디지털 교재

파이썬으로 시작하는

자료구조와 알고리즘

<3주차>

NAVER CONNECT

3 - 필수 알고리즘

재귀 (Recursion)

재귀란 함수가 자기 자신을 호출하는 프로그래밍 기법입니다. 문제를 작은 부분으로 나누어 해결하는 데 사용됩니다. 정의는 매우 단순하지만, 추후 그래프 탐색, 트리 , dp 등 주요 자료구조와 알고리즘과 연결되기 때문에 매우 중요한 개념입니다.

재귀에 대한 대표적인 예시는 **factorial**과 **fibonacci**가 있습니다. 자기 자신을 재참조한다는 것이 어떤 의미인지 코드를 통해 확인해 보도록 합시다.

재귀의 example

팩토리얼(factorial)

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

함수 `factorial` 의 반환에 `factorial` 를 재참조하는 것을 확인할 수 있습니다.

피보나치(fibonacci)

```
def fibo(n):
    if n == 1 or n == 2:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

함수 `fibo` 역시 반환에 `fibo` 를 재참조하는 것을 확인할 수 있습니다.

함수를 선언해 보았으니 각각의 함수가 어떻게 실행되는지 확인해 보도록 합시다.

팩토리얼 함수의 실행

https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/cdc046b4-3481-465b-aef6-8a5e2c3684ad/python_factorial.m4v

피보나치 **함수의 실행**

https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/39eef2e1-b021-4c4d-9d07-00b1ead7561e/python_fibo.m4v

재귀의 수학적 접근 - 점화식

앞선 `fibonacci`와 `factorial`을 자세히 살펴보면, `n`번째 함수를 `n-1`번째 함수로 표현하는 것을 알 수 있습니다. `factorial`의 경우, $f(n) = n * f(n-1)$로 표현되고, `fibonacci`의 경우, $f(n) = f(n-1) + f(n-2)$으로 표현됩니다.

이러한 식을 우리는 **Recurrence Relation(점화식)**이라고 합니다. 이것 역시 자기 자신을 참조하는 개념이 들어 있습니다. 그러나 계속 자기 자신을 참조하게 되면, 무한 루프에 빠질 수 있습니다.

무한 루프를 방지하기 위해 **Base case**를 설정해 주어야 합니다. **Base case**는 더 이상 자기 자신을 재참조하지 않는 상황을 의미합니다.

앞선 `factorial`와 `fibonacci` 의 상황으로 위 내용을 정리합시다.

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

- **Recurrence Relation (점화식)**

  ⇒ $f(n) = n * f(n-1)$

- **Base case**

  ⇒ $f(1) = 1$

```
def fibo(n):
    if n == 1 or n == 2:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

- **Recurrence Relation (점화식)**

  ⇒ $f(n) = f(n-1) + f(n-2)$

- **Base case**

  ⇒ $f(1) = 1,\ f(2) = 1$

재귀는 **Recurrence Relation**과 **Base case**로 구성되는 것입니다. **Recurrence Relation**은 해답을 직관적으로 제공해 주며, **Base case**는 무한 루프를 방지해줍니다. 재귀를 올바르게 사용하기 위해서는 두 가지를 모두 놓치지 않도록 해야 합니다.

1. 식들의 관계를 생각해본다. (**Recurrence Relation**)
2. $n = 0, n = 1, \cdots$ 인 경우를 생각해본다. (**Base case**)

재귀의 시간 복잡도

시간 복잡도는 항상 고려해야 하는 중요한 문제입니다.

> 💡 재귀의 시간복잡도 = 재귀 함수 호출 수 $\times$ (재귀 함수 하나당) 시간복잡도

앞서서 다룬 `factorial`과 `fibo` 의 시간 복잡도를 위 공식을 활용해서 구해보도록 하겠습니다.

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

- 재귀 함수 호출 수

  ⇒ n
- 재귀 함수 하나 당 시간복잡도

  ⇒ $O(1)$
- 재귀의 시간 복잡도

  ⇒ $O(n \times 1)$

```
def fibo(n):
    if n == 1 or n == 2:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

- 재귀 함수 호출 수

  ⇒ 2^n
- 재귀 함수 하나 당 시간복잡도

  ⇒ $O(1)$
- 재귀의 시간 복잡도

  ⇒ $O(2^n \times 1)$

Python의 함수에 대한 특수 문법

재귀를 실전적으로 활용하기 위해서는 사용 언어에서 함수에 대한 정의와 활용을 어떻게 하는지를 정확히 파악하고 있어야 합니다. 따라서 이번 파트에서는 python에서 함수에 대해 제공하는 특수 문법에 대해 소개하겠습니다.

중첩 함수 (Nested Function)

중첩 함수(Nested Function)는 단순하게 어떤 함수 안에 정의된 함수 객체를 말합니다. 다음 예시의 내부 함수가 중첩된 함수입니다.

```
def say(text):       # 외부 함수
    print("Hello!")
    def hello():       # 내부 함수
```

```
        print("Nice to meet you!")
    hello()

say()
```

▼ 결과

```
Hello!
Nice to meet you!
```

중첩 함수를 사용하는 가장 큰 이유는 2가지로 나뉩니다.

1. 간결한 코드 작성
2. 클로저[1]

중첩된 함수를 사용하면 외부 함수에서 만든 객체에 내부 함수가 접근할 수 있도록 만들어 코드를 간결하게 만들 수 있습니다. 만약 어떤 함수 내부에서 다른 함수에서 쓰인 변수를 사용하고 싶은 경우, 다른 언어라면 전역변수[2]로 설정해서 처리해야 합니다. 다음 예시로 알아보겠습니다.

```
def sum_of_n_list():
    n_list = [5, 2, 7, 3, 9]
    def get_sum():
        result = 0
        for num in n_list:
                    print(f"num : {num}")
            result += num
        return result
    return get_sum()

print(sum_of_n_list())
```

▼ 결과

```
num : 5
num : 2
num : 7
num : 3
num : 9
26
```

함수 `get_sum()` 에서 함수 `sum_of_n_list()` 의 리스트 `n_list` 를 접근 가능하여 외부에서 따로 함수 `get_sum()` 을 선언하지 않고도 코드를 쓸 수 있습니다. 클로저는 코딩 테스트에서 실질적으로 많이 사용하지 않기 때문에 넘어가도록 하겠습니다. 주석에 간단한 설명을 넣어놨으니 확인해 보시기 바랍니다.

주의 사항 : 외부 함수에서 참조하는 변수는 변경할 수 없다.

가장 잘 지켜야 하는 주의 사항입니다. 외부 변수에 접근할 수는 있지만, 그것을 직접적으로 수정할 수는 없습니다. 다음 예시 코드가 외부 변수에 대한 변경을 시도하는 상황입니다.

```
def compute():
    a = 5
    b = 9
    x = 3
    def linear_comb():
        a = a * x + b      # 예외 상황 발생!! (외부 변수 a 변경 시도)
        return result
    return linear_comb()
```

하지만 실제로 내부적으로 값을 변경해야 할 상황이 발생할 수도 있겠죠? 그때는 `nonlocal` 명령어로 이런 예외 상황을 피할 수 있습니다. `nonlocal` 명령어는 외부 변수에 대한 수정을 가능하게 해줍니다. 따라서 위의 코드를 변경하면 다음과 같습니다.

```
def compute():
    a = 5
    b = 9
    def linear_comb(x):
        nonlocal a
        a = a * x + b      # nonlocal 명령어로
        return a
    return linear_comb

f = compute()
print(f(5))
```

▼ 결과

```
34
```

함수를 통한 자유로운 데이터 전달과 반환

Python의 편리한 기능 중 하나가 반환 개수에 대한 제한이 없다는 것입니다. 소제목의 '자유롭다'라는 표현을 사용한 것은 다른 언어에 비해 데이터 이동에 대한 제약이 적다는 것을 전하고 싶었기 때문입니다. 다음 예시를 한 번 보겠습니다.

```
def return_lots_of_result():
    result1 = 0
    result2 = False
    result3 = []
    return result1, result2, result3

print(return_lots_of_result())
```

▼ 결과

```
(0, False, [])
```

Python의 함수 사용 팁

함수 인자로 Mutable 객체 넘기기 (얕은 복사 vs 깊은 복사)

Mutable 객체는 앞서 설명해 드린 바와 같이 변경 가능성 있는 객체를 의미합니다. list, dictionary, set이 이런 객체에 해당합니다. 코딩테스트에서 재귀 문제를 마주치게 되면, 이 객체들을 인자로 전달하는 경우가 많을 것입니다. 이때, **얕은 복사(shallow copy)**가 되어 원본 데이터까지 수정된다는 점을 반드시 기억하고 사용하셔야 합니다. 그래서 아래의 코드처럼 mutable 객체를 인자로 전달해서 재귀 함수를 호출한 뒤에 원본 객체를 출력해 보면 내부 데이터가 달라져 있음을 알 수 있습니다.

```
li = [i for i in range(1, 11)]

def only_odd(li):
    even_flag = 0
    for val in li:
        if val % 2 == 0:
            even_flag = 1
            li.remove(val)
            break
    if even_flag:
        only_odd(li)
    else:
        return

print("재귀 이전 원본 데이터 :", li)
```

```
only_odd(li)
print("재귀 이후 원본 데이터 :", li)
```

▼ 결과

```
재귀 이전 원본 데이터 : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
재귀 이후 원본 데이터 : [1, 3, 5, 7, 9]
```

만약 원본 데이터를 유지한 상태로 재귀 함수를 사용하고 싶다면, **깊은 복사(deep copy)**를 사용하시면 됩니다. 얕은 복사는 원본 데이터와 다른 객체를 만들어 내부 데이터를 공유하지 않게 됩니다.

```
li = [i for i in range(1, 11)]
new_li = li[:]       # 변경점! - 얕은 복사
def only_odd(li):
    even_flag = 0
    for val in li:
        if val % 2 == 0:
            even_flag = 1
            li.remove(val)
            break
    if even_flag:
        only_odd(li)
    else:
        return

print("재귀 이전 원본 데이터 :", li)
only_odd(new_li)     # 변경점! - 얕은 복사
print("재귀 이후 원본 데이터 :", li)
```

▼ 결과

```
재귀 이전 원본 데이터 : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
재귀 이후 원본 데이터 : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

재귀 최대 깊이 조정하기

파이썬으로 코딩 테스트 문제를 풀다 보면, 재귀 최대 깊이를 초과하여 에러가 발생하기도 합니다. (대표적으로 백준 문제) 이러한 경우, 아래의 코드를 이용해서 재귀 최대 깊이를 임의로 설정해서 해결할 수 있습니다.[3] 파이썬에서 기본 제공하는 최대 재귀 깊이는 1000이기 때문에 그 이상의 값으로 원하는 값을 입력하면 재귀 최대 깊이를 바꿀 수 있습니다.

```
import sys

sys.setrecursionlimit(10000)
```

코딩테스트 활용

재귀 자체의 개념은 간단하지만, 매우 다양한 분야에서 사용됩니다. 대표적으로 **완전 탐색, 동적 계획법(DP)의 top-down 방식, DFS** 등이 있습니다. 반복문으로도 표현할 수 있지만, 여러 다른 길이의 데이터에 대해 접근해야 할 때 재귀가 자주 사용됩니다.

점화식으로 표현가능한 문제

재귀를 사용하는 문제는 점화식으로 표현 가능하다고 윗부분에서 설명해 드렸습니다. 그러면 왜 이러한 문제 유형을 재귀로 푸는 것이 좋은지 반복문과 재귀로 풀었을 때의 차이를 보여드리면서 설명하겠습니다.

반복문

```
def fibo(n):
    if n == 1 or n == 2:
        return 1
    else:
        n_list = [0 for i in range(n)]
        n_list[0] = 1
        n_list[1] = 1
        for i in range(2, n):
            n_list[i] = n_list[i-1] + n_list[i-2]
        return n_list[-1]
```

재귀

```
def fibo(n):
  if n == 1 or n == 2:
    return 1
  return fibo(n - 1) + fibo(n - 2)
```

- **Recurrence Relation (점화식)**

  ⇒ $f(n) = f(n-1) + f(n-2)$

- **Base case**

  ⇒ $f(1) = 1,\ f(2) = 1$

두 방식을 비교해 보면, 우선 코드 길이부터 매우 차이가 크게 납니다. 그리고 점화식 형태를 그대로 가져와서 반환 값을 구성할 수 있기 때문에 만약 점화식을 세울 수 있는 문제라면, 재귀를 이용해서 더 직관적으로 문제를 해결할 수 있습니다.

완전탐색

💡 "완전탐색"은 문제의 해를 찾기 위하여 가능성이 있는 모든 해를 찾아내어 그 중 주어진 조건을 만족하는 최적해를 찾는 패러다임 입니다.

완전탐색은 알고리즘 패러다임중 하나로 가능한 모든 해를 찾아내어 그 중 조건을 만족하는 최적해를 찾아내는 방법입니다. 완전탐색은 모든 가능성을 확인하기때문에 가장 정확하고 최적의 해를 구할 수 있지만, 시간복잡도가 매우 높아질 수 있습니다.

모든 문제에 대해 완전탐색을 적용시키면 항상 답은 구해지겠지만 코딩테스트가 제한하는 시간안에 문제를 해결하지 못할 가능성이 큽니다. 그렇기에 완전탐색을 사용해야 할 시점은 다음과 같습니다.

1. 주어진 입력의 범위가 작아 가능한 모든 해를 찾는 시간이 적게 들 때

   코딩테스트의 문제는 대체로 시간복잡도가 10^8이내인 경우 통과가 됩니다. 그렇기에 입력의 범위가 100과 같이 작은 경우 완전 탐색으로 걸리는 시간복잡도가 O(n^2)이라면 해당 방법은 10^4의 시간복잡도를 갖기에 통과가 될 것입니다. 이와 같이 주어진 입력의 범위가 작아 완전 탐색의 시간복잡도 또한 낮아지는 경우 완전 탐색으로 빠르게 구현하여 문제를 해결할 수 있습니다.

   즉, 시간복잡도를 계산하고 될거같다 싶으면 일일이 다 하면 됩니다. 반복문이 몇개든, 비효율적인 풀이같아 보여도 상관 없이 풀이를 작성해나가면 됩니다.

2. 우선적으로 답을 구하고 그 과정을 최적화하여 시간을 줄이고 싶을 때

   주어진 입력의 범위가 커서 완전 탐색으로는 시간 제한을 맞추지 못하는 문제에도 우선 완전탐색을 적용시키는 방법을 고려할 수 있습니다. 완전 탐색은 앞서 설명했듯이 문제를 푸는 가장 기초적인 방법입니다. 그렇기에 완전 탐색을 우선 구현한 후 해당 방법을 개선해 나감으로써 시간 제한을 맞출 수 있습니다. 또한 완전탐색을 통해 진행되는 문제의 과정을 보고 문제의 유형을 파악할 수 있는 등 문제의 명확한 풀이방법을 찾지 못했을 경우에는 우선 완전탐색으로 구현하는 것도 좋은 방법입니다.

   - 정렬
   - 메모이제이션
   - 투포인터
   - DP
   - 백트래킹
   - 이진탐색

❓ **완전 탐색 vs 브루트 포스 brute force**

위 완전탐색과 브루트포스는 종종 혼용되어 사용하는데 차이는 존재합니다. 두 패러다임 모두 가능한 모든 해를 찾아내는 것은 같지만, 완전탐색은 그 모든 해를 찾아나가는 과정이 보다 체계적인 방면 브루트포스는 모든 해를 찾아나가는 과정이 이름 그대로 무식(Brute)하게 모든 해를 찾아나간다는 것이 패러다임의 차이입니다.
하지만 강의에서 두 용어를 엄밀하게 구분하진 않을 겁니다.

코딩테스트에서 자주 나오는 완전탐색 문제는 다음과 같습니다

- **모든 가능성 탐색:** 모든 가능성을 탐색해보고 문제에서 원하는 결과와 일치하는 값을 찾는 문제
- **부분집합 생성:** 모든 부분집합을 생성하여 특정 조건을 만족하는 부분집합을 찾는 문제.
- **순열과 조합:** 주어진 요소들로 가능한 모든 순열 또는 조합을 생성하는 문제.
- **격자 탐색:** 격자 내의 모든 위치를 탐색하며 특정 조건을 만족하는 위치를 찾는 문제.

구현

- 반복문

```
for i in range(n):
    for j in range(i+1, n):
```

- 재귀

```
a.append()
backtrakcing()
a.pop()
```

DP (Dynamic Programming)

💡 "다이나믹 프로그래밍(Dynamic Programming)"은 큰 문제를 작은 문제들로 나누어 해결한 후, 그 결과를 저장하여 중복 계산을 줄이는 최적화 기법을 의미합니다.

"다이나믹 프로그래밍(Dynamic Programming)"은 큰 문제를 작은 문제들로 나누어 해결한 후, 그 결과를 저장하여 중복 계산을 줄이는 최적화 기법을 의미합니다.

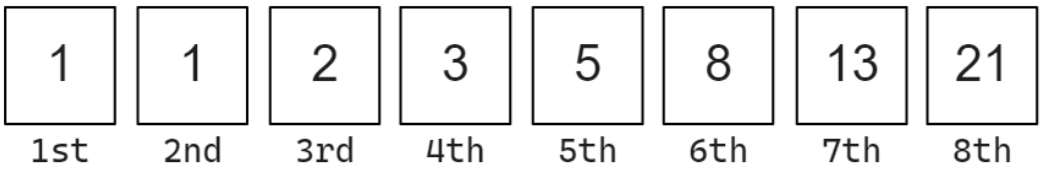
다이나믹 프로그래밍을 사용하는 알고리즘의 주요 특징은 다음과 같습니다:

1. **중복 계산 감소**: 이미 계산한 결과를 저장하고, 필요할 때마다 재활용하여 중복된 계산을 피합니다.
2. **점화식**: 주어진 문제의 해를 작은 문제의 해를 통해 표현하는 식을 정의합니다. 이를 점화식이라고 부릅니다.
3. **최적 부분 구조**: 큰 문제의 최적해가 작은 문제들의 최적해를 통해 구할 수 있어야 합니다.

모든 해결책을 다 탐색해보는 '완전 탐색'은 기본적으로 높은 시간 복잡도를 가집니다. 이를 '체계적'이고 '효율적으로 탐색하기 위해서는 몇 가지 조건이 있어야 합니다.

- Overlapping Subproblems - 중복 하위 문제
  - 중복 계산해야 하는 하위 문제가 존재합니다.
- Optimal Substructure - 최적 부분 구조
  - 하위 문제에서 구한 최적의 답이 큰 문제의 최적의 답을 구할 수 있는 구조여야 합니다.

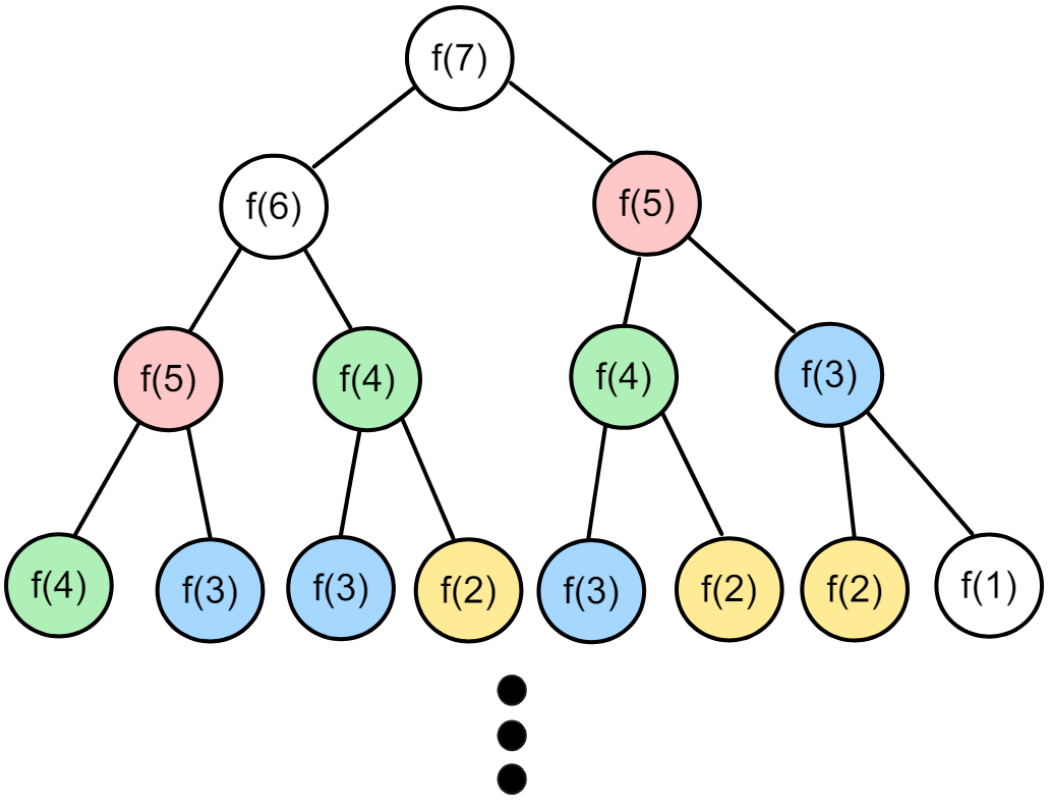
피보나치 수열을 통해 dp에 대한 개념을 가져가 보도록 하겠습니다.



피보나치 수열은 다음과 같이 $F(n) = F(n-1) + F(n-2) \quad F(1) = 1, F(2) = 1$인 수열을 말합니다. 저희는 앞선 재귀 파트에서 $F(n) = F(n-1) + F(n-2)$가 **recurrence relation(점화식)**임과 $F(1) = 1, F(2) = 1$이 **base case**임을 배웠습니다. 그리고 코드는 다음과 같았습니다.

```
int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

이 때 시간복잡도는 매 번 2개의 함수를 호출함으로 $O(2^n)$의 시간복잡도를 가집니다. 아무래도 완전 탐색이다보니, 꽤 높은 시간복잡도를 가집니다. 그러나 이 피보나치에는 중복된 하위 문제들이 존재합니다. `fibo(7)`를 보도록 합시다.



위와 같이 상당한 중복이 발생하는 것을 볼 수 있습니다. 만약 전에 계산했던 것을 기억한다면, 위와 같은 중복 문제를 피할 수 있을 것입니다. 예를 들어, `f(5)`를 전에 계산 했으면, 후에 같은 과정으로 `f(5)`를 계산할 필요 없는 것입니다. 이렇게 될 시, 시간 복잡도는 `O(n)`으로 획기적으로 줄게 됩니다.

```
Map<Integer, Integer> memo = new HashMap<>();

int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
        if (!memo.containsKey(n)) {
                memo.put(n, fibonacci(n - 1) + fibonacci(n - 2));
        }
    return memo.get(n);
}
```

Top-down 방식과 Bottom-up 방식

```
Map<Integer, Integer> memo = new HashMap<>();

int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
        if (!memo.containsKey(n)) {
                memo.put(n, fibonacci(n - 1) + fibonacci(n - 2));
        }
    return memo.get(n);
}
```

위와 같은 방식은 `f(n)` 부터 `f(1)` 로 접근하므로, **top-down 방식**이라고 합니다. 반대로 `f(1)` 부터 `f(n)` 까지 접근하는 방식을 **Bottom-up 방식**이라고 합니다. **Bottom-up 방식**의 코드는 아래와 같습니다.

```
Map<Integer, Integer> memo = new HashMap<>();

int fibonacci(int n) {
    memo.put(1, 1);
    memo.put(2, 1);
    for (int i = 3; i <= n; i++) {
        memo.put(i, memo.get(i - 1) + memo.get(i - 2));
    }
    return memo.get(n);
}
```

Top-down 방식은 재귀(recursion)로 구현되는 반면, Bottom-up 방식은 반복문(iteration)을 통해 구현됩니다. 또 Top-down 방식에서 `memo` 를 채워 나가는 것을 **memoization**이라 하며, Bottom-up 방식에서 `memo` 를 채워 나가는 것을 **tabulation**라고 합니다.

---

1. 클로저 (Closure)

클로저(closure)란 외부 함수에서 정의된 데이터를 외부 함수가 종료되었음에도 그 데이터를 사용할 수 있는 중첩된 함수(Nested Function) 객체를 의미합니다. 클로저가 되기 위한 조건으로 3가지가 필요합니다.

1. 특정 함수의 내부 함수일 것
2. 외부함수의 변수(**프리 변수**)를 반드시 참조할 것
3. 외부함수가 내부 함수를 반환할 것

💡 프리 변수 : 어떤 함수에서 사용하지만, 그 함수 내부에서 정의되지 않은 변수를 의미합니다.

이것을 모두 만족해야 클로저 구현을 할 수 있습니다. 예시 코드를 보면서 알아보겠습니다.

```
def greeting():    # 외부함수
    s = "Good "
    def hello(t):    # 내부함수
        greet = s + t
        return greet
    return hello

g = greeting()
print(g("morning"))
```

▼ 결과

```
Good morning
```

코드 설명

함수 `greeting`은 외부함수로, 호출되면 함수 `hello`를 변수 `g`에 반환합니다. 이때, 함수 `hello`는 외부 변수 `s`를 참조해서 새로운 문자열 `greet`을 반환하게 되며, 함수 `hello`는 클로저가 됩니다.

다시 말하면, 함수 `greeting`은 함수 `hello`를 호출하는 것으로 메모리에서 사라지지만, 함수 `hello`가 클로저로 남아서 (함수 `greeting`이 존재하지 않더라도) 변수 `s`에 접근할 수 있게 됩니다. 위의 코드에서 `g = greeting()`을 통해 변수 `g`에는 함수 `hello`가 존재하게 되고, 이 때 문자열 `morning`이 인자로 넘어가서 내부 변수 `greet`을 반환합니다.

2. 파이썬에서 전역 변수 설정하기

`global` 명령어를 통해 특정 변수를 전역 변수로 설정할 수 있습니다.

```
def foo():
    global a      # global a = 1 (X)
    a = 1
    print("In the function :", a)

foo()
a += 1
print("Out of the function :", a)
```

▼ 결과

```
In the function : 1
Out of the function : 2
```

보통 함수에서 선언된 변수를 외부에서도 접근 가능하도록 만드는 데에 사용합니다. 다만, `global` 을 지정함과 동시에 할당하는 것은 불가능하다는 점을 알고 계셔야 합니다.

3. 재귀 한도 구하기

지원하는 재귀 한도를 구하는 방법도 `sys` 패키지에서 지원합니다.

```
import sys

print(sys.getrecursionlimit())
```