Algorithm
Recursive

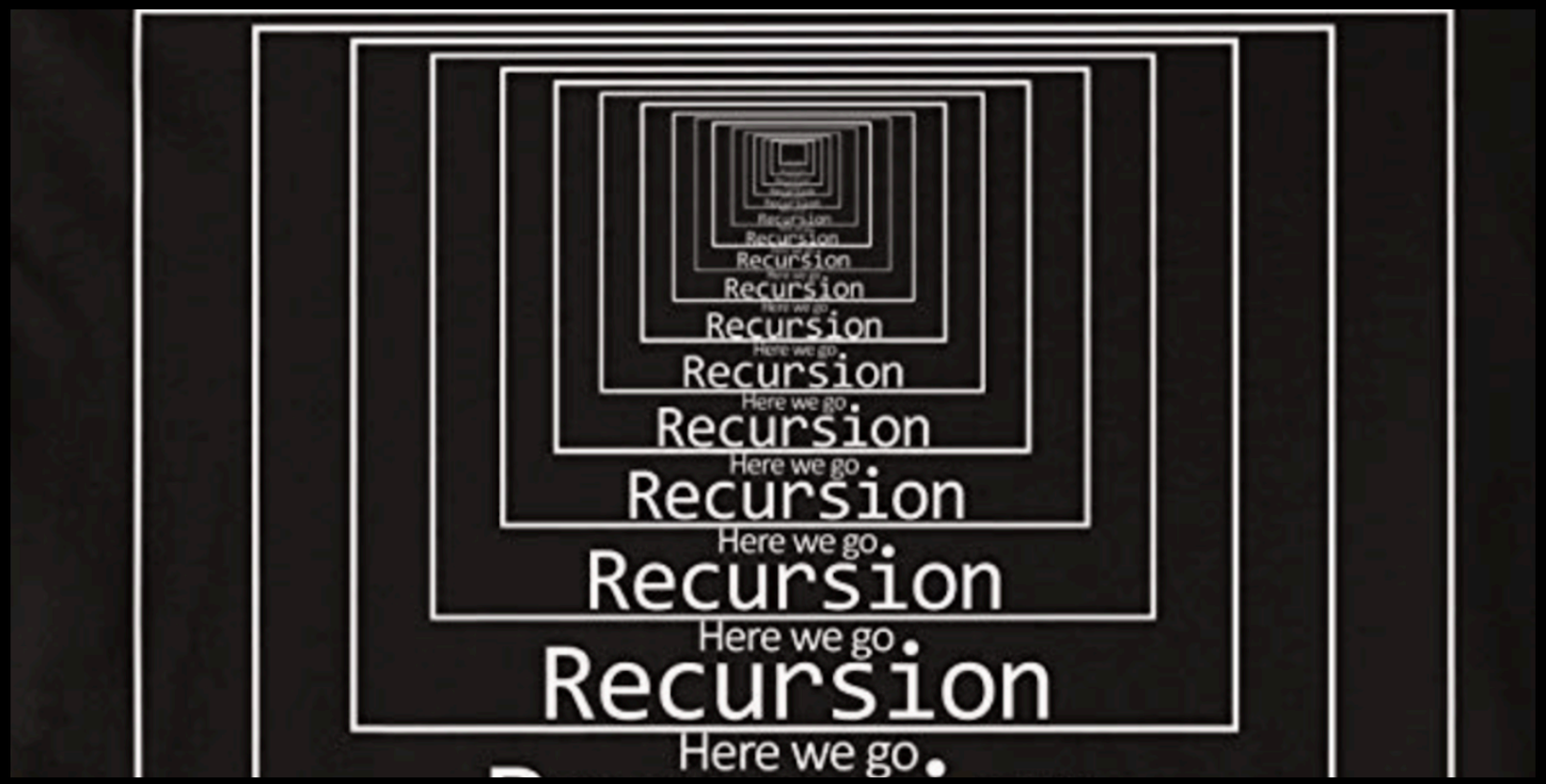
Bill Kim(김정훈) | ibillkim@gmail.com

목차

Recursive

Recursive는 우리말로 풀이하면 '재귀' 라는 단어로서 컴퓨터 과학(CS)에서의 재귀는 자신을 정의할 때 자기 자신을 재참고 하는 방법을 뜻합니다.

이를 프로그래밍에서 적용하면 재귀 호출(Recursive call)의 형태로 많이 사용합니다.



Recursive Function

Recursive Function이란 Recursive를 수행하는 함수로서 하나의 함수를 재귀적으로 호출하는 전형적인 재귀 호출 방식입니다.

보통 루프문(For, While 등)과 변수 등을 사용하여 함수를 재귀 호출합니다.

시스템 내부적으로는 재귀 호출을 할 때 마다 메모리 영역에 스택을 만들고 해당 스택 안에 관련 컨텍스트등을 담아서 참조 및 관리합니다.

Recursive Function

```swift
func recursive() {
    recursive()
}

func recursiveSum(_ n: Int) -> Int {
    guard n > 0 else { return 0 }
    return n + recursiveSum(n - 1)
}

print(recursiveSum(5)) // 15
```

Recursive Function

방금 예시로 제시한 Recursive Function 의 동작 흐름을 살펴보면 아래와 같습니다.

```
recursiveSum(5) =
-> 5 + recursiveSum(4)
-> 5 + 4 + recursiveSum(3)
-> 5 + 4 + 3 + recursiveSum(2)
-> 5 + 4 + 3 + 2 + recursiveSum(1)
-> 5 + 4 + 3 + 2 + 1 + recursiveSum(0)
-> 5 + 4 + 3 + 3
-> 5 + 4 + 6
-> 5 + 10
-> 15
```

다음처럼 재귀를 하면서 **시스템 내부적으로 메모리 영역의 끝단에 별도의 스택을 만들고 거기에 인자값 및 함수 지역 변수를 복사**합니다. 그리고 함수의 실행이 끝나면 스택 영역을 파괴하고 리소스를 회수합니다.

따라서 해당 방색의 재귀 함수는 재귀의 수가 많아질 경우 시스템은 스택 영역을 계속 추가적으로 사용하게 되면 일정 **메모리 한계 영역을 넘어서게 되면 바로 스택 오버플로우가 발생**할 수 있는 큰 단점이 있습니다.

Tail Recursion

위에서 언급했던 스택 오버플로우를 피하기 위하여 바로 꼬리 재귀(Tail Recursion)라는 방식을 사용합니다.

꼬리 재귀(Tail Recursion)는 함수 자신의 결과를 바로 리턴하여 추가적인 연산을 하지 않지 않고 이전 재귀 함수를 종료하는 방식입니다.

추가적인 연산이 없다는 것은 재귀 결과를 받는 시점에는 이전 함수 내의 컨텍스트(인자값 및 지역 변수 등)를 더이상 참조하지 않는다는 의미로 해석할 수 있습니다.

재귀가 호출되어을 때 새로 생성해야 할 컨텍스트는 사실상 현재의 컨텍스트와 동일하기에 별도의 추가적인 스택 생성과 파괴가 필요 없게 됨으로 메모리 및 성능 낭비를 막을 수 있습니다.

Tail Recursion

앞서 살펴본 예시를 꼬리 재귀 방식으로 바꾸면 아래와 같습니다.

```
func recursiveTailSum(_ n: Int, _ acc: Int) -> Int {
    guard n > 0 else { return acc }
    return recursiveTailSum(n - 1, acc + n)
}
```

Tail Recursion

```
print(recursiveSum(500000000)) // 스택 오버플로우(Stack Overflow) 등으로 런타임 오류 발생
print(recursiveTailSum(500000000, 0)) // 125000000250000000
```

꼬리 재귀를 사용하면 위의 예시처럼 5억 단위의 숫자 증가 연산도 별다른 문제없이 수행이 가능합니다.

꼬리 재귀가 아닌 방식을 사용하면 아래와 같은 런타임 오류가 발생합니다.

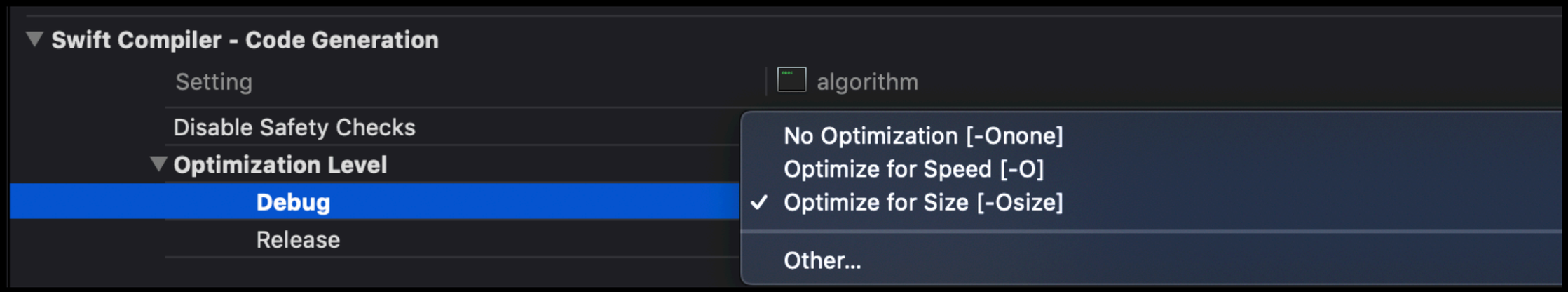
```
func recursiveSum(_ n: Int) -> Int {
    guard n > 0 else { return 0 }

    return n + recursiveSum(n - 1)        ≡ Thread 1: EXC_BAD_ACCESS (code=2, address=0x7ffeef3ffff8)
}
```

Tail Recursion

앞단 위의 꼬리 재귀가 정상적으로 시스템에서 최적화되어 동작하려면 컴파일 관련 최적화 옵션을 설정할 필요가 있습니다.

Xcode 내에서 Swift를 예시로 설명하면 Build Setting에서 아래와 같은 설정이 필요합니다.



Features

지금까지 살펴본 재귀에 대해서 특징을 살펴보면 아래와 같습니다.

1. 반복문 대신에 재귀 함수 호출로 코드를 간결하게 표현 가능
2. 반복되는 알고리즘 연산 등을 간단히 코드 상으로 구현 가능
3. 간결한 코드로 상당히 깊은 문제 해결을 도와줌
4. 일반 재귀 함수의 경우 콜스택 생성으로 인한 많은 메모리 사용
5. 흐름을 추적하기 어렵다
6. 흐름 추적이 어려움으로 인하여 디버깅 시 어려움
7. 재귀 호출로 인한 실행 속도의 저하

Implementation

Swift를 활용하여 다양한 알고리즘에서의 재귀 함수를 살펴보겠습니다. 본 강의에서 살펴볼 재귀 함수의 예시들은 다음과 같습니다.

알고리즘

1. Factorial(팩토리얼)
2. Fibonacci(피보나치)
3. Power(거듭제곱)
4. Greatest Common Divisor(최대공약수)

Implementation

Factorial(팩토리얼)

```
public func factorial(n: Int) -> Int {
    if n == 0 {
        return 1
    }

    return n * factorial(n: n-1)
}

print("재귀 호출에서 5! 의 값은 \(factorial(n: 5))") // 재귀 호출에서 5! 의 값은 120
```

Implementation

Fibonacci(피보나치)

```
public func fibonacci(n: Int) -> Int {
    if n == 0 || n == 1 {
        return n
    } else {
        return fibonacci(n: n-1) + fibonacci(n: n-2)
    }
}

// 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...
print("재귀 호출에서 피보나치 수열의 3번째 값은 \(fibonacci(n: 3))") // 재귀 호출에서 피보나치 수열의 3번째 값은 2
```

Implementation

Power(거듭제곱)

```
public func power(x:Double, n:Int) -> Double {
    if n == 0 {
        return 1
    }
    else if n > 0
    {
        if n % 2 == 0
        {
            return power(x: x, n: n / 2) * power(x: x, n: n / 2)
        } else {
            return x * power(x: x, n: n-1)
        }
    } else {
        return 1 / power(x: x, n: -n)
    }
}

print("재귀 호출에서 2 의 3 제곱 의 값은 \(power(x: 2, n: 3))") // 재귀 호출에서 2 의 3 제곱
의 값은 8.0
```

Implementation

Greatest Common Divisor(최대공약수)

```
public func gcd(a:Int, b:Int) -> Int {
    if a == b { return a }
    else if a > b { //
        if a % b == 0 { return b }
        else { return gcd(a: b, b: a % b) }
    } else{
        if b % a == 0{ return a }
        else { return gcd(a: a, b: b % a) }
    }
}

// 12의 약수 : 1, 2, 3, 4, 6, 12
// 18의 약수 : 1, 2, 3, 6, 9, 18

print("재귀 호출에서 12와 18의 최대공약수 값은 \(gcd(a: 12, b: 18))") // 재귀 호출에서 12와 18
의 최대공약수 값은 6
```

References

[1] [스위프트 : 알고리즘] 재귀호출 (1 / 6) : recursive: 재귀호출 : https://the-brain-of-sic2.tistory.com/29

[2] Building Recursive Algorithms in Swift : https://medium.com/swift-algorithms-data-structures/building-recursive-algorithms-in-swift-7f242b96a4ad

[3] Swift로 꼬리 재귀 사용하기 : https://academy.realm.io/kr/posts/swift-tail-recursion/

[4] [Swift][Algorithm]꼬리 재귀 : http://minsone.github.io/programming/tail-recursion-in-swift

[5] Working With Recursive Algorithms In Swift : https://learnappmaking.com/swift-recursion-how-to/

References

[6] Swift Recursion : https://www.programiz.com/swift-programming/recursion

[7] Tail 과 꼬리재귀(Tail Recursion) - Swift : https://sooooprmx.com/archives/5699

[8] Chapter 8: Recursion Swift Programming from Scratch : https://www.weheartswift.com/recursion/

[9] Recursion (재귀) : https://dev-jiwon.github.io/swift-grammar-8/

[10] [재귀] 재귀 vs 꼬리 재귀 : https://ledgku.tistory.com/37

References

[11] This week I learned - 재귀편 : https://velog.io/@wondernova/This-week-I-learned-재귀편-

Thank you!