

# DP(다이나믹 프로그래밍)

WEEK 3 백민희

# DP(다이나믹 프로그래밍)이란?

**“하나의 문제는 단 한 번만 풀게 하는 알고리즘”**

- 하나의 큰 문제를 여러 개의 작은 문제로 나누어서  
그 결과를 저장하여 다시 큰 문제를 해결할 때 사용  
하는 것**

코딩테스트에서 자주 출제되는 중요한 개념!  
문제를 많이 풀면서 감을 잡는 것이 중요하다.

# 기존 방법:분할 정복 기법

- 분할정복기법:문제를 나눌 수 없을 때까지 나누어서 각각을 풀면서 다시 합병하여 문제의 답을 얻는 알고리즘.
- 대표알고리즘:병합정렬, 퀵 정렬, 이진탐색, 슈트라센 알고리즘 등
- 동일한 문제를 다시 푼다는 단점 있음 (퀵, 병합 정렬 제외)
- 단순 분할 정복으로 풀면 매우 비효율적임 ex) 피보나치수열

# 피보나치 수열

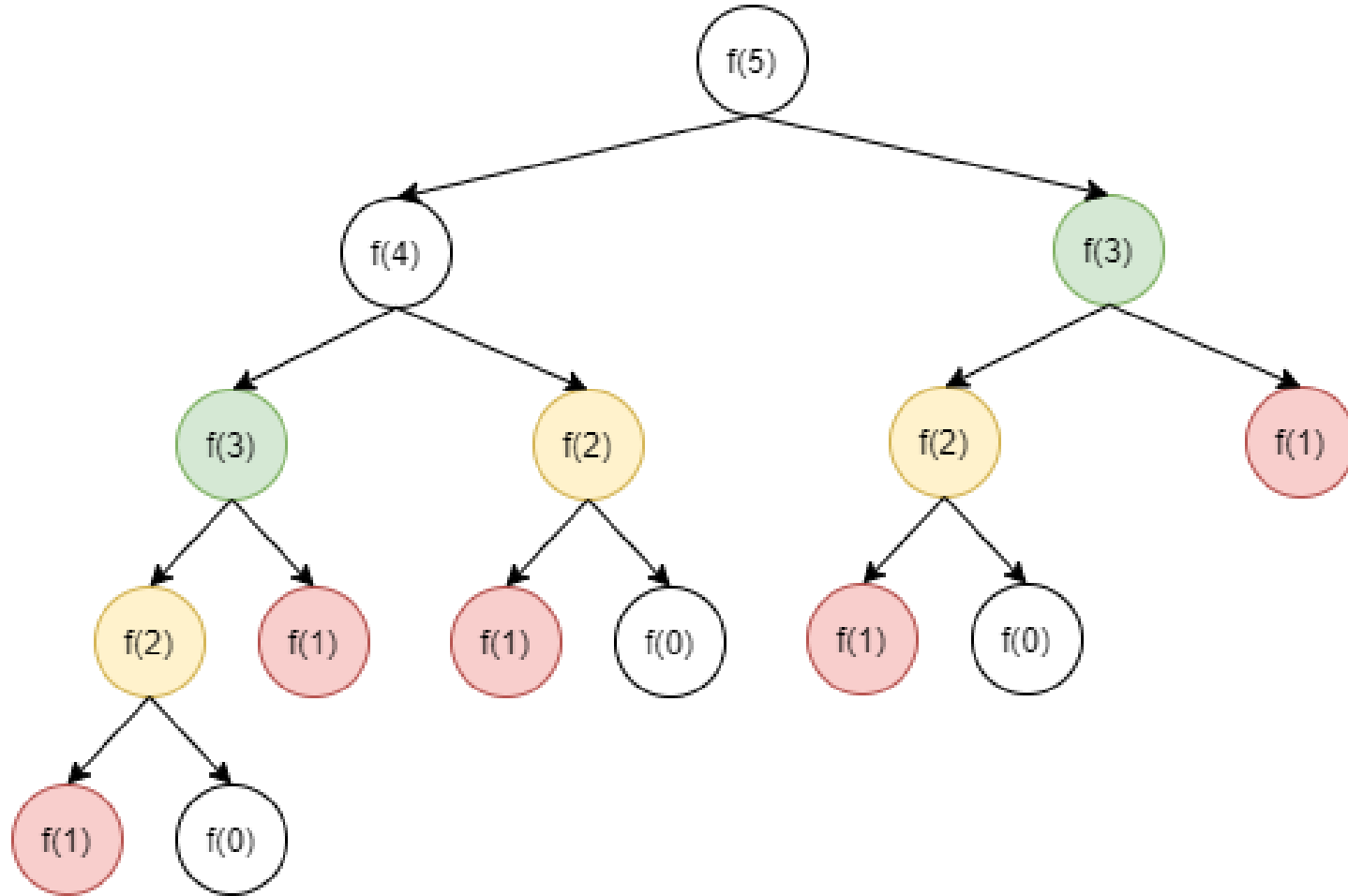
• 피보나치 수:  $F_n = F_{n-1} + F_{n-2}$

n번째 피보나치 수  $f(n)$  =  
n-1번째 피보나치 수 + n-2번째 피보나치 수의 합

1, 1, 2, 3, 5, 8, 13, 21, ...

# 피보나치 수열 함수 호출 트리 (재귀함수 이용)

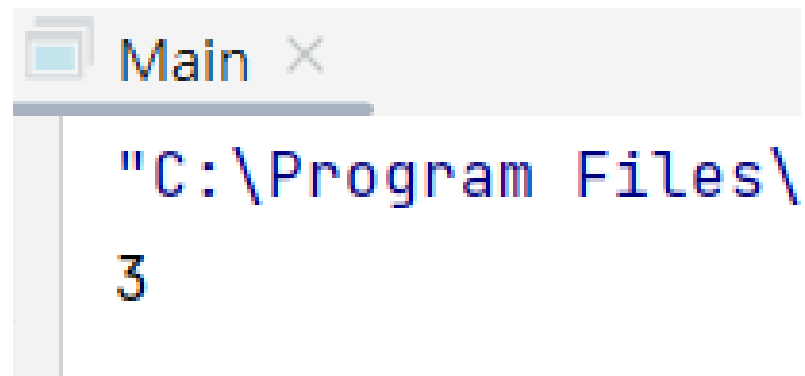
피보나치 수열 : 방정식 :  $f(n) = f(n-1) + f(n-2)$



ex.  $f(1)$ 이 5번이나 반복됨을 알 수 있다. => 굉장히 비효율적

# 피보나치 수열:단순 재귀 소스코드

```
1 import java.util.*;
2
3 public class Main{
4     //피보나치 수열을 재귀함수로 구현
5     public static int fibo(int x){
6         if(x == 1 || x == 2){
7             return 1;
8         }
9         return fibo(x - 1) + fibo(x - 2);
10    }
11    public static void main(String[] args){
12        System.out.println(fibo(4));
13    }
14
15 }
```



Main X

"C:\Program Files\  
3

# DP 사용조건

- DP를 사용하려면 2가지 조건을 만족해야 함
- 1) Overlapping Subproblems(겹치는 부분 문제)
- 2) Optimal Substructure(최적 부분 구조)

# 1)Overlapping Subproblems (겹치는 부분 문제)

- DP는 기본적으로 문제를 나누고 그 문제의 결과 값을 재활용해서 전체 답을 구한다. 그래서 **동일한 작은 문제들이 반복하여 나타나는 경우에 사용이 가능**
- 부분 문제가 반복적으로 나타나지 않는다면 재사용이 불가능하니 부분 문제가 중복되지 않는 경우에는 사용할 수 없다.



## 2)Optimal Substructure(최적 부분 구조)

- 부분 문제의 최적 결과 값을 사용해 전체 문제의 최적 결과를 낼 수 있는 경우를 의미
- 부분 문제에서 구한 최적 결과가 전체 문제에서도 동일하게 적용되어 결과가 변하지 않을 때 DP를 사용할 수 있게 된다.

# DP 과정

- 1) DP로 풀 수 있는 문제인지 확인한다.
- 2) 문제의 변수 파악
- 3) 변수 간 관계식 만들기(점화식)
- 4) 메모하기(memoization or tabulation)
- 5) 기저 상태 파악하기
- 6) 구현하기

# 1.DP로 풀 수 있는 문제인지 확인

- 현재 직면한 문제가 작은 문제들로 이루어진 하나의 함수로 표현될 수 있는지를 판단해야 한다.

=>이 과정자체가 어려움, 문제를 많이 풀면서 감 잡기

- 보통 특정 데이터 내 최대화 / 최소화 계산을 하거나 특정 조건 내 데이터를 세야 한다거나 확률 등의 계산의 경우 DP로 풀 수 있는 경우가 많다.

## 2) 문제의 변수 파악

- DP는 현재 변수에 따라 그 결과 값을 찾고 그것을 전달하여 재 사용하는 것을 거친다.
- Ex. 피보나치수열에서는  $n$ 번째 숫자를 구하는 것이므로  $n$ 이 변수가 됨

### 3) 변수 간 관계식 만들기(점화식)

- 예를 들어 피보나치 수열에서는  $f(n) = f(n-1) + f(n-2)$

### 4) Memoization

변수 간 관계식까지 정상적으로 생성되었다면 **변수의 값에 따른 결과를 저장**해야 한다

### 5) 기저상태 파악하기

가장 작은 문제의 상태를 알아야 한다.

보통 몇 가지 예시를 직접 손으로 테스트하여 구성하는 경우가 많다.  
피보나치 수열을 예시로 들면,  $f(0) = 0$ ,  $f(1) = 1$ 과 같은 방식

## 6) 구현하기

1) Bottom-Up (Tabulation 방식) - 반복문 사용

2) Top-Down (Memoization 방식) - 재귀 사용

# 1) Bottom-Up (Tabulation 방식) - 반복문 사용

- 아래에서부터 계산을 수행 하고 누적시켜서 전체 큰 문제를 해결하는 방식
- 메모를 위해서 dp라는 배열을 만들었고 이것이 1차원이라 가정했을 때,  $dp[0]$ 가 기저 상태이고  $dp[n]$ 을 목표 상태라고 하자.  
Bottom-up은  $dp[0]$ 부터 시작하여 반복문을 통해 점화식으로 결과를 내서  $dp[n]$ 까지 그 값을 전이시켜 재활용하는 방식
- 결과값을 기억하고 재활용한다는 측면에서 메모하기(Memoization)와 크게 다르지 않다.

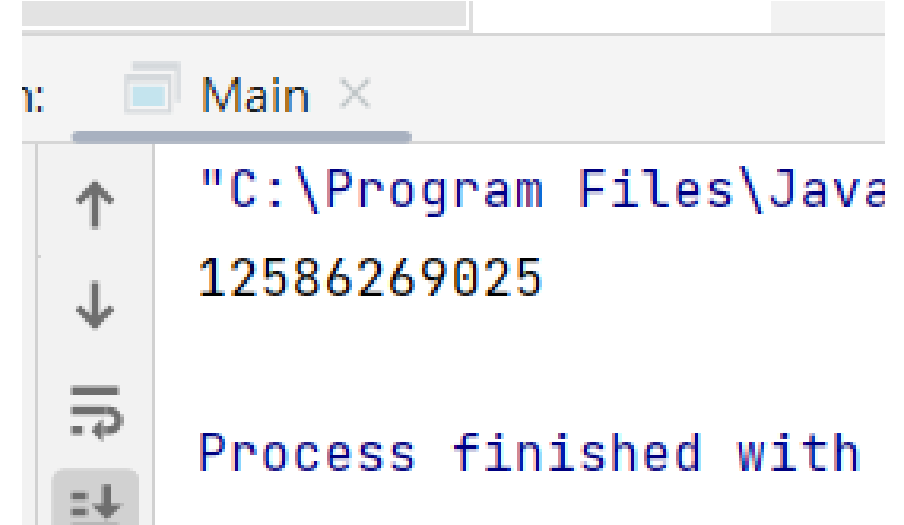
## 2) Top-Down (Memoization 방식) - 재귀 사용

- $dp[0]$ 의 기저 상태에서 출발하는 대신  $dp[n]$ 의 값을 찾기 위해 위에서 부터 바로 호출을 시작하여  $dp[0]$ 의 상태까지 내려간 다음 해당 결과 값을 재귀를 통해 전이시켜 재사용하는 방식이다.



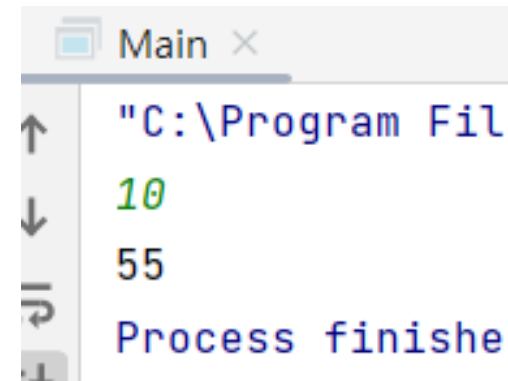
# 피보나치 수열: Bottom-Up 방식

```
1 import java.util.*;
2
3 public class Main{
4     public static long[] d = new long[100];
5     public static void main(String[] args){
6         //첫 번째 피보나치 수와 두 번째 피보나치 수는 1
7         d[1]=1;
8         d[2]=1;
9         int n=50; //50번째 피보나치 수를 계산
10
11         //피보나치 함수를 반복문으로 구현 (보텀업 다이나믹 프로그래밍)
12         for(int i=3; i<=n; i++){
13             d[i]=d[i-1]+d[i-2];
14         }
15         System.out.println(d[n]);
16     }
17 }
```



# 피보나치 수열: Top-Down 방식

```
3 public class Main {
4
5     static long[] memo;
6     // 메모이제이션 배열 생성
7     static long fibonacci(int n) {
8         if(n<=1) {
9             // 0과 1은 답을 구할 수 없는 가장 작은 수
10            return n;
11        }else {
12            if(memo[n]>0) {
13                // 메모 되어 있다면
14                return memo[n];
15                // 메모 값을 리턴
16            }
17
18            memo[n] = fibonacci(n-1) + fibonacci(n-2);
19            // 재귀 호출 보내서 받은 값을 메모에 담고
20            return memo[n];
21            // 메모를 리턴
22        }
23    }
24
25    public static void main(String[] args) throws IOException {
26
27        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
28        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));
29        // reader writer 생성
30        memo = new long[91];
31
32        int n = Integer.parseInt(br.readLine());
33
34        long answer = fibonacci(n);
35
36        bw.write(String.valueOf(answer));
37        bw.flush();
38
39    }
```



```
Main ×
"C:\Program Fil
10
55
Process finishe
```

# 피보나치 시간 복잡도

- $O(2^n) \rightarrow O(N)$  로 개선

## Bottom-Up vs Top-Down

- 효율성 비교 => 직접 해봐야 알 수 있다. 뭐가 더 나은지는 해 보기 전까지 알 수 없다.