (예: 2중 루프)	
$O(2^n)$	지수 시간	모든 부분 집합 탐색 (브루트포스)	
O(n!)	팩토리얼 시간	모든 순열 탐색 (최악의 경우)	
Θ(n)	정확한 선형 성능	단순 순회 + 조건 검사	입력 크기에 정확히 비례
Ω(n)	최소 **n** 이상	검색이 항상 최소 1번 실행됨	최선의 경우를 보장할 때
Ω(1)	최소 상수 시간	해시맵에서 키 찾기 (최선 경우)	최소 1ms 내로 응답 가능
Ω(log n)	최소 로그 시간	이진 탐색의 최선 경우 (전체 탐색 없이 바로 찾음)	최선의 경우가 최소 로그 시간이면 가능
$Ω(n^2)$	최소 제곱 시간	두 반복문이 반드시 실행될 때 (예: 모든 쌍 비교)	최악의 경우보다 더 느려질 수 있음

✅ **Big O (O)**: 최악의 경우(worst-case)를 기준으로 **상한**(upper bound)을 나타냄
✅ **Ω (오메가)**: 최선의 경우(best-case)를 기준으로 **하한**(lower bound)을 나타냄
✅ **Θ (세타)**: **최악과 최선의 경우**가 거의 동일할 때, **정확한 경계**(tight bound)를 나타냄

> ✅ 일반적으로 **O(n log n)** 이하가 효율적이고, **$O(n^2)$** 이상은 큰 데이터에서는 비효율적

📈 복잡도 비교 (입력 크기 증가 시)

입력 크기 `n`	O(1)	O(n)	$O(n^2)$	O(n log n)	$O(2^n)$	O(n!)
n = 10	1ms	10ms	100ms	~100ms	~100ms	~3.6ms
n = 100	1ms	100ms	10,000ms (10초)	~100ms	무한	매우 큰 값

> 💡 예: `n=20` 일 때 `O(2ⁿ)` 은 약 10^6, `O(n!)` 은 약 10^{18} → 현실적으로 불가능

📌 주요 특징 및 활용

복잡도	장점	한계
O(1)	초당 응답 가능	데이터 구조에 의존 (예: 해시맵)
O(log n)	입력 크기 증가에도 빠름	이진 트리, 이진 탐색 필요
O(n)	간단하고 실용적	큰 데이터에서는 느려질 수 있음
O(n log n)	대부분의 정렬 알고리즘	최적 수준 (예: 병합 정렬)
$O(n^2)$	간단한 문제에 유용	n=1000 이상에서는 지연 발생
$O(2^n)$, O(n!)	정확한 해를 보장	실무에서 **절대 사용 금지** (예: 모든 조합 탐색)

📝 실생활 예시

문제	사용하는 복잡도	이유
검색 (검색 목록)	O(n)	순회하며 찾기
이진 탐색	O(log n)	반씩 줄여서 찾기
정렬 (병합 정렬)	O(n log n)	효율적인 정렬 방법
모든 부분 집합 찾기	$O(2^n)$	모든 조합을 탐색해야 하므로 비효율적
전체 순열 찾기	O(n!)	n=10 이상에서는 불가능

문제	사용하는 복잡도	이유
이진 탐색 (최선)	$\Omega(\log n)$	최소 1번 비교로 찾을 수 있음 (예: 정렬된 배열)
선택 정렬 (최선)	$\Omega(n^2)$	최소 1번 비교로 시작하므로 제한적
검색 (해시맵)	$\Omega(1)$	최선 경우 1번 비교로 찾음 (해시 충돌 없음)
전체 순열 탐색	$\Omega(n!)$	최소 1개의 순열을 탐색해야 하므로

📌 핵심 요점 정리

복잡도	의미	중요성
O(n)	최악의 경우 최대	알고리즘의 상한을 나타냄 (실무에서 주의)
Ω(n)	최선의 경우 최소	최소 성능 보장 (예: 최소 1번 반복)
Θ(n)	정확한 경계	알고리즘이 효율적이고 일관성 있음 (예: 정렬)

> ✅ **Big O는 "최악의 경우"를 보여주고,**
> ✅ **Ω는 "최선의 경우"를 보여주고,**
> ✅ **Θ는 "정확한 경계"를 보여줌** → 알고리즘의 성능을 더 정확하게 평가 가능

📌 시간 복잡도를 이해하는 3가지 질문

1. **입력 크기가 10배 증가하면, 실행 시간이 얼마나 증가할까?**
   → $O(n^2)$은 100배 증가, $O(2^n)$은 지수 증가 (매우 빠르게 증가)
2. **어떤 문제는 브루트포스로 풀 수 있나요?**
   → 입력 크기가 작고, 경우의 수가 적을 때 (예: 4자리 패스워드, 10개 중 3개 고르기)
3. **효율적인 알고리즘은 어떤 것인가요?**
   → $O(n \log n)$ 이하 → **병합 정렬, 힙 정렬, 이진 탐색**

📚 관련 태그 (Obsidian에 추가할 경우)

- `#시간복잡도`
- `#알고리즘`
- `#O_n`
- `#복잡도`
- `#효율성`
- `#Big O`

💡 팁

- **시간 복잡도는 최악의 경우**(worst-case)를 기준으로 평가
- **입력 크기 n이 1000 이상일 때는 O(n²) 이상은 피하고 O(n log n) 이하를 선호**
- **실제 성능은 하드웨어, 언어, 최적화 등에 영향** → 복잡도는 이론적 기준
- **Big O만 보는 것은 부족** → 최선/최악/평균 성능을 함께 고려해야 함
- **Θ(n)** 은 알고리즘이 **정확한 효율성**을 보임을 의미 (예: 병합 정렬)
- **Ω(n)** 은 **최소 성능 보장** → 예를 들어, "최소 1초 이상 소요" 같은 경우에 유용
- **실무에서 O(n log n)** 이하를 선호하되, **Θ(n)** 은 더 신뢰할 수 있음

📌 예제: 정렬 알고리즘 (병합 정렬) – O(n log n), Θ(n log n), Ω(n log n)

```c
// C: 병합 정렬 (최악/최선/평균 모두 O(n log n) → Θ(n log n))
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int left, int mid, int right);
void merge_sort(int arr[], int left, int right);

void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++) L[i] = arr[left + i];
    for (j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    i = j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
    free(L); free(R);
}

void merge_sort(int arr[], int left, int right) {
```

```cpp
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

> ✅ 복잡도:
>
> - **$O(n \log n)$** (최악)
> - **$\Omega(n \log n)$** (최선)
> - **$\Theta(n \log n)$** (평균)
>   → 정확한 경계를 보여주는 대표적인 예

📌 예제: 선택 정렬 (최선/최악/평균)

```cpp
// C++: 선택 정렬 (최선: O(n²), 최악: O(n²), 평균: O(n²))
#include <iostream>
#include <vector>
#include <algorithm>

void selection_sort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        if (min_idx != i) {
            std::swap(arr[i], arr[min_idx]);
        }
    }
}
```

> ✅ 복잡도:
>
> - **$O(n^2)$** (최악, 평균)
> - **$\Omega(n^2)$** (최선) → 이미 정렬된 배열일 때도 `n-1` 번 반복 → 최소 `n²/2` 비교
> - **$\Theta(n^2)$** → 평균/최악/최선 모두 비슷한 성능

📌 예제: 이진 탐색 (최선/최악/평균)

```python
# Python: 이진 탐색 (최선: O(1), 최악: O(log n), 평균: O(log n))
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    # 최선 경우: 첫 번째 비교에서 찾음 (예: arr[0] == target)
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # 찾지 못함
```

> ✅ **복잡도:**
>
> - **O(log n)** (최악)
> - **Ω(1)** (최선) → 첫 번째 비교에서 바로 찾을 경우 (예: `arr[0] == target`)
> - **Θ(log n)** (평균) → 정렬된 배열에서 평균적으로 `log n` 번 비교

📌 예제: 해시맵 검색 (최선/최악)

```cpp
// C++: 해시맵 (해시 충돌 없을 때 최선 경우 O(1))
#include <unordered_map>
#include <iostream>

void hash_search() {
    std::unordered_map<int, std::string> map;
    map[100] = "value1";
    map[200] = "value2";

    auto it = map.find(100);
    if (it != map.end()) {
        std::cout << "Found: " << it->second << std::endl;
    }
}
```

> ✅ **복잡도:**
>
> - **O(1)** (최악, 최선, 평균) → 해시 충돌 없을 때
> - **Ω(1)** (최선) → 최소 1번 비교로 찾음
> - **Θ(1)** → 정확한 상한/하한

📌 예제: 두 반복문 ($O(n^2)$, $\Omega(n^2)$)

```python
# Python: 두 반복문 (최악/최선/평균 모두 O(n²))
def print_pairs(arr):
    n = len(arr)
    for i in range(n):
        for j in range(i, n):
            print(f"({i}, {j})")
```

> ✅ **복잡도**:
>
> - **$O(n^2)$** (최악)
> - **$\Omega(n^2)$** (최선) → `i=0` 부터 시작해 `j` 가 `n-1` 까지 반복할 때
> - **$\Theta(n^2)$** → 평균적으로 `n²/2` 번 반복

📌 요약: 복잡도와 코드 관계

코드	최악	최선	평균	복잡도 종류
병합 정렬	$O(n \log n)$	$\Omega(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
선택 정렬	$O(n^2)$	$\Omega(n^2)$	$O(n^2)$	$\Theta(n^2)$
이진 탐색	$O(\log n)$	$\Omega(1)$	$\Theta(\log n)$	$\Theta(\log n)$
해시맵	$O(1)$	$\Omega(1)$	$\Theta(1)$	$\Theta(1)$
두 반복문	$O(n^2)$	$\Omega(n^2)$	$O(n^2)$	$\Theta(n^2)$

> ✅ **코드를 통해 복잡도를 직접 확인 가능**
> ✅ **최선/최악/평균**을 구분하여 **Ω, O, Θ**를 정확히 이해할 수 있음

💡 **Tip**:

- **$\Omega(n)$** 은 "최소 n번 반복"을 보장하는 경우 (예: 최소 1번 검색)
- **$\Theta(n)$** 은 "정확한 경계"를 보여줌 → 실무에서 신뢰할 수 있음
- **$O(n)$** 은 "최악의 경우"를 보여줌 → 실무에서 **최소한의 성능 보장** 필요할 때 사용