구디 아카데미 알고리즘 스터디 중급반 - 4주차

[멘토 유준혁]

스터디 주제

- 재귀 함수 (Recursive Function)
- 문제 내에서의 규칙을 찾는 방법들
- 브루트 포스 설명

재귀 함수란 무엇인가?

함수 내에서 자기 자신을 재호출하는 방식을 재귀적 호출, 또는 재귀 함수라고 부른다.

잘 모르겠다면 다음 예제를 살펴보자.

> 어느 한 컴퓨터공학과 학생이 유명한 교수님을 찾아가 물었다.
>
> "재귀함수가 뭔가요?"
>
> "잘 들어보게. 옛날옛날 한 산 꼭대기에 이세상 모든 지식을 통달한 선인이 있었어.
>
> 마을 사람들은 모두 그 선인에게 수많은 질문을 했고, 모두 지혜롭게 대답해 주었지.
>
> 그의 답은 대부분 옳았다고 하네.

그런데 어느 날, 그 선인에게 한 선비가 찾아와서 물었어.

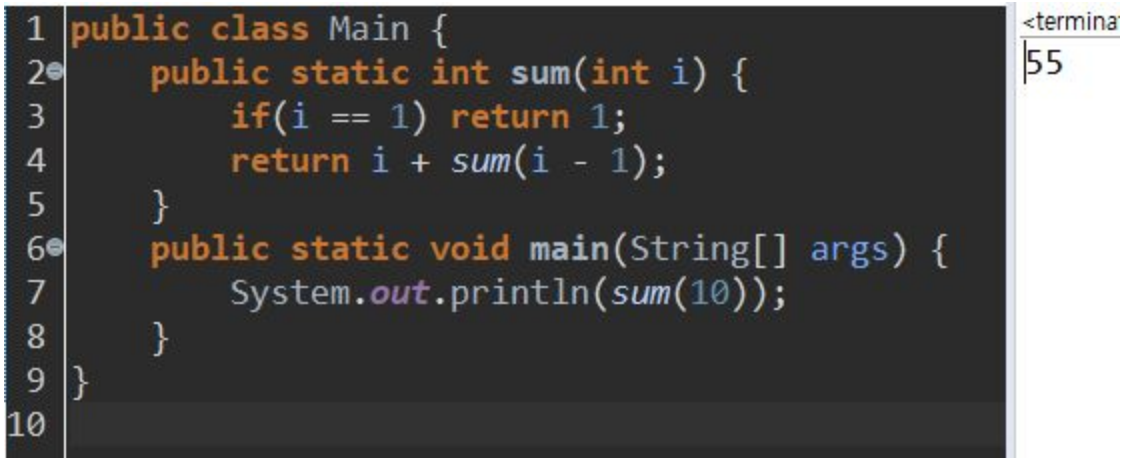
"재귀함수가 뭔가요?"

"잘 들어보게. 옛날옛날 한 산 꼭대기에...

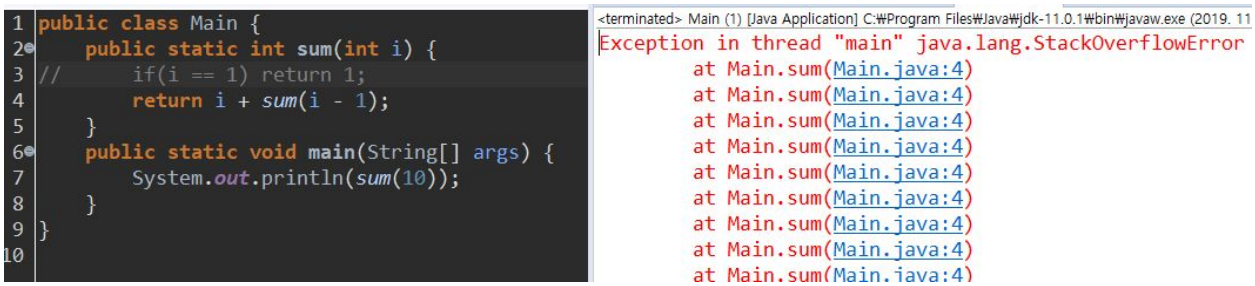
(꽤 유명한 재귀 함수 설명)

이 처럼 끝 없이 자기 자신을 재 호출하는 경우를 재귀 함수 라고 일컫는다.

종료 조건이 제시되지 않는다면 무한루프 **(정확히는 StackOverFlow 발생)** 에 빠지게 되버린다.



(종료 조건을 제대로 걸었을 때의 재귀함수)



(종료 조건을 걸지 않았을 때의 재귀함수)

얼핏 보면 일반적인 반복문과 다른 점이 없어보인다.

반복문도 재귀 함수와 똑같이 종료 조건을 제대로 걸지 않는다면 무한루프에 빠지기 때문이다.

그렇다면 재귀 함수는 왜 쓰일까?

반복문과의 차이점

1. 분명한 종료조건이 존재하고, 이를 만나면 반복이 종료된다.
2. 종료조건에 해당하는 변수는 값이 종료조건에 수렴하는 형태로 진행해야 한다.
3. 반복이 진행중인 동안 다음 코드는 실행되지 않는다.

위의 3가지 조건은 반복문이 가지는 기본적인 조건들이다.

이미 for문과 while문을 통해 개념들은 습득하셨을 거라 생각하여 부가적인 설명은 하지 않겠다.

그렇다면 재귀 함수는 어떤가? 위의 조건들에 부합하지 않는 게 있나..?

없다. 반복문과 기본적인 조건들이 동일하다.

함수가 메모리 (스택) 에 등재되는 걸 생각해보면 3번의 조건 또한 충족하게 된다.

심지어 스택 메모리 공간을 사용하게 되는 재귀함수 특성상 메모리 면에서는 일반 반복문이 더 낫다.

이를 바꿔말하면 반복문으로 구현하는 코드들은 재귀 함수로도 구현이 가능하며,

재귀 함수 코드 역시 반복문으로 구현이 가능하다는 말이 된다.

그렇다면 '함수' 에 차이가 있는 걸까?

제대로 정의된 함수라면 매개변수를 통해 값을 전달받아, 이를 가공하여 return 하게 된다.

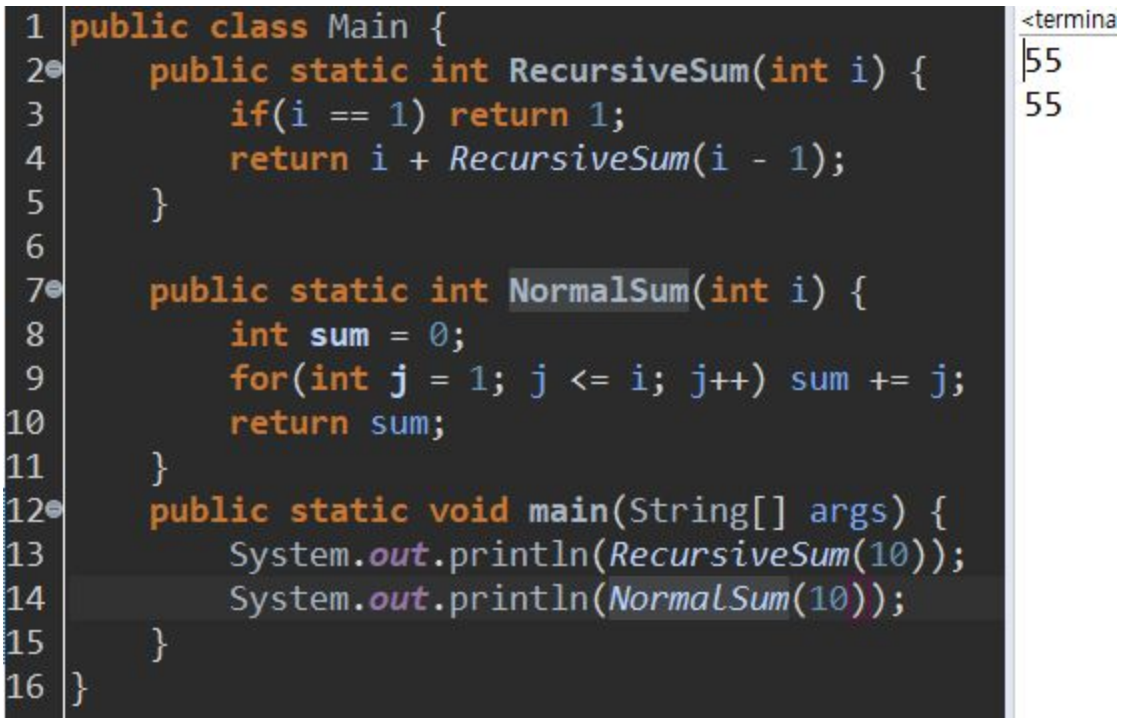
동일한 알고리즘이 여러번 사용된다면 이를 함수로 만들어두었다가

매개변수 값을 조정하여 재사용하는 게 보편적인 방식이다.

이와 같은 점에서 재귀 함수가 더 이점을 갖게 된다.

정말 그럴까?

다음의 코드를 살펴보자.



```java
public class Main {
    public static int RecursiveSum(int i) {
        if(i == 1) return 1;
        return i + RecursiveSum(i - 1);
    }

    public static int NormalSum(int i) {
        int sum = 0;
        for(int j = 1; j <= i; j++) sum += j;
        return sum;
    }
    public static void main(String[] args) {
        System.out.println(RecursiveSum(10));
        System.out.println(NormalSum(10));
    }
}
```

```
<termina
55
55
```

(동일한 결과를 return 하는 두 개의 함수)

위는 재귀 함수로 구현한 N까지의 합이고, 아래는 일반 반복문을 함수화 시켰을 뿐이다.

이를 통해 재귀 '함수' 가 갖던 이점 역시 반복문도 동일하게 가질 수 있게 되었다.

…. 아무리 봐도 차이점이 없어보인다.

그렇다면 소스 상에서의 차이점을 살펴보도록 해보자.

자세히 살펴보면 두 개의 코드는 분명한 차이점을 가진다.

첫번째로 재귀 함수가 코드의 라인이 더 짧다.

두번째로 변수의 사용이 적다.

코드의 라인이 더 짧다는 말은 코드가 상당히 직관적으로 구성되었다는 말이 된다.

재귀 함수를 처음 접한다면 **멍멍이소리하네!** 하겠지만.. 실제로 직관적으로 짜여지게 된다.

이는 알고리즘이 갖는 점화식 때문에 그렇다.

일반적으로 알고리즘 문제를 풀 때는 점화식이라는 것을 우선적으로 파악하게 되는데,

후에 이 **점화식을 그대로 코드로 옮기게 되면 위와 같은 재귀 함수가 구현되게 된다.**

위의 재귀 함수를 점화식으로 나타내본다면 다음과 같다.

F(n) = F(n-1) + n　(1부터 n까지 자연수의 합을 구하는 함수의 점화식)

n까지의 합을 구하려면, n-1까지의 합을 구한 값에 + n을 해주면 된다는 말이다.

여기서 종료 조건인 n == 1 까지 부여된다면 위의 재귀함수가 그대로 구현되는 걸 볼 수 있다.

따라서 점화식을 토대로 코드를 작성하는 경우, 상당히 빠른 시간 내에 코드를 짤 수 있다는 말이 된다.

게다가 만약 코드가 정상적으로 작동된다는 것을 증명해야 할 때

점화식과 코드의 상관관계를 보여주며 증명이 가능하게 된다.

두번째로 변수의 사용이 적다는 점을 들 수 있다.

다음 문제는 실제로 **네이버 오프라인 면접**에서 출제 된 문제라고 한다.

int형 변수 n을 사용해서 1부터 n까지의 합을 구하는 메서드를 작성하려고 한다.

이때 추가적인 변수의 할당이 있으면 안된다고 할 때, 해당 메서드를 작성하려면 어떻게 해야 할까?

재귀 함수에 대한 기본적인 정보를 가지고 있지 않다면 충분히 헤맬만한 문제이다.

이 처럼 변수의 사용을 줄여주고, 알고리즘을 직관적으로 파악하기 용이하다.

라는 이점을 가지기 때문에 알고리즘 문제에서 재귀 함수가 자주 사용되게 된다.

그래서 점화식이 뭔데?

점화식을 토대로 작성한 함수가 재귀 함수라고 하는 건 알겠다.

그럼 점화식이란건 뭘까?

위에서 잠깐의 예시를 통해 단편적으로나마 느꼈겠지만, 해당 알고리즘이 갖는 규칙을 식으로 나타낸 것을 점화식이라고 한다.

더 자세한 정의로는 **어떠한 함수를 자신보다 더 작은 변수에 대한 함수와의 관계로 표현한 식** 이다.

위와 마찬가지로 1부터 N까지의 자연수들의 합을 구하는 함수가 있다고 해보자.

N의 변수값이 5, 4, 3일때의 총 식은 다음과 같다.

$$f(5) = 5 + 4 + 3 + 2 + 1$$

$$f(4) = \quad 4 + 3 + 2 + 1$$

$$f(3) = \quad\quad 3 + 2 + 1$$

이를 점화식의 정의대로 함수화 시켜보자. (단순화 한다고 생각해도 무방하다.)

점화식의 정의에서 자신보다 더 작은 변수에 대한 함수와의 관계로 표현한다고 했으니

f(5) = f(4) ~~~ 가 될 것이다.

f(4) 와의 차이점이라곤 +5 밖에 없다.

이는 *f(5)* 에서의 매개 변수 값과 같다.

마찬가지로 *f(4)* 역시 동일한 표현이 가능하다.

따라서 다음과 같이 줄일 수 있다.

$$f(n) = f(n-1) + n \text{ (1부터 n까지의 자연수들의 합을 구하는 점화식)}$$

이처럼 해당 문제의 핵심 알고리즘을 정형화 or 단순화 하여 식으로 나타낸 것을 점화식 이라고 한다.

앞에서도 설명했듯이 점화식을 작성한 뒤 재귀 함수로 구현하게 되면 시간이 단축되고 직관적이 된다.

규칙을 찾는 방법들

재귀 함수의 기본적인 사용 방법도, 점화식의 도출 방법도 알겠다.

그런데 점화식을 파악하는 데 있어 가장 중요하다고 볼 수 있는 ***규칙 찾기****를 못하겠다.*

여기는 사실 멘토가 어떻게 해줄 수 있는 부분이 아니다..

몇 번 해보다보면 감이 잡히겠지만 직관을 굉장히 필요로 하는 분야다.

차이점을 단번에 파악하는 능력이 필요하기 때문에 넓은 범위의 그림을 보는 연습이 필요할 듯 싶다.

팁이라고 하기엔 모호하지만.. 공유를 해보자면

멘토 역시 문제를 풀다가 규칙 찾기에서 막혀 끙끙대던 문제들이 있었다.

그럴때면 우선 문제 푸는 걸 멈추고 내가 그 현장에 있다면? 이라는 상상을 하곤 했다.

대부분의 문제들이 사람이 풀긴 쉬우나 그걸 코드로 짜기가 어려운 문제들이기 때문에

내가 직접 데이터를 손으로 이리저리 조작하면서 푼다고 생각하면 의외로 손쉽게 풀린다.

상상 속에서 직접 푸는 걸 끝마쳤다면 이젠 복기의 시간을 가진다.

데이터를 옮기거나 조작하는 행위들 사이의 인과관계를 분석한다고 보면 되겠다.

백준 온라인 저지 _ 재귀 _ 피보나치 수 5

예를 들어 위와 같은 문제를 만났고 점화식을 도출하지 못하고 있는 중이라고 가정하고

10번째의 피보나치 수를 구하려고 할 때 나라면 어떻게 행동할까? 를 따져보자.

```
// 피보나치 수 란

  첫번째는 0, 두번째는 1로 시작하면서 n번째의 수는 n - 1번째 수 + n - 2번째 수인 수열을 뜻한다.
```

1. 우선 땅이나 공책에 0 과 1을 쓴다.

2. 그 뒤에 0 과 1을 더한 1을 쓴다.

3. 그 뒤에 1 과 1을 더한 2를 쓴다.

4. 위의 과정을 10번째의 수를 찾을 때 까지 반복한다.

이런 과정을 겪게 될 것이 자명하다.

그럼 왜 위의 과정을 겪게 되는 걸까?를 생각해보자.

- 9번째의 수와 8번째 수가 무엇인지 모르기 때문에 (n = n-1 + n-2)
- 위와 마찬가지로 8번째… 3번째 까지의 수가 무엇인지 모르기 때문에.
- 확실하게 알고 있는 건 1번째와 2번째이기 때문에.
- 1번째와 2번째를 알고있다면 3번째의 수를 알 수 있기 때문에.
- 위와 마찬가지로 2번째와 3번째 수를 알고있다면 4번째 수가 도출 가능하기 때문에.
- 찾으려 하는 수는 10번째의 수이기 때문에.

따라서 10, 9, 8 … 4, 3 번째의 수를 알려면 1번째와 2번째 수에서부터 계산하며 올라가야 한다.

라는 전제를 찾을 수 있게 된다.

글로 옮겨적으려니 제대로 전달이 되지 않을까 싶어 걱정이 된다…

어찌됐건 간에 하고 싶었던 말은 **여러분의 직감을 믿어라** 였었다.

직감은 인간이 가진 뛰어난 능력 중 하나다. 직면한 문제의 조건들을 따지기 전에 이미 여러분은 그 문제를 어떻게 해결해야 할 지 알고있다. 단지 이를 코드로 옮기기 어려워서 그렇지 ㅠ

그러니 만약 막히는 구간이 있다고 한다면 직감을 최대한 활용하기를 바란다.

브루트 포스 (Brute-Force) 알고리즘

이번엔 브루트 포스 알고리즘에 대해서 알아볼 차례다.

정보처리자격증을 공부하신 분이라면 한 번쯤 봤을 그 단어가 맞다.

무작위 대입 공격 이라고도 불리는 브루트 포스는 생각보다 단순한 알고리즘이다.

“가능한 경우의 수를 다 해보면 언젠간 맞겠지”

어이가 없겠지만 진짜 이런 의미를 가진 알고리즘이 맞다.

만약 문제를 풂에 있어 시간제한이 전혀 없는 문제라고 한다면

브루트 포스 알고리즘을 사용해서 **언젠간! 반드시!** 맞출 수 있다. (100년이 넘게 걸릴 수도 있다.)

이건 여담이지만 낡아보이는 방식임에도 불구하고 해킹에선 굉장히 자주 쓰인다.

비밀번호 안전성 테스트 사이트

못믿겠다면 위의 사이트에서 본인의 비밀번호를 입력해보자.

이처럼 간단하고 비교적 쉬운 알고리즘을 사용하는 데에는 이유가 있다.

위에서 길게 설명했던 것 과는 다르게

점화식의 도출이 불가능하고, 일정한 규칙을 가지지 않는 문제

일 경우에 사용해야 하기 때문이다.

규칙이 없으므로 모든 경우의 수를 대입해봐야 하는 수 밖에 없고.

점화식 또한 존재하지 않기 때문에 더더욱 일일이 계산해봐야 하는 것이다.

백준 온라인 저지 _ 브루트 포스 _ 블랙잭

위의 문제를 보면 어떤 규칙을 찾을 수가 없는 문제임을 알 수 있다.

물론 21(M) 을 넘지 않아야 한다는 전제가 있지만,

이를 포함해서 3개의 수를 찾는 데에는 전혀 규칙이 없다.

a b c 세 개가 정답인 경우도 있을 테고, c d e 세개가 정답일 수도 있는데

둘 사이의 연관성을 찾을 수 없기 때문에 브루트 포스 알고리즘을 사용하게 되는 것이다.

이런 브루트 포스 문제들 같은 경우에는 문제에서 제시되는 n의 수가 작은 경우가 많다.

위의 문제도 시간복잡도를 따지고 보면 O(n3) 을 나타내는데 이는 n이 10000만 된다고 하더라도

1000억번 이라는 횟수를 시행해야 하기 때문에 굉장한 시간을 잡아먹는다.

때문에 위의 문제는 n이 100 이하 라는 조건을 걸어둔 걸 알 수 있다.

정리해보자면 n이 작으면서, 규칙을 찾을 수 없다면 브루트 포스를 사용하는게 바람직하다.

구현하는 데는 어렵지 않은, 어떻게 보면 기본에 가까운 알고리즘 이므로

단순하게 이런 알고리즘도 있구나. 하는 정도만 알아도 충분하다.