정윤성

동적 계획법

Dynamic Programming

By POSCAT

피보나치 수열

- 아래와 같이 정의되는 수열이다.
- F(N) = F(N-1) + F(N-2);
- F(0) = F(1) = 1;

- 피보나치 수열의 N번째 항 F(N)을 구해보자.

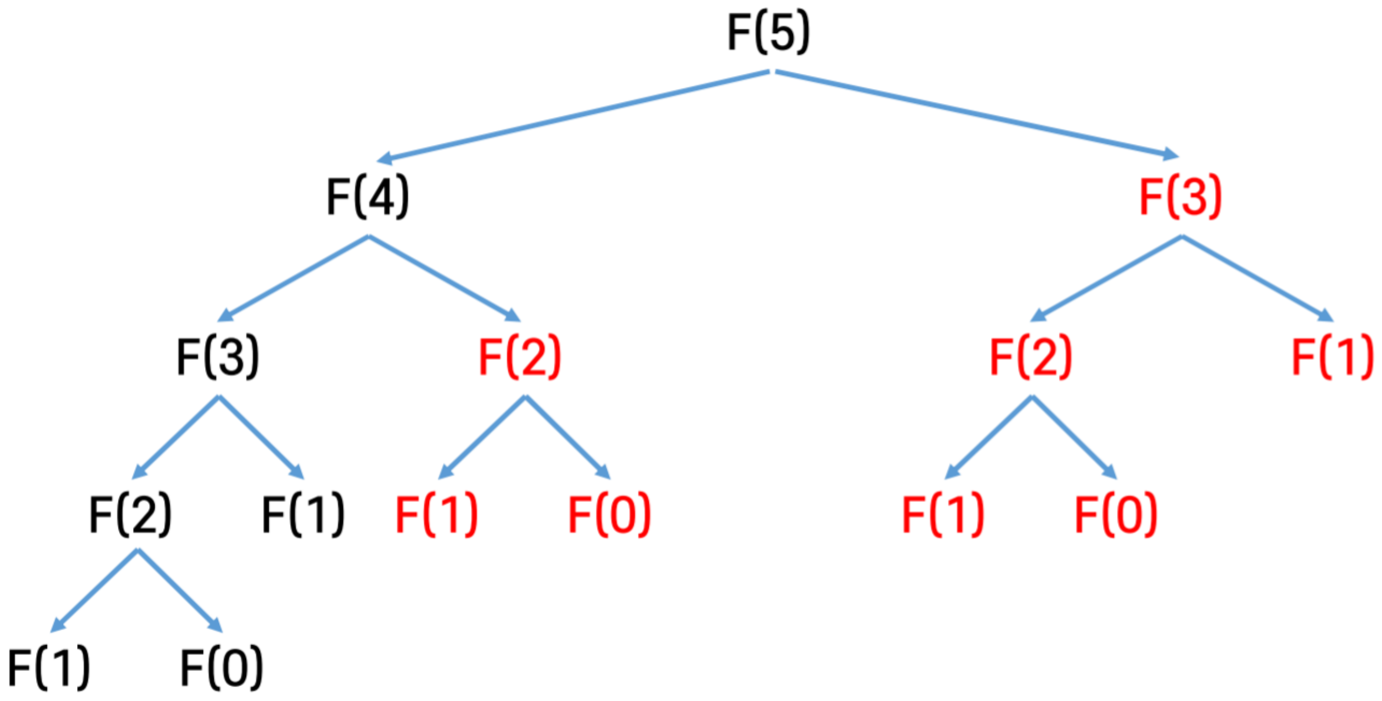
피보나치 수열 - 재귀함수

Code Explanation

```
int F(int n){

    if(n == 0 || n == 1) return 1;

    return F(n-1) + F(n-2);

}
```

피보나치 수열 - 재귀함수

- 앞선 코드를 적용하면 O(F(N))만큼의 호출이 발생한다.



DP

- 앞선 코드의 문제점은, 이미 구해놓은 F(k) 값을 다시 구하는 일이 반복된다는 점이다.
- 예를 들어 F(5)가 F(4),F(3)을 호출하는데 F(4)가 또 F(3)을 호출한다.

- Dynamic Programming(DP)
- 부분문제의 결과값이 여러 번 사용될 때 부분문제의 결과를 저장해두고 이후의 계산에 사용하는 테크닉이다
- 값을 저장해두는 것을 메모이제이션(Memoization) 이라고 한다.

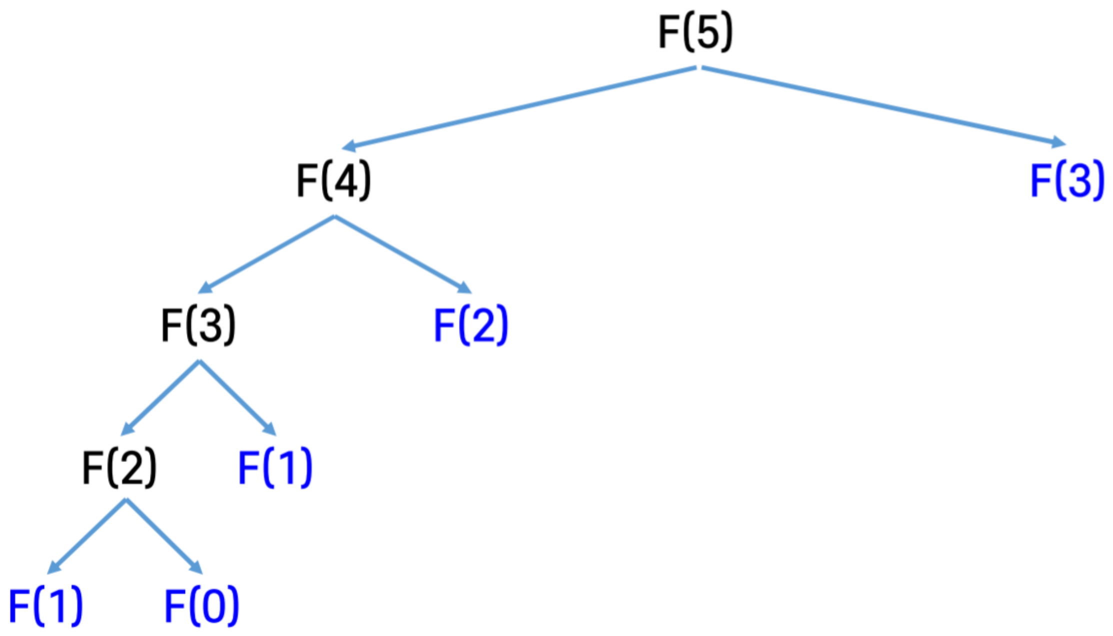
피보나치 수열 - DP

Code Explanation

```
int dp[100000];  //모두 0으로 초기화

int F(int n){

    if(n == 0 || n == 1) return 1;

    if(dp[n] != 0) return dp[n];   ///이미 구한 값이 있다면 그냥 return

    return dp[n] = F(n-1) + F(n-2);    //dp[n]에 구한 값을 대입함

}
```

피보나치 수열 - DP

- 메모이제이션을 적용하고 나면 한번 구한 값을 O(1)만에 가져올 수 있다.
- 결과적으로 O(N) 시간 내에 작동한다.



피보나치 수열 – Bottom up

Code Explanation

```
int F[100000];

F[0] = F[1] = 1;

for(int i=2;i<=N;i++){

    F[i] = F[i-1] + F[i-2];

}
```

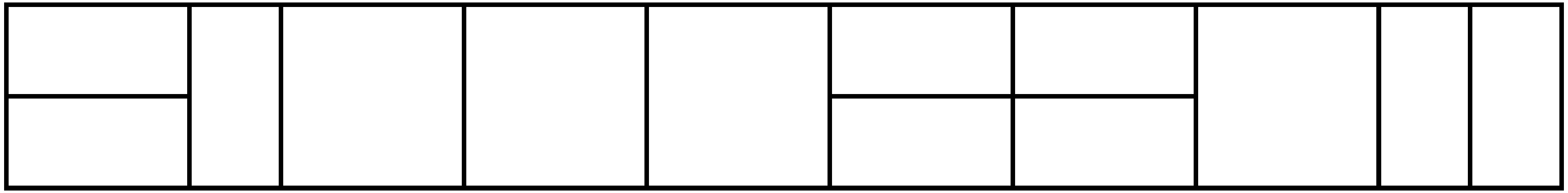
Top-down vs Bottom up

- Top-down은 F(N)을 호출하여 그것들이 순차적으로 부분문제를 호출하는 방식이다. (재귀적 구현)
- 사람의 생각대로 구현하기 편리하다.

- Bottom-up 방식은 F(1) ~ F(N-1)을 순서대로 구하여 F(N)을 조립하는 방식이다. (반복문으로 구현)
- 탑다운 방식에 비해 메모리, 시간을 약간 덜 사용한다는 장점이 있다.

2xN 타일링

- 2xN의 타일을 2x1 타일로 완전히 덮는 경우의 수를 생각해보자.
- N = 2 -> 2가지
- N = 3 -> 3가지
- N = 4 -> 5가지
- N = 5 -> 8가지
- 어디서 많이 본 수열인데?



2xN 타일링

- DP[i]: 2*i 칸에 2*1 타일로 타일링을 하는 경우의 수
- 현재 칸에는 세로로 하나의 타일을 놓거나, (DP[i-1])
- 가로로 2개의 타일을 놓을 수밖에 없다. (DP[i-2])
- 이제 DP[i]는 피보나치 수와 같게 된다.

1로 만들기

- 어떤 정수에 아래의 세 가지 연산만을 사용해 1로 만드는데 필요한 연산의 최소 횟수를 구해 보자.

1. 3의 배수이면 3으로 나눈다.
2. 2의 배수이면 2로 나눈다.
3. 1을 뺀다.

1로 만들기

- DP[i]: i를 1로 만드는 데 필요한 연산의 최소 수
- DP[i] = min(DP[i/3] (3의 배수이면), DP[i/2] (2의 배수이면), DP[i-1]) + 1 이다.
- DP[1] = 0이다.

1로 만들기 – top down

Code Explanation

```
모든 i에 대해 dp[i] = INF로 초기화

int go(int n){

    if(n==1) return 0;

    if(dp[n] != INF) return dp[n];

    if(n%3 == 0) dp[n] = min(dp[n], go(n/3) + 1);

    if(n%2 == 0) dp[n] = min(dp[n], go(n/2) + 1);

    dp[n] = min(dp[n], go(n-1) + 1);

    return dp[n];

}
```

1로 만들기 – bottom up

Code Explanation

```
모든 i에 대해 dp[i] = INF로 초기화

dp[1] = 1;

for(int i=1;i<=N;i++){

    dp[i*3] = min(dp[i*3], dp[i] + 1);

    dp[i*2] = min(dp[i*2], dp[i] + 1);

    dp[i+1] = min(dp[i+1], dp[i] + 1);

}
```

Longest Increasing Subsequence

- 모든 부분수열 중 증가이며 길이가 가장 긴 것을 구하는 문제이다.
- 부분수열의 개수는 2^N가지이고, 증가하는지를 검사하려면 그 길이만큼의 반복이 필요하다.
- 즉 완전탐색을 하면 O(N * 2^N)가 걸린다. 더 빠르게 해결할 수 없을까?

LIS Sol. by DP

- DP[i]: i번째 값을 사용할 때 LIS의 길이
- DP[i] = max{DP[j]} + 1, for all j<i and arr[j]<arr[i]
- 이제 답은 모든 DP[i] 중 최댓값이 된다.

	10	20	40	25	20	50	30	70	85
i	0	1	2	3	4	5	6	7	8
DP[i]	1	2	3	3	2	4	4	5	6

DP 문제에서 주의할 점

- 반복 횟수가 몇 억(10^8) 번을 넘어가면 1초 내에 동작할 수 없다. N의 제한을 보고 DP로 풀릴지를 판별하자.
- Top-down에서 dp[]를 초기화하는 값은 실제로 나올 수 없는 값이어야 한다. 보통 (-1)이나 (INF=2e9) 정도를 사용한다.
- dp[i]의 값이 2^31 (약 21억)을 넘는 것 같다면 int 대신 long long을 사용하는 등 자료형에 주의.

오늘의 문제

- 그룹 내에서 문제집 확인

POSCAT

Thank you :-)