POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

[CSED213-01] – 2024 1학기

문제해결 실습 및 응용

Dynamic Programming

20232733 CG Lab Beomsu Kim

POSTECH 컴퓨터공학과 통합과정

qjatn0120@postech.ac.kr

Contents

Dynamic Programming

- ✓ 동적 계획법 (Dynamic Programming)
  - 어려운 문제를 쉬운 문제들로 나누어 푼 뒤 그 결과를 이용하여 어려운 문제를 푸는 기법
  - 이미 풀어서 답을 아는 부분의 결과를 다시 푸는 경우에는
    해당 값을 다시 계산하는 대신 이전에 계산했던 결과를 가져와서 사용
- ✓ 문제의 풀이가 점화식 꼴인 경우, 동적 계획법을 통해 풀 수 있다.
  - 예시) 피보나치 수열

Dynamic Programming

✓ 피보나치 수열 구하기

- $F_n = F_{n-1} + F_{n-2},\ F_0 = F_1 = 1$
- F_n이라는 문제를 풀기 위해서는 F_{n-1}과 F_{n-2}라는 2개의 작은 문제를 풀어야 한다.
- 이 문제를 재귀 함수로 구현하면 아래와 같다.

```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

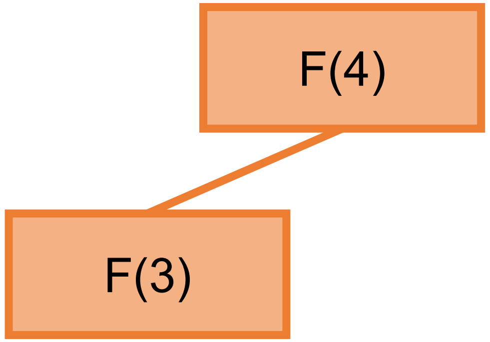
- ✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.

F(4)

```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

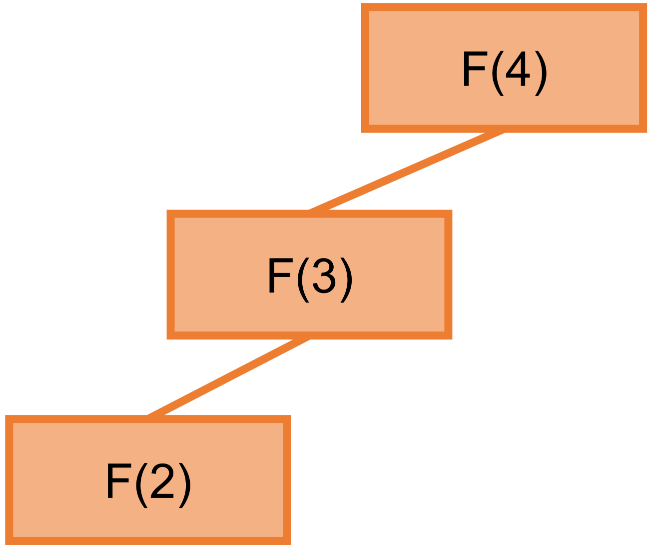
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

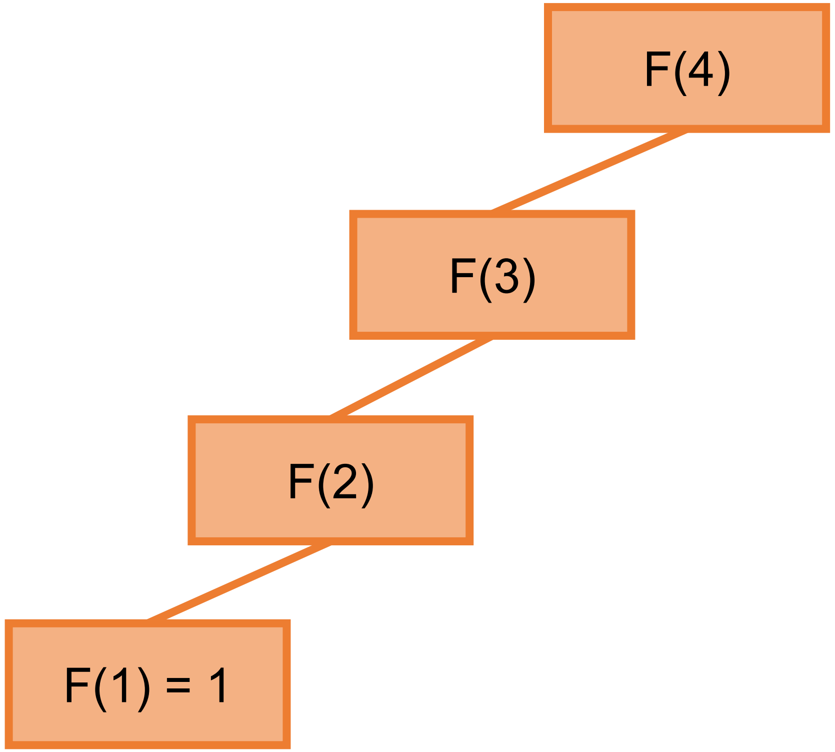
- ✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

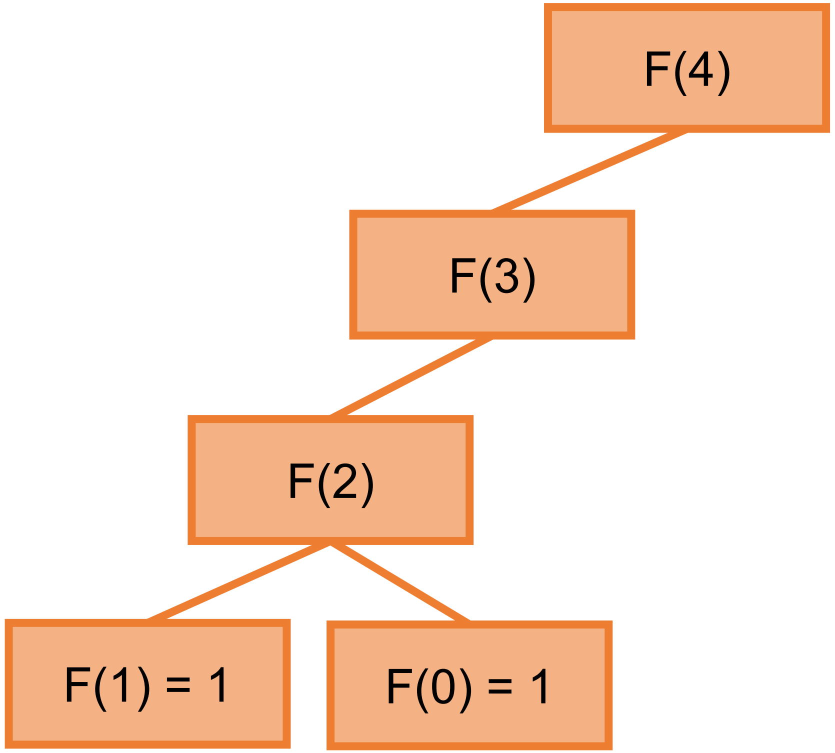
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

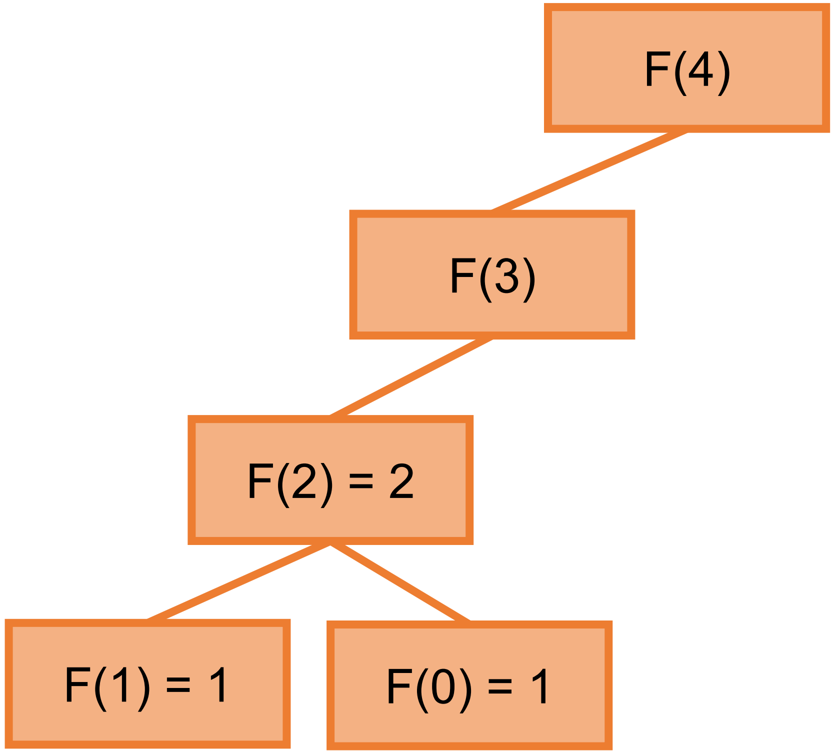
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

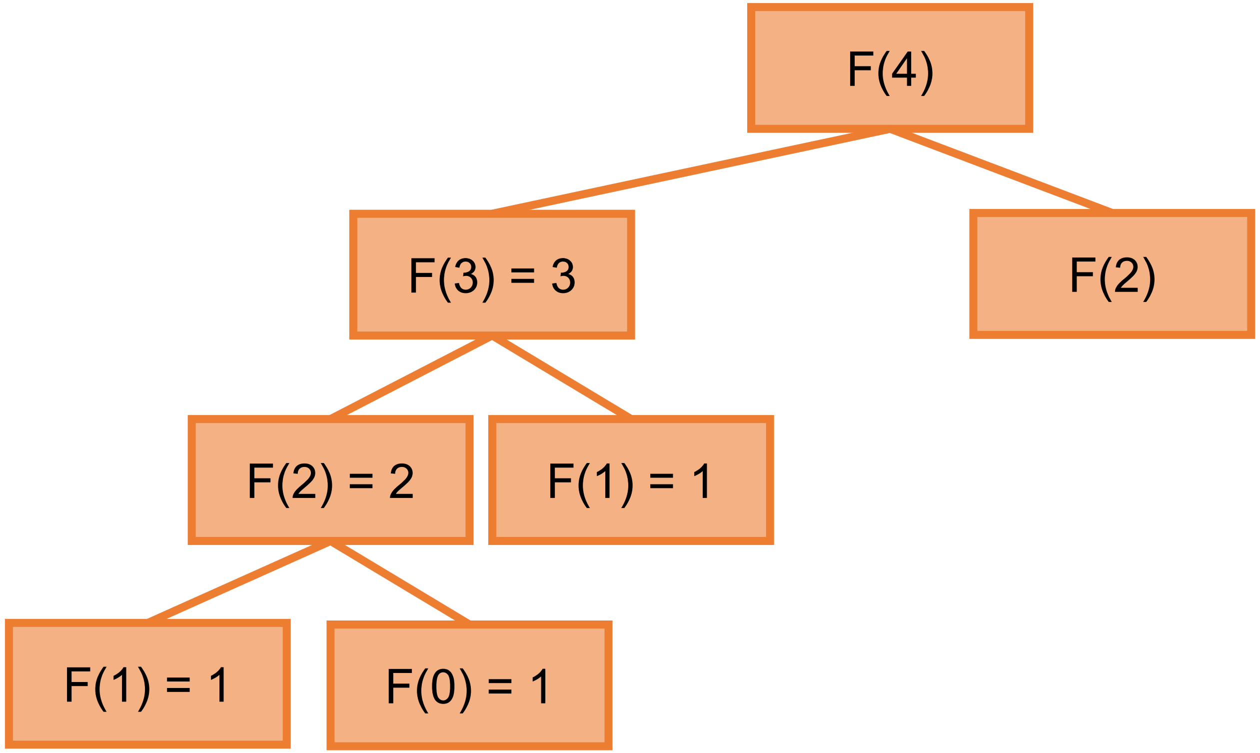
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

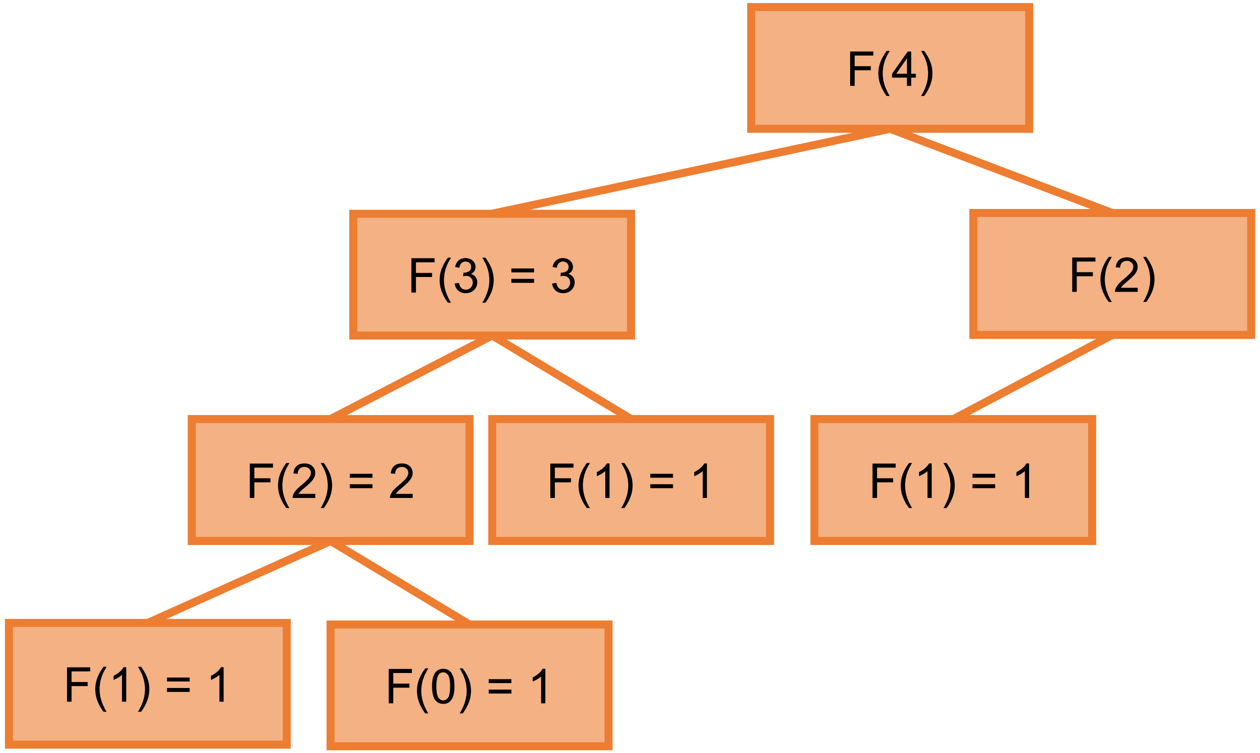
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

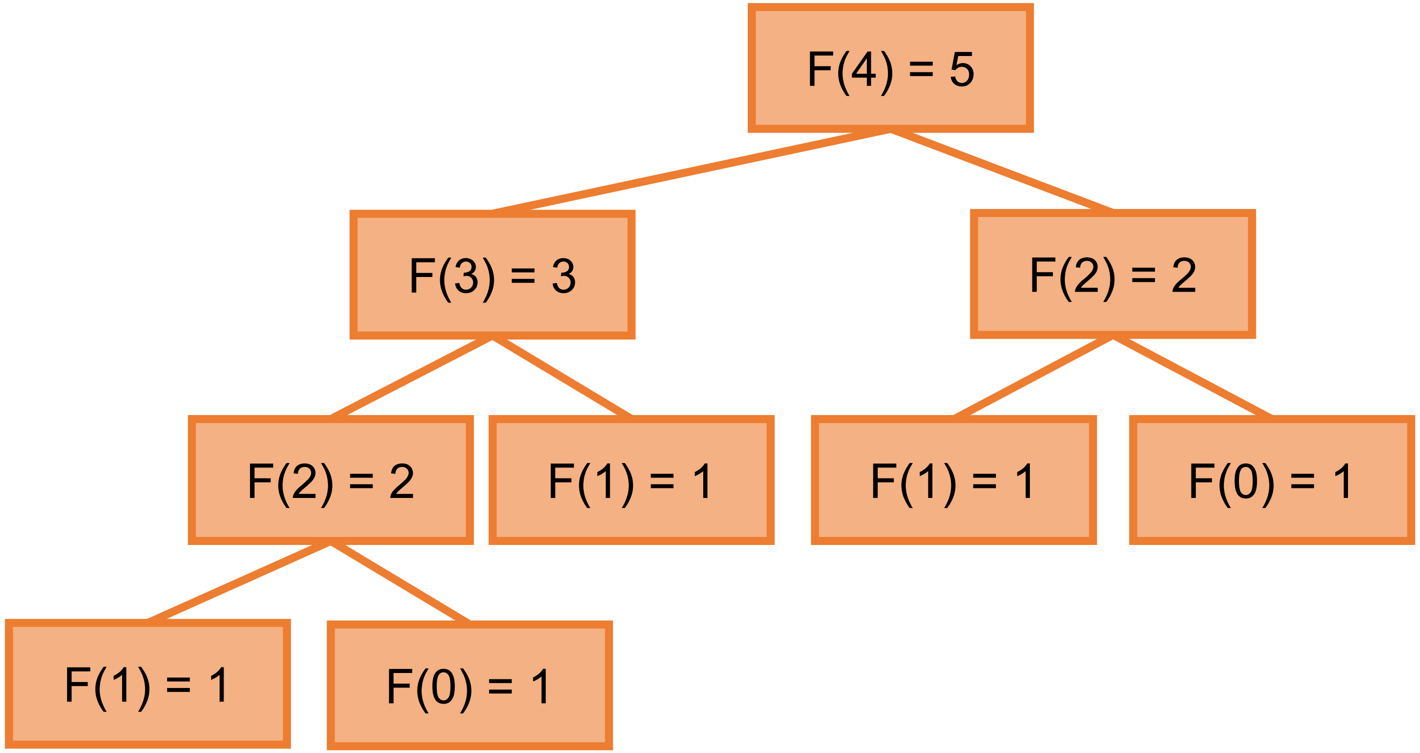
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.




```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.

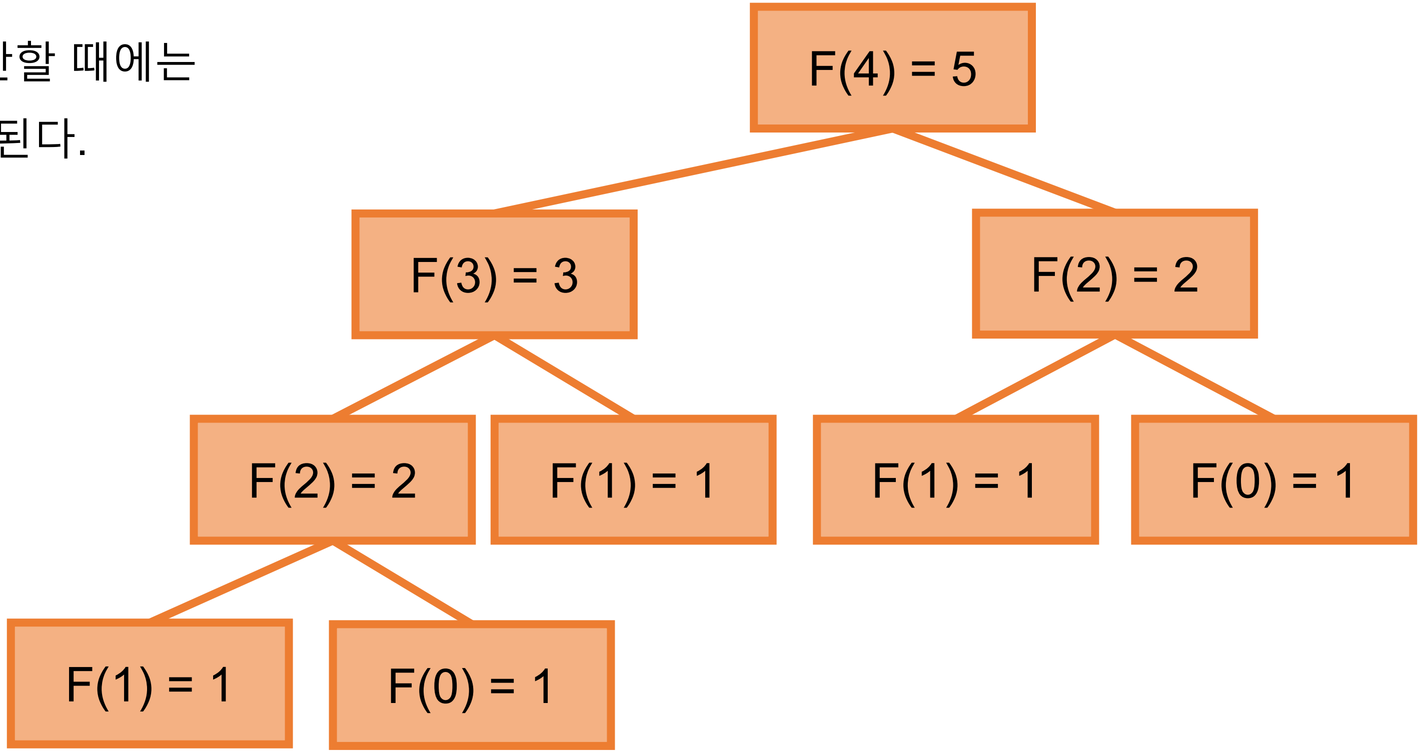



```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.

- 아래의 과정에서 F(2)는 총 2번 계산하였다.
- F(2)를 계산하기 위해 F(1)과 F(0)을 호출하고 그 결과를 더해서 계산하였다.
- 하지만, F(2)를 2번이나 계산할 필요가 있을까?
- 처음에는 F(2)를 계산해야 하지만, 2번째로 계산할 때에는 이전에 계산하였던 F(2)의 값을 다시 사용하면 된다.




```c
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Dynamic Programming

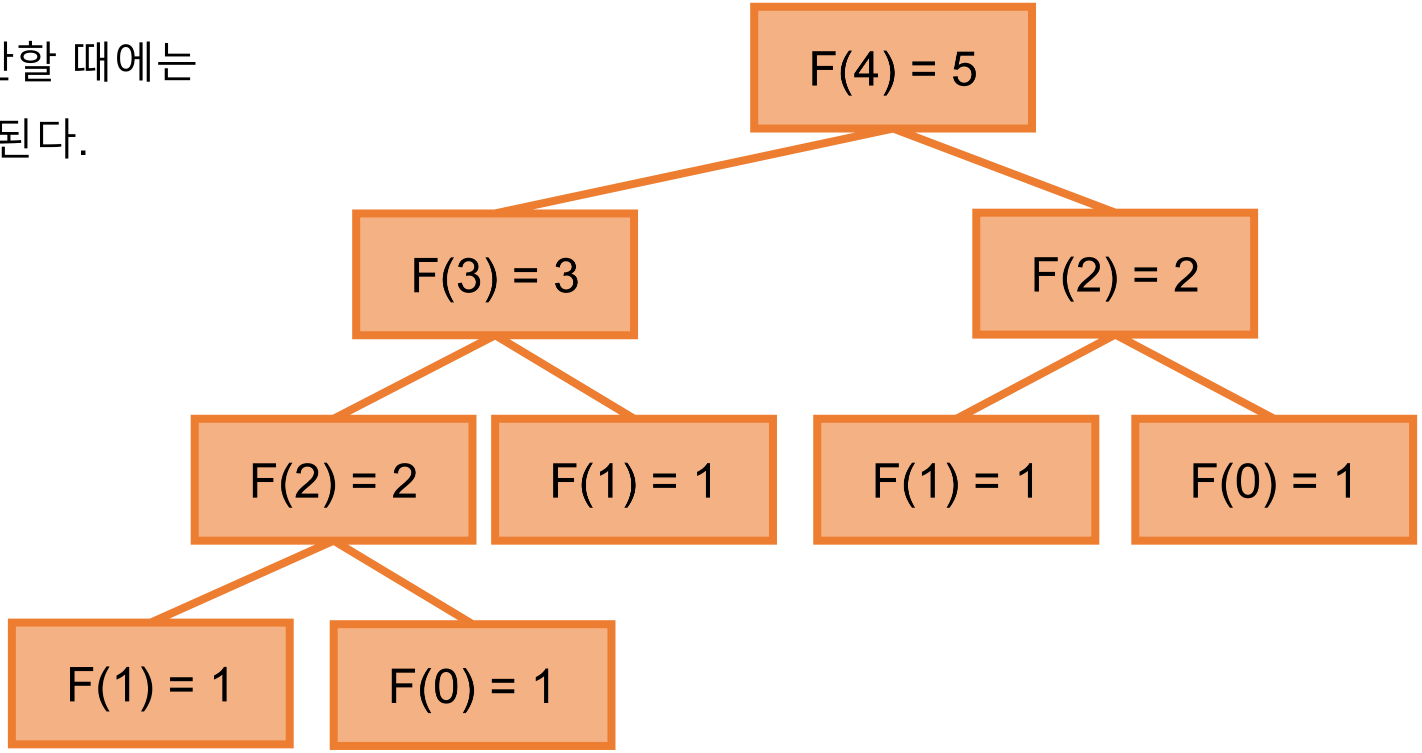
✓ 아래의 코드를 이용해서 F(4)를 구하는 과정을 생각해보자.

- 아래의 과정에서 F(2)는 총 2번 계산하였다.
- F(2)를 계산하기 위해 F(1)과 F(0)을 호출하고 그 결과를 더해서 계산하였다.
- 하지만, F(2)를 2번이나 계산할 필요가 있을까?
- 처음에는 F(2)를 계산해야 하지만, 2번째로 계산할 때에는 이전에 계산하였던 F(2)의 값을 다시 사용하면 된다.

✓ 메모이제이션 (Memoization)

- 계산한 결과를 배열에 저장하자.
- 그리고 다시 계산하는 대신에 배열에 저장한 값을 이용하자.

```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```



Dynamic Programming

✓ 메모이제이션을 적용하면 코드는 아래와 같이 변한다.

```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

⇨

```
vector <int> mem(100);

int f(int n){
    if(n <= 1) return 1;
    if(mem[n]) return mem[n]
    return mem[n] = f(n - 1) + f(n - 2);
}
```

Dynamic Programming

✓ 메모이제이션을 적용하면 코드는 아래와 같이 변한다.

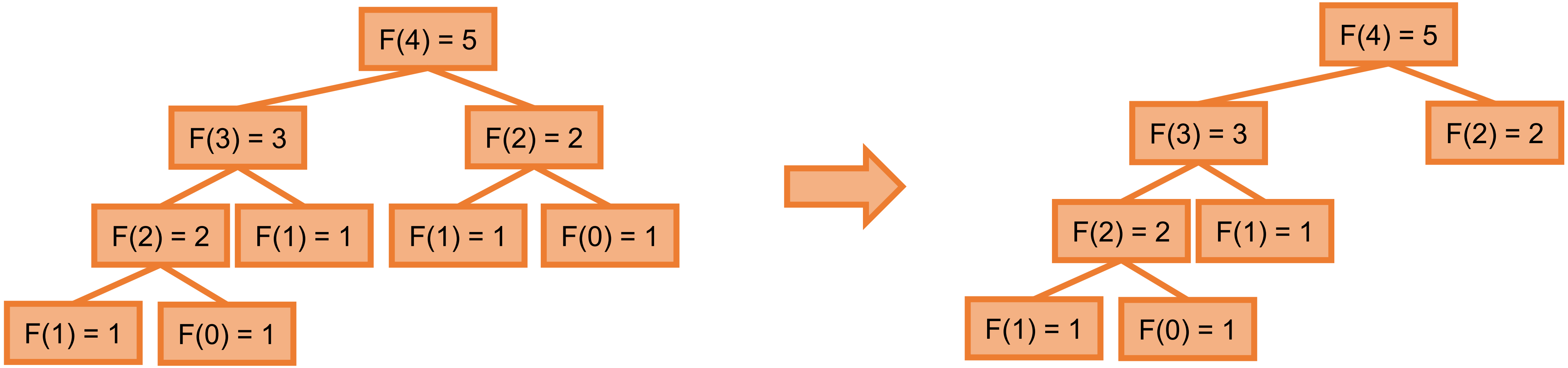
```
int f(int n){
    if(n <= 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

```
vector <int> mem(100);

int f(int n){
    if(n <= 1) return 1;
    if(mem[n]) return mem[n]
    return mem[n] = f(n - 1) + f(n - 2);
}
```

✓ 메모이제이션을 적용하면 F(4)를 계산하는 과정이 아래와 같이 변한다.

- F(2)를 다시 계산할 때, 이전에 계산한 결과를 가져왔기 때문에 F(1)과 F(0)을 계산하지 않고도 바로 계산할 수 있다.



Contents

Bottom up vs Top down

✓ Top down 방식은 어려운 문제를 풀기 위해 작은 문제를 호출해서 푸는 방식이다.

- 재귀 호출을 이용해서 푸는 방식을 말한다.
- 메모이제이션 기법을 사용한다.

✓ Bottom up 방식은 작은 문제부터 풀어가면서 어려운 문제를 푸는 방식이다.

- 반복문을 통해 푸는 방식이다.
- 재귀 호출이 없기 때문에 top down 방식과 비교해
  속도와 메모리 면에서 효율적이다.

Bottom up vs Top down

- ✓ Top down 방식은 어려운 문제를 풀기 위해 작은 문제를 호출해서 푸는 방식이다.
  - 재귀 호출을 이용해서 푸는 방식을 말한다.
  - 메모이제이션 기법을 사용한다.
- ✓ Bottom up 방식은 작은 문제부터 풀어가면서 어려운 문제를 푸는 방식이다.
  - 반복문을 통해 푸는 방식이다.
  - 재귀 호출이 없기 때문에 top down 방식과 비교해 속도와 메모리 면에서 효율적이다.

```
vector <int> mem(100);

int f(int n){
    if(n <= 1) return 1;
    if(mem[n]) return mem[n]
    return mem[n] = f(n - 1) + f(n - 2);
}
```

Top down

```
vector <int> mem(100);
mem[0] = mem[1] = 1;
for(int i = 2; i <= n; i++){
    mem[i] = mem[i - 1] + mem[i - 2];
}
```

Bottom up

Bottom up vs Top down

✓ Bottom up vs Top down

- 거의 모든 문제는 두 방식 중 어느 방식을 선택하더라도 풀 수 있다.
- 하지만 문제에 따라 bottom up 방식이 쉬울 수도 있고, top down 방식이 쉬울 수도 있다.
- 보통 점화식이 간단하면 bottom up 방식, 점화식이 복잡하면 top down 방식을 사용한다.

Contents

Example

- ✓ DP 문제에서 가장 중요한 것은 올바른 점화식을 세우는 것이다.
  - 점화식을 올바르게 세웠는지, 그리고 문제의 주어진 시간 제한을 만족하는 지를 확인해야 한다.
  - 올바른 점화식을 세우고 난 후에는 이 점화식을 코드로 구현하기만 하면 된다.
- ✓ 따라서 DP 문제를 풀기 위해서는 아래의 2가지를 잘 정의해야 한다.
  - F[n]을 어떻게 정의할 것인가?
  - F[n]을 어떻게 계산할 수 있나?
- ✓ 피보나치 수열의 경우는 아래와 같이 간단하게 풀 수 있다.
  - $F[n] = n$번째 피보나치 수
  - $F[n] = F[n-1] + F[n-2]$

Example

✓ 가장 긴 증가하는 부분 수열

- 부분 수열이란 주어진 수열에서 몇 개의 원소를 제거해서 만들 수 있는 수열이다.
  - (1, 6, 10, 4, 2)의 부분 수열은 (1, 10, 2), (6, 2), (), (1, 6, 10, 4, 2) 등이 있다.
- 증가하는 수열이란 (1, 5, 7, 10) 처럼 값이 점점 증가하는 수열을 의미한다.
- 수열이 주어졌을 때 모든 증가하는 부분 수열 중에서 가장 긴 부분 수열의 길이를 출력해야 한다.

✓ 풀이

- $F[n] = 1$번째 원소부터 n번째 원소까지 사용하였을 때, 만들 수 있는 증가하는 부분 수열의 최대 길이
- $F[n] = \max(F[i] + 1)$, $i < n \text{ and } a[i] < a[n]$
  - 1번째 원소부터 i번째 원소까지 사용하여 길이가 F[i]인 증가하는 부분 수열을 만들 수 있다.
  - a[i] < a[n]이면 여기에 n번째 원소를 추가하여 길이가 F[i] + 1인 증가하는 부분 수열을 만들 수 있다.
  - a는 입력 받은 수열을 의미한다.

Contents

DP and DnC

- ✓ 피보나치의 N번째 항을 구하는 것을 O(N)에 할 수 있다.
- ✓ 하지만 분할 정복을 이용하면 N번째 항을 O(logN)으로도 구할 수 있다!
- ✓ 그 방법은 바로 행렬 곱셈을 이용하는 것이다.
- ✓ 아래와 같이 M * N 크기의 행렬과 N * K 크기의 행렬을 곱해서 M * K 크기 행렬의 곱셈 결과를 얻게 된다.
- ✓ 행렬 곱셈의 중요한 성질은 결합 법칙이 성립한다는 것이다.
- ✓ 우리는 결합 법칙을 이용해서 피보나치의 N번째 항을 바로 구할 것이다.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

DP and DnC

✓ 먼저 피보나치 수열은 아래와 같다.

- $F_n = F_{n-1} + F_{n-2}$

✓ 따라서 행렬로 이를 아래와 같이 나타낼 수 있다.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

✓ 이를 조금 더 일반화하면 아래와 같이 나타낼 수 있다.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} = \cdots = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^{n-2} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

- 지난 시간에 거듭 제곱을 이용해서 x^n을 O(log n)에 구하는 법을 배웠다.
- 이를 응용하면, 행렬의 n승 또한 O (log n)에 계산할 수 있다!
- 피보나치와 비슷한 형태의 선형 점화식 n번째 항을 행렬 곱셈 log n번으로 구할 수 있다.

Contents

DP techniques

✓ 다차원 배열 메모이제이션

- Top down 방식
- 함수의 매개 변수가 2개 이상인 경우 다차원 배열을 이용해서 메모이제이션을 할 수 있다.
- Example) Combination
- ${}_{n}C_{r} = {}_{n-1}C_{r} + {}_{n-1}C_{r-1}$

```
vector <vector <int> > mem(100, vector <int> (100));

int C(int n, int r){
    if(n == 0 || n == r) return 1;
    if(mem[n][r]) return mem[n][r];
    return mem[n][r] = C(n - 1, r) + C(n - 1, r - 1);
}
```

- Bottom up 방식으로 구현할 경우에는 n중 반복문으로 구현할 수 있다.

DP techniques

✓ 기초 DP 최적화 (메모리 줄이기)

- Bottom up 방식으로 구현할 때 활용 가능
- 피보나치 수열에서 F[100]을 계산하기 위해 필요한 배열의 크기는 얼마일까?

DP techniques

✓ 기초 DP 최적화 (메모리 줄이기)

- Bottom up 방식으로 구현할 때 활용 가능
- 피보나치 수열에서 F[100]을 계산하기 위해 필요한 배열의 크기는 얼마일까?
  - 간단하게 생각하면 F[0]부터 F[100]까지 계산해야 하므로 배열의 크기가 101이어야 한다고 생각할 수 있다.
  - 하지만 배열의 크기가 3이기만 해도 F[100]을 계산할 수 있다.
  - 아래의 코드와 같이 F[i-2]와 F[i-1]을 mem[0]과 mem[1]에 저장하고,
    mem[2]에 F[i]를 계산하여 저장하면 된다.
  - 이는 F[i]를 계산할 때에는 F[i-2], F[i-1]만 필요하고, F[i-3], F[i-4], … 는 더 이상 필요가 없기 때문이다.
  - 이와 같이 필요가 없는 값들을 저장하지 않으면 메모리를 줄일 수 있다.

```
vector <int> mem(3);
mem[0] = mem[1] = 1;

for(int i = 2; i <= n; i++){
    mem[2] = mem[0] + mem[1];
    mem[0] = mem[1];
    mem[1] = mem[2];
}
```

C++ 구조체

- ✓ C++에서는 변수를 관리하기 위한 변수형이 존재한다.
  - 정수를 관리하기 위해서 int, long long int를 사용한다.
  - 마찬가지로 실수를 관리하기 위해서는 float, double을 사용한다.
  - 배열을 관리하기 위해서는 vector 등을 사용한다.
- ✓ 하지만 기존의 변수형으로는 원하는 변수를 효율적으로 관리하기가 힘든 경우도 있다.
  - 행렬을 관리하기 위해서는 행렬의 크기, 원소의 각 값 등을 관리해야 한다.
  - 이를 int, float, vector 등을 활용해서 관리하려면 구현이 난잡 해진다.
- ✓ 따라서 문제를 해결하기 위해 나만의 변수형을 정의할 수 있다. 이를 구조체라 한다.

C++ 구조체

- ✓ int, float와 같은 연산은 더하기, 빼기, 곱하기 등이 가능하다.
- ✓ 마찬가지로 구조체 간의 연산 또한 정의가 가능하다.
- ✓ 따라서 행렬 간 곱셈을 구현하고 싶을 때는 행렬 구조체를 정의한 후, 행렬 곱셈을 따로 정의해주면 곱하기 기호 하나만으로 행렬 간 곱셈을 수행할 수 있다.

Problem 1

- ✓ 돌다리도 두드려보고 건너야 한다 (Easy)
  - 피보나치 수열과 매우 유사한 문제
  - Fn = Fn-1 + Fn-2 + Fn-3 점화식을 사용하면 된다.
  - 몇 개의 돌은 사용할 수 없으니, 이를 고려해야 한다.

Problem 2

- ✓ 돌다리도 두드려보고 건너야 한다 (Medium)
  - 피보나치 수열과 매우 유사한 문제
  - Fn = Fn-1 + Fn-2 + Fn-3 점화식을 사용하면 된다.
  - N의 범위가 매우 크므로 분할 정복을 이용해야 한다.

Hard Problem

✓ 돌다리도 두드려보고 건너야 한다 (Hard)

- 이전의 두 문제와 비슷해 보이지만 사실 완전히 다른 문제이다
- 따라서 문제의 접근법도 매우 달라진다.
- Hint: 마법 슈즈를 가지고 원래 위치로 돌아오는 경우의 수는 2^(밟지 않은 돌의 개수) 이다.
- Hint 2: N의 범위가 1,000으로 줄어든 것을 이용하자.