
순환 (재귀) Recursion

만든이 서정현
작성일 2017.09.23

00. 과제

- 리스트의 핵심 성질이 왜 **중복 허용**인가?

리스트 (컴퓨팅)

위키백과, 우리 모두의 백과사전.

리스트(list)는 **컴퓨터 과학**에서 같은 값이 한 번 이상 존재할 수 있는 일련의 값이 모여있는 **추상적 자료형**이다. **시퀀스**(sequence)라고도 부른다.

00. 과제

- 중복 허용하지 않는 자료형
-

1. Set : 집합

- 순서가 없다.
- 중복 데이터를 허용하지 않는다.

2. Map

- key: 중복 허용 X
- value: 중복 허용 가능
- Key는 데이터의 성격을 설명해주는 값

01. 순환 함수? 자기 자신을 호출하는 함수

- 무한 루프에 빠지지 않기 위해 일정한 탈출 조건이 있어야 한다.
-

장점

- 이해가 쉽고 코드의 가독성이 좋다.
- 코드를 단순화할 수 있다.
- 알고리즘의 정의가 순환적으로 되어 있는 경우에 유리

단점

- 비슷한 양의 연산을 할 경우, 실행 시간이 오래 걸린다.
- 디버깅 및 실행 흐름을 파악하기 힘들다.

01 순환 함수?

- 순환과 반복의 차이

순환

- 코드를 이해하고 유지하는 것이 더 중요
- 메소드가 자신을 호출 할 때에 메모리를 위한 비용과 CPU 사용 시간이 더 많이 소요
- 자기 자신을 호출하는 형태라 스택을 많이 사용
→ 스택 오버플로우 발생
- 함수 호출 횟수가 늘어남에 따라 오버헤드가 많이 커진다.
 - 호출 횟수는 스택의 남은 공간과 재귀 함수의 지역 변수 사이즈에 따라 달라진다.

→ 해결 방법: 꼬리 재귀

반복

- 코드가 복잡해진다.
- 반복적인 메소드 호출을 위한 메모리는 한번만 필요하기 때문에 성능적인 면에서 유리

메소드의 호출은
매개변수 리스트를 보관할 메모리 영역과
메소드를 실행할 수 있는 복사 공간이 필요

02. 꼬리 재귀?

- 재귀 호출이 끝난 후, 현재 함수에서 추가 연산을 요구하지 않도록 구현하는 재귀의 형태

함수 호출이 반복되어 스택이 깊어지는 문제를 컴파일러가 선형으로 처리하도록 알고리즘을 바꿔 스택을 재사용
-> 더 이상 값이 변할 여지가 없으므로 스택을 덮어씀

** 사용하기 위해 컴파일러의 최적화 기능 지원 여부 확인해야 함

```
int FactorialTail (int n, int acc)
{
    if (n == 1)
        return acc;

    return FactorialTail(n - 1, acc * n);
}
```

컴파일러 해석

<일반 재귀함수>

호출 : Factorial(3)

3 * Factorial(2)

3 * (2 * Factorial(1))

3 * (2 * 1)

3 * 2

6



```
int Factorial(int n)
{
    if (n == 1) return 1;

    int result = Factorial(n - 1);
    return n * result;
}
```

<꼬리 재귀함수>

호출 : FactorialTail(3, 1)

FactorialTail(2, 3)

FactorialTail(1, 6)

6

6

6



```
int FactorialTail(int n)
{
    int acc = 1;

    do
    {
        if (n == 1) return;
        acc = acc * n;
        n = n - 1;

    } while (true);
}
```

acc 변수에 계산한 값을 저장 후,
값을 한번에 반환

03. 재귀 예시

- 팩토리얼

시간 복잡도: $O(n)$

```
public int factorial(int num)
{
    if(num <= 1)
        return 1;
    else
        return num * factorial(num-1);
}
```

시간 복잡도: $O(n)$

```
public int factorial_iter (int num)
{
    int i, result = 1;

    for (i=num; i>0; i--)
        result = result * i;

    return result;
}
```


03. 재귀 예시

- 피보나치

시간 복잡도: $O(2^n)$

```
public int Fibonacci(int num)
{
    if(num <= 1)
        return 1;

    return Fibonacci(num-1) + Fibonacci(num-2);
}
```

```
public int Fibonacci_iter(int num)
{
    int a = 1, b = 1, sum = 0;

    if( num <= 2)
        return 1;

    for (int i = 0; i <= num ; i++)
    {
        sum = a +b;
        a = b;
        b = sum;
    }

    return sum;
}
```

시간 복잡도: $O(n)$

03. 재귀 예시

- 하노이의 탑

시간 복잡도: $O(2^n)$

```
void HanoiTowerMove(int num, char from, char by, char to){  
    if (num == 1)  
        System.out.println(from + "-->" + to);  
  
    else{  
        HanoiTowerMove(num - 1, from, to, by);  
        System.out.println(from + "-->" + to);  
        HanoiTowerMove(num - 1, by, from, to);  
    }  
}
```

03. 재귀 예시

- 순환이 더 좋은 예: 거듭제곱 계산

시간 복잡도: $O(\lg(n))$

```
double power(double x, int n)
{
    if (n == 0)
        return 1;

    else if (n % 2 == 0)
        return power(x * x, n/2);
}
```

비슷한 예제로,
탐색: 이진 탐색, DFS (depth first search)
정렬: 퀵 정렬, 병합 정렬

시간 복잡도: $O(n)$

```
double power_iter(double x, int n)
{
    int i;
    double result = 1.0;

    for (i=0; i>n; i++)
        result *= x;

    return result;
}
```

04. 참고 사이트

재귀, 반복, Tail Call

<http://homoefficio.github.io/2015/07/27/%EC%9E%AC%EA%B7%80-%EB%B0%98%EB%B3%B5-Tail-Recursion/>

꼬리 재귀

<http://bozeury.tistory.com/entry/%EA%BC%AC%EB%A6%AC-%EC%9E%AC%EA%B7%80-%EC%B5%9C%EC%A0%81%ED%99%94Tail-Recursion>