Time Complexity (시간 복잡도)

시간 복잡도는 특정한 크기의 입력에 대하여 알고리즘이 얼마나 오래 걸리는지를 의미하며, 알고리즘을 위해 필요한 연산의 횟수를 의미한다. 우리가 만약 어딘가로 이동할 때 최단 경로로 이동한다면 대개는 최단 시간으로 이동하게 될 것이다. 그렇다면 우리는 최단 경로, 최단 시간으로 이동하기 위해 **'알고리즘'**을 작성 해야하고 또 이 알고리즘이 수행되는 연산의 횟수인 **'시간 복잡도'** 를 고려해야한다.

시간 복잡도는 3 가지 경우로 나타낸다.

최선의 경우 (Best Case)

- 빅 오메가 표기법 사용
- 최선의 시나리오로 최소 이만한 시간이 걸림
- 만약 정해진 연산의 수는 n^2 번이지만, n 번만 수행해도 된다면 오메가 표기법에 의해 n 의 시간 복잡도를 갖게 된다.

최악의 경우 (Worst Case)

- 빅 오 표기법 사용
- 최악의 시나리오로 아무리 오래 걸려도 이 시간보다 덜 걸림
- 만약 n 번만 수행해도 되는 프로그램이지만, 정해진 연산의 수가 n^2 번이라면, 불필요한 연산까지 수행하게 되고 n^2 의 시간 복잡도를 갖게 된다.

평균적인 경우 (Average Case)

- 빅 세타 표기법 사용
- 빅 오메가와 빅 오 표기법의 평균 시간을 나타냄

우리는 보통 알고리즘의 시간 복잡도를 나타낼 때 빅오 표기법을 이용한다. **빅오 표기법**은 **최악의 경우를 고려**하므로, 프로그램이 실행되는 과정에서 소요되는 **최악의 시간까지 고려**할 수 있기 때문이다.

- O(1)
```
* System.out.printf("%d", num);
* -> 한번의 연산 수행.
```
- O(log n)

  $O(\log_x n)$은 로그 복잡도(logarithmic complexity)라고 부르며, Big-O표기법중 O(1) 다음으로 빠른 시간 복잡도를 가진다.

```
*  for(int i=1; i<n; i*=2)
*       System.out.printf("%d", num);
* -> 반복문이 실행되면 i는 2의 배수 단위로 증가, O(log2n)의 복잡도를 갖게 됨.
```

- O(n)

  O(n)은 선형 복잡도(linear complexity)라고 부르며, 입력값이 증가함에 따라 시간 또한 같은 비율로 증가하는 것을 의미한다.

```
*  for(int i=0; i<n; i++)
*       System.out.printf("%d", num);
* -> 반복문이 실행되면 출력문이 n번 반복됨.
```

- $O(n^2)$

  $O(n^2)$은 2차 복잡도(quadratic complexity)라고 부르며, 입력값이 증가함에 따라 시간이 n의 제곱수의 비율로 증가하는 것을 의미한다.

```
*   for(int i=0; i<n; i++){
*        for(int j=0; j<n; j++)
*            System.out.printf("%d", num);
*          }
* -> 반복문이 실행되면 안쪽 반복문에서 n번의 연산을 수행하는데
*    이때 안쪽 반복문은 바깥 반복문에 의해 n번만큼 안쪽 반복문이 수행된다.
*    즉, n*n번의 연산이 이루어지기 때문에 시간 복잡도는 O(n2)가 된다.
```

점근적 표기법!! 최고차 항의 계수와 불필요한 상수나 나머지 항들을 제거하는 방식이다.

시간 복잡도를 구하는 방법

1. 먼저 프로그램의 연산 횟수를 구한다.
2. 만약 $5n^2 + 3n + 1$의 연산을 수행하는 프로그램이라면,
3. 점근적 표기법에 따라 $O(n^2)$의 시간 복잡도로 나타낼 수 있다.