구디 아카데미
알고리즘 스터디
중급반 - 6주차

[멘토 유준혁]

스터디 주제

- 동적 계획법 (Dynamic Programming)
- 메모이제이션 (Memoization)
- Top-Down, Bottom-Up 방식 설명

동적 계획법 (Dynamic Programming)이란 무엇인가?

> **동적 계획법이란 복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법을 말한다.**

이름과는 달리 전혀 다이나믹 하지도 않고 게다가 프로그래밍이라고 하기도 좀 그렇다.

알고리즘이라고 하기 보단 오히려 **문제 해결 방식** 에 가깝다고 볼 수 있다.

이는 여러가지 설이 있는데 고안자인 벨만 (Richard E. Bellman) 이

Dynamic 이라는 단어가 멋있어서 작명에 끼워넣었다. 라는 말도 있고

유동적으로 대처한다. 는 뜻으로 Dynamic을 넣었다는 말도 있다.

Programming은 벨만이 근무하던 연구소 특성 상

Process라고 작명을 하게 되면 투자를 받기 어려워서 그랬다는 말이 가장 유력하다고 한다.
궁금하시면 나중에 찾아보시길!

이름이야 어찌 됐든 정의를 살펴보자면, 문제가 주어질 경우 이를 작은 문제로 쪼개어 푸는 방식을 동적 계획법 이라고 표현한다고 한다. 말로 보면 이해가 잘 안가니 예제를 살펴보자.

저번 시간에 재귀함수에 대해서 공부할 때 피보나치 수열을 연습해보았었다.

피보나치 수열 = N번째의 수가 N-1번째와 N-2번째 의 합으로 이루어진 수열

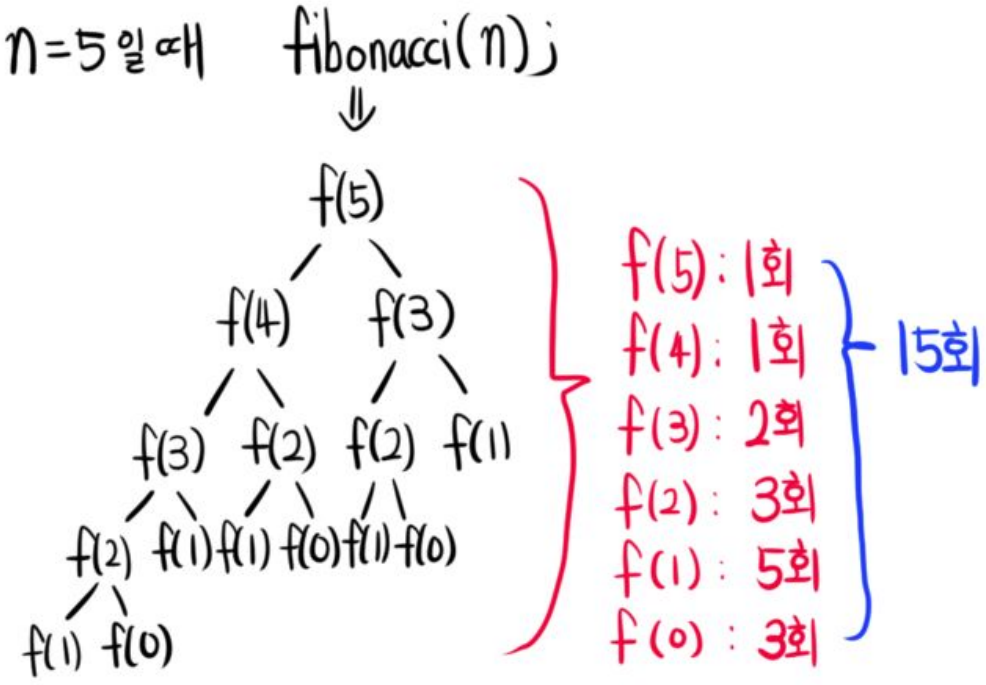
우리는 이를 점화식으로 나타내고 [f(n) = f(n-1) + f(n-2)]

만들어진 점화식을 그대로 재귀함수로 구현해보았다.

이 역시도 넓은 범위에선 동적 계획법 중 하나에 속한다.

그러나 위의 점화식대로 코딩을 했을 경우 심각한 결함이 존재한다.

지난 시간에도 설명했듯이 수가 조금만 커져도 함수의 호출이 기하급수적으로 증가한다는 점이다.

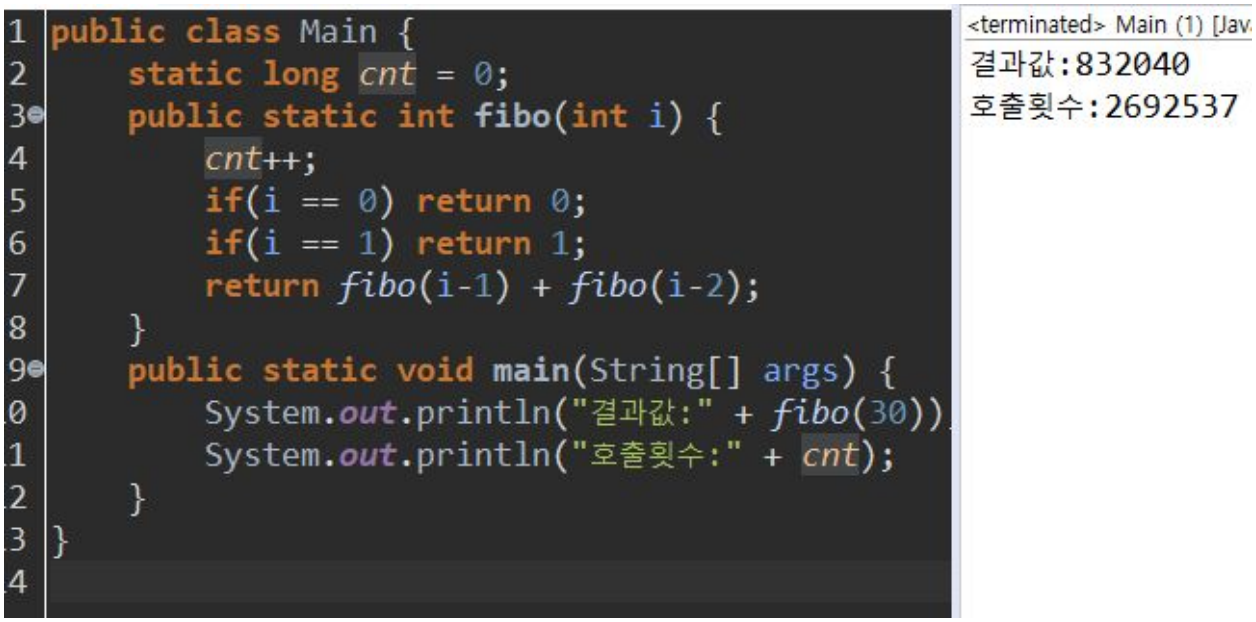


위의 그림처럼 같은 함수를 여러번 호출하는 경우가 생기기 때문에 시간복잡도가 $O(2^n)$ 가 된다.

10번째 피보나치 수를 구하려고 한다면 2의 10승인 1000번 가량을 계산해야 한다는 말이다.

(물론 실제 시도한 횟수는 다를 수 있다. 시간복잡도는 점근적 분석 방법일 뿐이다.)

그렇다면 시간복잡도를 줄이려면 어떻게 해야 할까?



```java
public class Main {
    static long cnt = 0;
    public static int fibo(int i) {
        cnt++;
        if(i == 0) return 0;
        if(i == 1) return 1;
        return fibo(i-1) + fibo(i-2);
    }
    public static void main(String[] args) {
        System.out.println("결과값:" + fibo(30));
        System.out.println("호출횟수:" + cnt);
    }
}
```

```
<terminated> Main (1) [Jav
결과값:832040
호출횟수:2692537
```

(일반적 방식의 피보나치 함수 호출 횟수)

메모이제이션 (memoization)이란 무엇인가?

위와 같은 문제 때문에 고안한 방식이 메모이제이션이다.

메모이제이션이란 **계산한 값이 존재한다면 그 값을, 아니라면 새로이 계산하는 방식을 뜻한다.**

우선 위의 기본적인 방식과 그 시도횟수를 보자.

30번째 위치한 피보나치 수를 찾기 위해 300만에 가까운 함수를 호출했다. 굉장히 비효율적이다.

그렇다면 이번엔 메모이제이션 정의대로 구현을 해보자.

가장 먼저 **계산한 값을 저장할 공간이 필요하다.**

이는 **맨 처음 함수를 호출할때의 i 값과 동일한 크기의 배열**을 가지면 된다.

다음으로는 함수를 호출하면서 이미 계산 된 함수라면 그 값을, 아니라면 새로이 계산하면 된다.

```java
public class Main {
    static long cnt = 0;
    static long[] val;
    public static long fibo(int i) {
        if(val[i] != 0) return val[i];
        cnt++;
        if(i == 0) return 0;
        if(i == 1) return 1;
        return val[i] = fibo(i-1) + fibo(i-2);
    }
    public static void main(String[] args) {
        val = new long[91];
        System.out.println("결과값:" + fibo(90));
        System.out.println("호출횟수:" + cnt);
    }
}
```

```
<terminated> Main (1) [Java Application] C:₩P
결과값:2880067194370816120
호출횟수:92
```

(메모이제이션을 활용한 피보나치 수열 구하기)

너무 간단하지만 정말 이게 끝이다. **저장해두었다가 다시 사용하는 것.**

이 방식대로만 한다면 시간복잡도를 현저히 낮출 수 있다.

피보나치 수열 같은 경우 i번째 값을 얻기 위해 한 번의 계산만 필요하므로

시간복잡도가 O(N) 까지 줄어들게 된다.

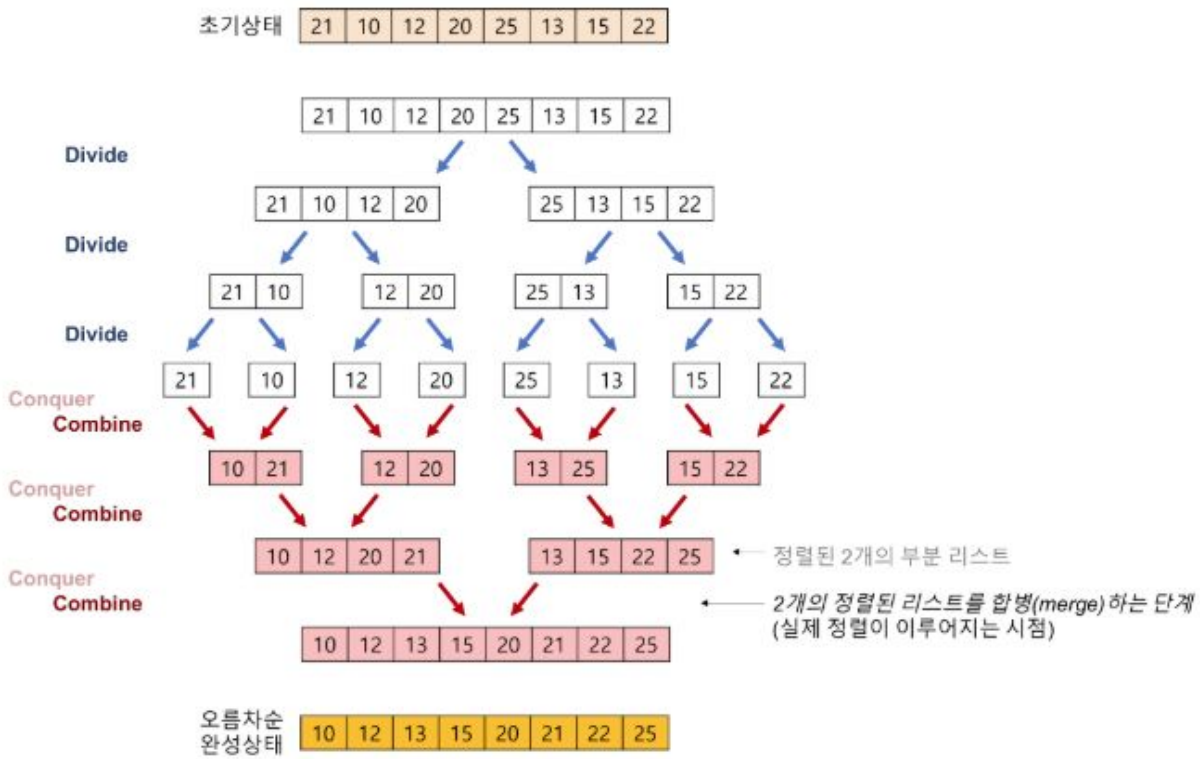
분할 정복 (Divide And Conquer) 과의 차이점

정보처리 자격증을 공부하면서 병합 정렬 에 관해 들어본 적이 있을거라 생각한다.

이 병합 정렬에서 사용하는 방식이 바로 분할 정복인데 동적 계획법과 상당히 유사하다.

병합 정렬이란 기존 배열을 더 이상 나눠지지 않을 때 까지 이등분을 한 뒤

다시 합치면서 오름/내림으로 정렬을 하는 방식을 뜻한다.



(병합 정렬 그림 표현)

병합 정렬도 마찬가지로 큰 문제 **(전체 배열 정렬)** 를 작은 문제 **(부분 배열 정렬)** 로 나누고

이를 이용해 다시 큰 문제를 구하는 걸 보면 동적 계획법과 크게 다르진 않다.

> 굳이 차이점을 찾자면 병합 정렬은 **최대한 겹치지 않도록, 반드시 구분 되도록** 나눈다고 한다면
>
> 동적 계획법에서는 이와 반대로 **최대한 겹치도록** 나눈다는 점이 다르다.

Top - Down 방식 (하향식 계산법) 이란?

다이나믹 프로그래밍에서 대해서 알아보았으니 이번엔 그 속에서도 사용 가능한 방식을 알아보자.

다이나믹 프로그래밍이란 **큰 문제를 작은 문제로 분할하여 푸는 방식**이라고 했었다.

거기에 예시들로 설명했던 코드들이 모두 Top - Down 방식이다.

큰 문제부터 시작해서 작은 문제로 쪼개고 이후에 답을 구하는 방식 말이다.

그렇다면 여기서 다음과 같은 질문이 나올 수도 있다.

> ***'굳이 큰 문제를 작은 문제로 분할해야 되나? 애초에 작은 것 부터 시작하면 되지'***

맞는 말이다.

시작부터 작은 문제로 시작해서 큰 문제의 답을 구하는 방식 역시 가능해야 한다.

이는 시점의 차이로 볼 수도 있고 초심자가 DP를 이해하기 어렵다고 여기게 되는 부분 중 하나이다.

기본적인 Top - Down 방식은 설명했던 것 처럼 큰 문제를 작은 문제들로 쪼개고

이를 이용해 다시 큰 문제의 답을 구하게 된다.

어떻게 보면 비효율적이라고 느낄 수도 있다. 편도로 직빵으로 가면 될 걸 왕복하는 셈이니까 (?)

그렇다면 왕복이 아닌 편도로 가는 방법을 알아보자.

Bottom - Up 방식 (상향식 계산법) 이란?

큰 문제를 도착점으로, 작은 문제들 중 가장 작은 문제를 시작점으로 지칭해보자.

피보나치 수열의 경우엔 0 인 경우가 시작점이 될 것이고 i인 경우가 도착점이 될 것이다.

여기서 f(n-1) , f(n-2)를 저장할 변수 두개와 f(n) 을 저장할 변수만 두고

순차적인 반복문을 사용한다면 Bottom - Up 방식이 된다.

```
public int fibo(int n){
	if(n < 2) return n;
	int a = 0, b = 1;
	for(int i = 2; i <= n; i++) {
		int tmp = a + b;
		a = b;
		b = tmp;
	}
	return b;
}
```

(피보나치 수열 Bottom - Up 방식)

이처럼 작은 문제에서 시작하여 큰 문제로 도달하는 방식을 Bottom - Up 이라고 부른다.

이런 좀 더 쉬운 방식이 있음에도 Top - Down을 먼저 설명한 이유가 있다.

> **Bottom - Up 방식의 진행 과정을 비유해보자면 100미터 달리기와 같다.**
>
> **출발선에서 시작하여 도착선까지 직진만 하면 된다는 점에서 그렇다.**
>
> **바꿔말하면 Bottom - Up 방식은 일종의 선형적인 과정을 겪게 된다는 말이 된다.**

피보나치 같은 경우에서는 별 문제가 되지 않겠지만 일반적인 DP 문제들은 선형적이지가 않다.

그래프나 트리처럼 비선형 자료구조, 또는 이차원 배열을 특이한 방식으로 접근하는 경우가 많으므로

주로 Top - Down의 방식이 사용된다.

게다가 점화식을 찾고 그대로 구현하는 방식이 바로 Top - Down이기 때문에 더더욱 그렇다.

PS.

DP (Dynamic Programming) 문제들은 실제 코딩테스트

또는 알고리즘 대회에서 절대 빠지지 않는 문제라고들 여긴다.

조건문, 기본 자료구조, 알고리즘의 전체 흐름, 문제 파악 능력, 규칙 도출 능력, 문제 해결 능력 등

모든 부분에서의 능력이 점검 가능하므로 기업에선 거의 필수로 내는 문제 분류이다.

삼성 같은 경우엔 대부분의 문제가 DP로 이루어졌다는 말이 있을 정도다.

그럼에도 불구하고 DP는 압도적 재능이 아닌 이상 누구나 노력만으로 풀 수 있다고 생각한다.

최대한 많은 문제를 접해보고, 도전해보고 실패해야 더 잘 풀 수 있게 될 것 이다.

도전하고 실패해야 얻는 게 더 많다.

여러번 무모하게 도전하고 실패의 경험을 쌓길 바란다.