구디 아카데미
알고리즘 스터디
중급반 - 2주차

[멘토 유준혁]

스터디 주제

- 알고리즘 분석의 시간복잡도와 공간복잡도
- 1차원 배열의 정의와 이론
- 문자열의 정의와 메모리구조
- StringBuilder (StringBuffer) 와 문자열의 차이점

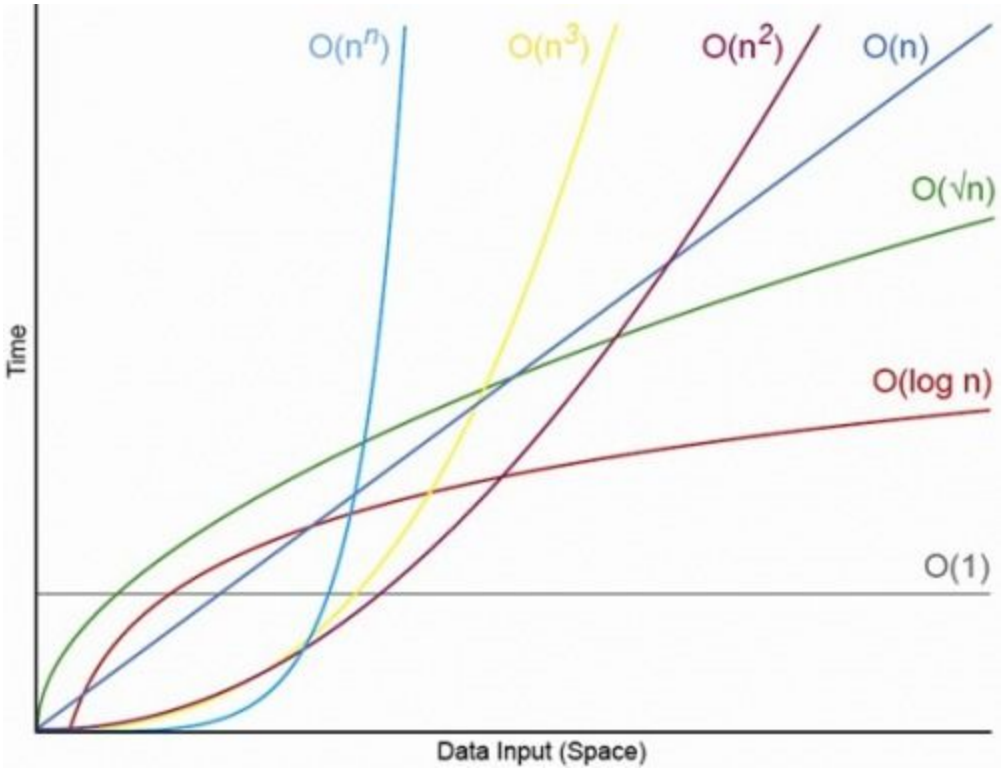
알고리즘 분석 - 시간복잡도와 공간복잡도

- 알고리즘 분석에는 시간 복잡도(Time Complexity)와 공간 복잡도(Space Complexity)를 이용한다.
- 시간 복잡도 : 핵심 알고리즘이 수행되는 총 수행시간 **(반복의 횟수)**
- 공간 복잡도 : 알고리즘이 수행되면서 차지하는 **메모리의 총량**
- 기업 코딩테스트에선 시간 복잡도를 노리는 문제가 종종 출제된다. (공간은 가아아끔 나옴)

인스턴스특성 n							
시 간	이 름	1	2	4	8	16	32
1	상수형	1	1	1	1	1	1
log n	로그형	0	1	2	3	4	5
n	선형	1	2	4	8	16	32
n log n	선형로그형	0	2	8	24	64	160
n^2	평방형	1	4	16	64	256	1024
n^3	입방형	1	8	64	512	4096	32768
2^n	지수형	2	4	16	256	65536	42949672976
n!	계승형	1	2	24	40626	20922789888800	26313×10^{33}

http://blog.naver.com/loudon23

↑사진1. 인수 n에 따른 각 시간복잡도의 완료 시간 | ↓사진2. 시간복잡도의 기울기 그래프



시간복잡도의 예제

1부터 100까지의 합을 산출해내는 문제를 마주했다고 가정해보자.

가장 보편적인 방법은 1부터 100까지 모두 더하여 출력하는 알고리즘이 되겠다.

```
int sum = 0;

for (int i = 1; i < 101; i++)
    sum += i;

System.out.println(sum);
```

1부터 100까지 모두 더하는 코드

두번째 방법은 **가우스의 일화로 유명한 등차수열의 합 공식**으로 푸는 방법이 있다.

```
int i = 100;
System.out.println((i * (i + 1)) / 2);
```

1부터 100까지 모두 더하는 코드

현재는 100정도밖에 되지 않기 때문에 시간차이가 얼마 나지 않겠지만
만약, 1부터 10억까지의 합을 구하려고 한다면 어떻게 될까?

첫번째 방식은 총 10억번의 연산이 필요하게 되고 - **빅오 표기법 [O(n)]**
두번째 방식은 **숫자의 크기와는 상관없이 일정한 연산**이 필요하다. - **빅오 표기법 [O(1)]**

이처럼 동일한 결과를 내는 알고리즘이라고 해도
어떤 방식을 사용하느냐에 따라 수행속도가 현저한 차이를 보인다는 것을 알 수 있다.
그렇지만 항상 시간복잡도를 낮추려고만 하지는 않아도 된다. (고민은 해보는 게 좋지만)
범위가 크지 않은 경우에는 $O(n^3)$ 알고리즘을 작성해도 상관없는 문제들이 더러 존재한다.

공간복잡도의 예제

공간복잡도 같은 경우엔 자주 출제되는 문제는 아니다.
나날이 PC의 성능이 좋아지고 있는 요즘엔 거의 무의미 하다고 생각되기도 한다.
그럼에도 불구하고 종종 출제되는 이유는 String 클래스의 특징 때문이라고 생각하면 된다.

스터디 시간에도 소개했다시피 **String은 기본 자료형이 아니라 참조 자료형**이다.
일정한 바이트의 크기를 할당하고 값을 바꾸는 기본 자료형들과는 다르게,
String 크기만큼 새로운 메모리를 할당 시킨 후 주소값을 참조하는 방식으로 변수 할당을 한다.

초기 수업시간에 했던 대표적인 예제를 들어보도록 하자.
양의 정수 n을 입력받고 1부터 n번째 줄 까지 * 을 출력하는 '별 찍기 문제'를 예로 들겠다.

```
String star = "";
for (int i = 1; i < n; i++) {
    star += "*";
    System.out.println(star);
}
```

n개의 줄에 i개의 별을 찍는 코드

별 문제가 없어 보이는 위 코딩 방식은 사실 대단히 위험한 방식이다.
n이 억에 가까운 수로 배정이 된다면 새로운 String을 그만큼 새로 할당해야 하기 때문이다.

이와 같은 위험을 피하기 위해 주로 사용되는 게 StringBuilder 혹은 StringBuffer이다.
(둘의 차이점은 Thread에 safe하냐의 차이라고 생각하면 된다.)

String의 경우엔 크기 만큼의 메모리를 할당시켰다면,
StringBuilder(Buffer)는 객체가 생성될 때 충분한 크기를 할당해주고 빈 공간을 채우도록 한다.
StringBuilder 또한 담을 수 있는 크기를 벗어나게 된다면 새로운 공간을 할당시키게 되지만
상대적으로 String보다 여유가 있기 때문에 메모리 면에서는 효율적이게 된다.

****String에 관련된 문제는 (거의) 필수적으로 물어보기 때문에**

문제를 많이 풀어보면서 관련된 메서드와 감각을 익히는 게 중요하다.

될 수 있는 한 많이 풀어보는 걸 추천한다.

1차원 배열

알고리즘 문제에서 가장 많이 쓰이는 자료구조 중 하나가 바로 배열이다.
동일한 자료형들을 담아두고, 변경시키고, 값을 가져올 수 있기 때문이다.
인덱스를 통해 접근이 가능하기에 값을 참조하는 데에 있어 O(1)의 시간복잡도를 가지는 이점이 있고
범위를 벗어나면 **IndexOutOfBoundException**을 발생시키기에 인수의 범위도 체크가 가능하다.

Collection 중에도 이와 같은 이점을 가지는 동일한 자료구조가 존재한다.
바로 List이다.
배열과 동일하게 인덱스를 통해 접근이 가능하며, 범위를 벗어나면 Exception을 Throw 한다.

둘의 차이점이라고 한다면 **메모리의 정적 / 동적 할당** 의 차이가 가장 크다.

배열의 경우엔 한 번 크기를 할당하게 되면 크기를 재설정 할 수 없다.
반면에 List의 경우는 동적으로 메모리를 할당시키기에 원하는 만큼 add를 할 수 있게 된다.
(후에 나오는 Tree 구조 같은 경우 List를 활용하기도 한다.)

따라서 **인수의 갯수가 정해지지 않은 경우 List**를,
갯수가 정해져있다면 배열을 사용하면 된다.

배열을 좀 더 사용하기 쉽게 해주는 Arrays 클래스와,

Array와 List를 연결해주며 반복작업을 쉽게 만들어주는 Stream에 대해서는

트렐로에 관련 게시글을 올려두었으니 참고하기를 바란다.