

알튜비튜

동적 계획법

오늘은 '다이나믹 프로그래밍'이라고도 불리는 동적 계획법 알고리즘에 대해 배웁니다.
과거에 구한 해를 현재 해를 구할 때 활용하는 알고리즘이죠. 문제에 많이 나오는 굉장히 중요한 알고리즘 중 하나예요.

동적 계획법

- 특정 범위까지의 값을 구하기 위해 이전 범위의 값을 활용하여 효율적으로 값을 얻는 기법
- 이전 범위의 값을 저장(Memoization)함으로써 시간적, 공간적 효율 얻음

/<> 10870번 : 피보나치 수 5 – Bronze 2

문제

- $F(n) = F(n-1) + F(n-2)$ ($n \geq 2$)
- n 번째 피보나치 수를 구하는 문제

제한 사항

- 입력 범위는 $0 \leq n \leq 20$

/<> 10870번 : 피보나치 수 5 - Bronze 2

문제

- $F(n) = F(n-1) + F(n-2) \ (n \geq 2)$
 - n 번째 피보나치 수를 구하는 문제
- n 부터 시작하면 계속 전 단계 함수를 호출

제한 사항

- 입력 범위는 $0 \leq n \leq 20$

재귀함수로 풀면 안되나?

피보나치 수 5

- $F(n) = F(n-1) + F(n-2)$ ($n \geq 2$)
- n 번째 피보나치 수를 구하는 문제
- n 의 범위 ≤ 20

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

✓ 완전 가능!

피보나치 수 7

- $F(n) = F(n-1) + F(n-2)$ ($n \geq 2$)
- n 번째 피보나치 수를 구하는 문제
- n 의 범위 $\leq 1,000,000$

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

✓ 완전 가능?

피보나치 수 7

- $F(n) = F(n-1) + F(n-2)$ ($n \geq 2$)
- n 번째 피보나치 수를 구하는 문제
- n 의 범위 $\leq 1,000,000$

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

✓ 절대 불가능

→ 재귀로 풀기엔 n 의 범위가 커서 시간초과가 난다

재귀함수는

- $n = 4$

[함수 호출 수]

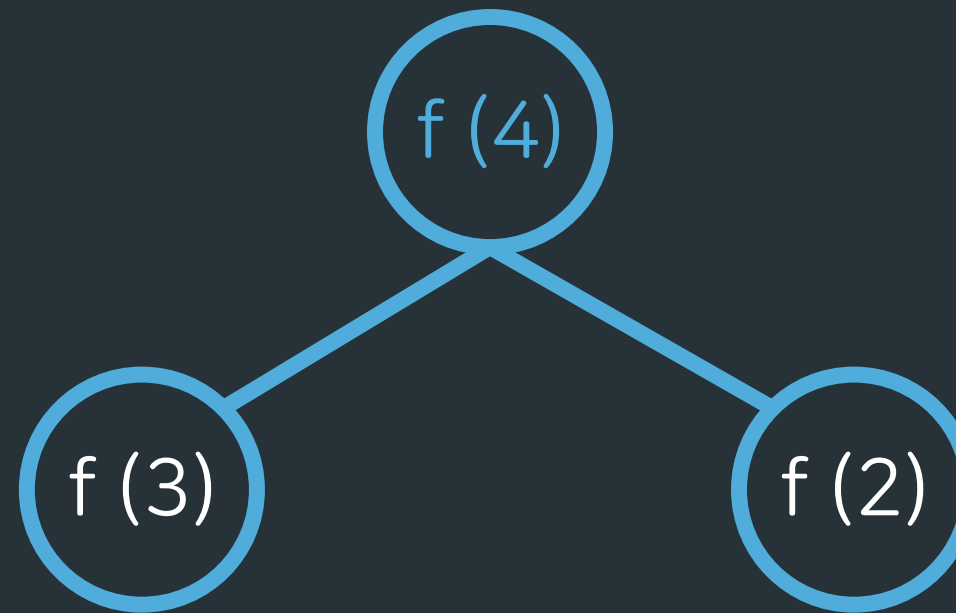
$f(4): 1$

$f(3): 1$

$f(2): 1$

$f(1): 0$

$f(0): 0$



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


재귀함수는

- $n = 4$

[함수 호출 수]

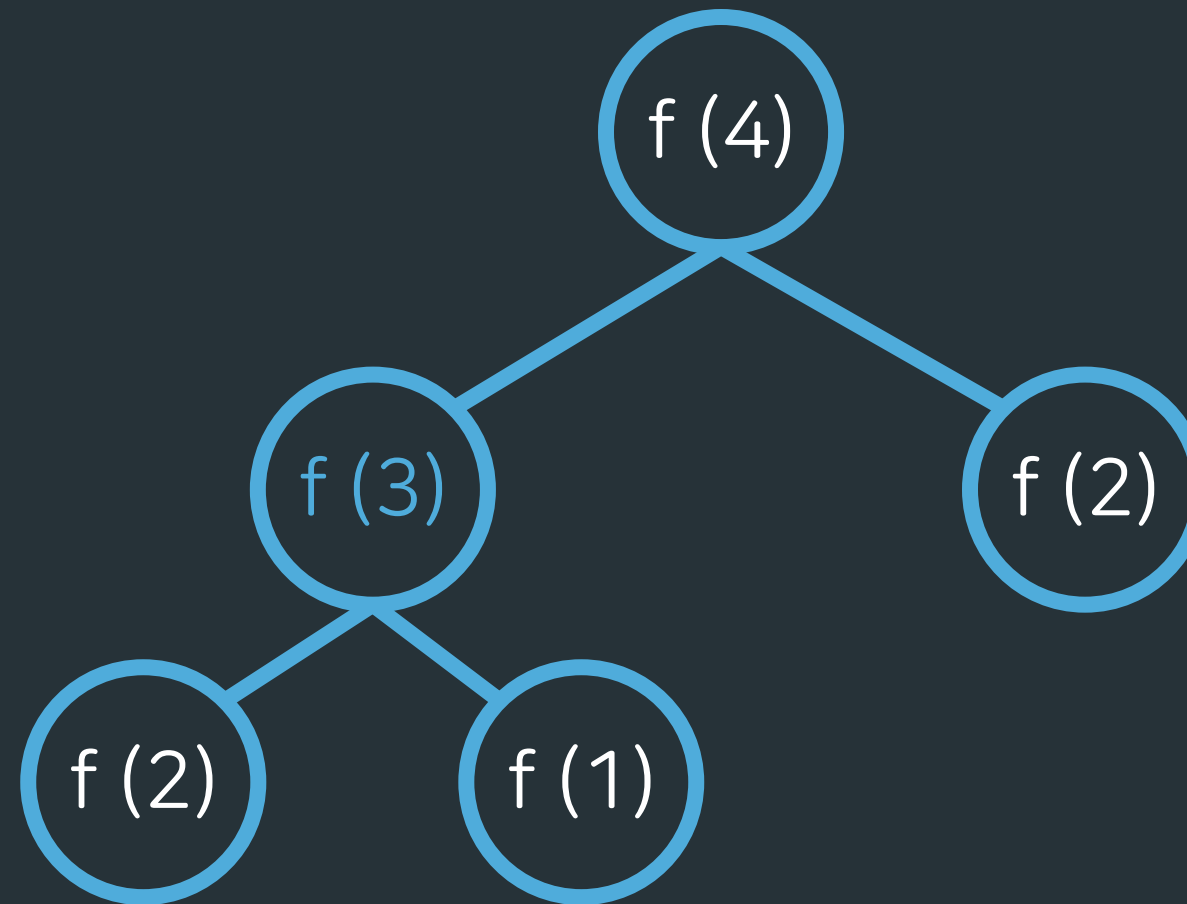
$f(4)$: 1

$f(3)$: 1

$f(2)$: 2

$f(1)$: 1

$f(0)$: 0



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

재귀함수는

● $n = 4$

[함수 호출 수]

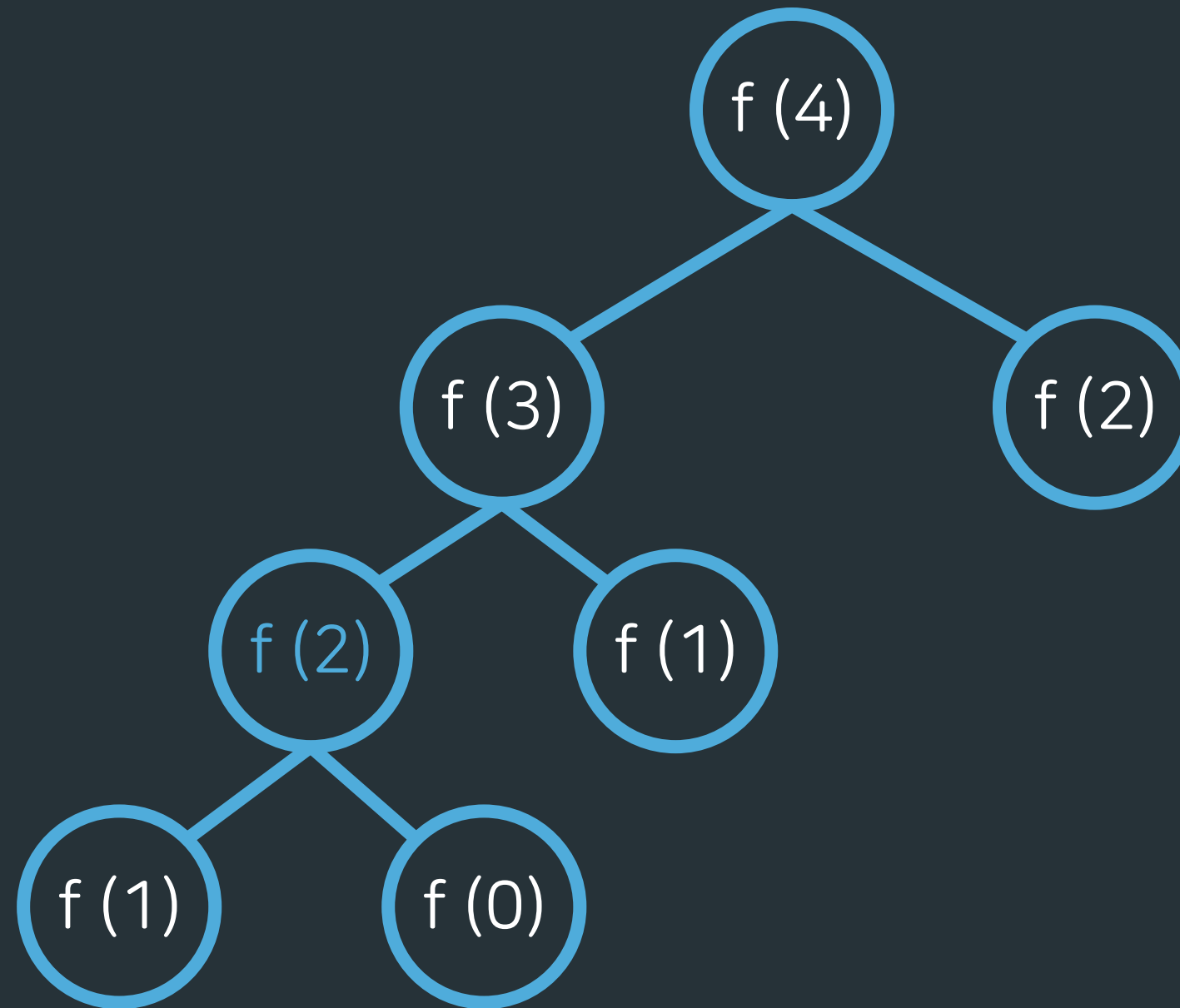
$f(4)$: 1

$f(3)$: 1

$f(2)$: 2

$f(1)$: 2

$f(0)$: 1



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

재귀함수는

● $n = 4$

[함수 호출 수]

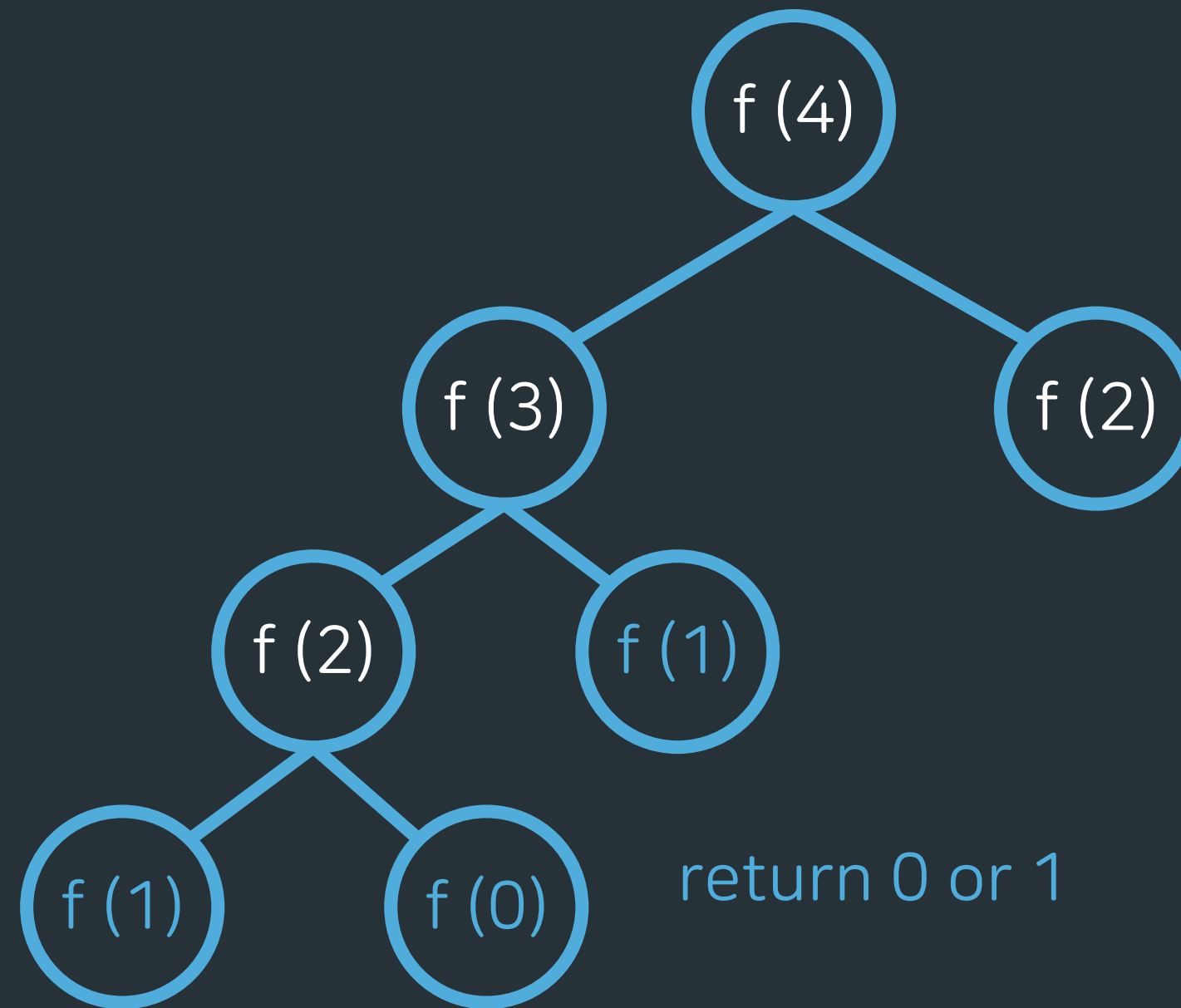
$f(4)$: 1

$f(3)$: 1

$f(2)$: 2

$f(1)$: 2

$f(0)$: 1



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

재귀함수는

● $n = 4$

[함수 호출 수]

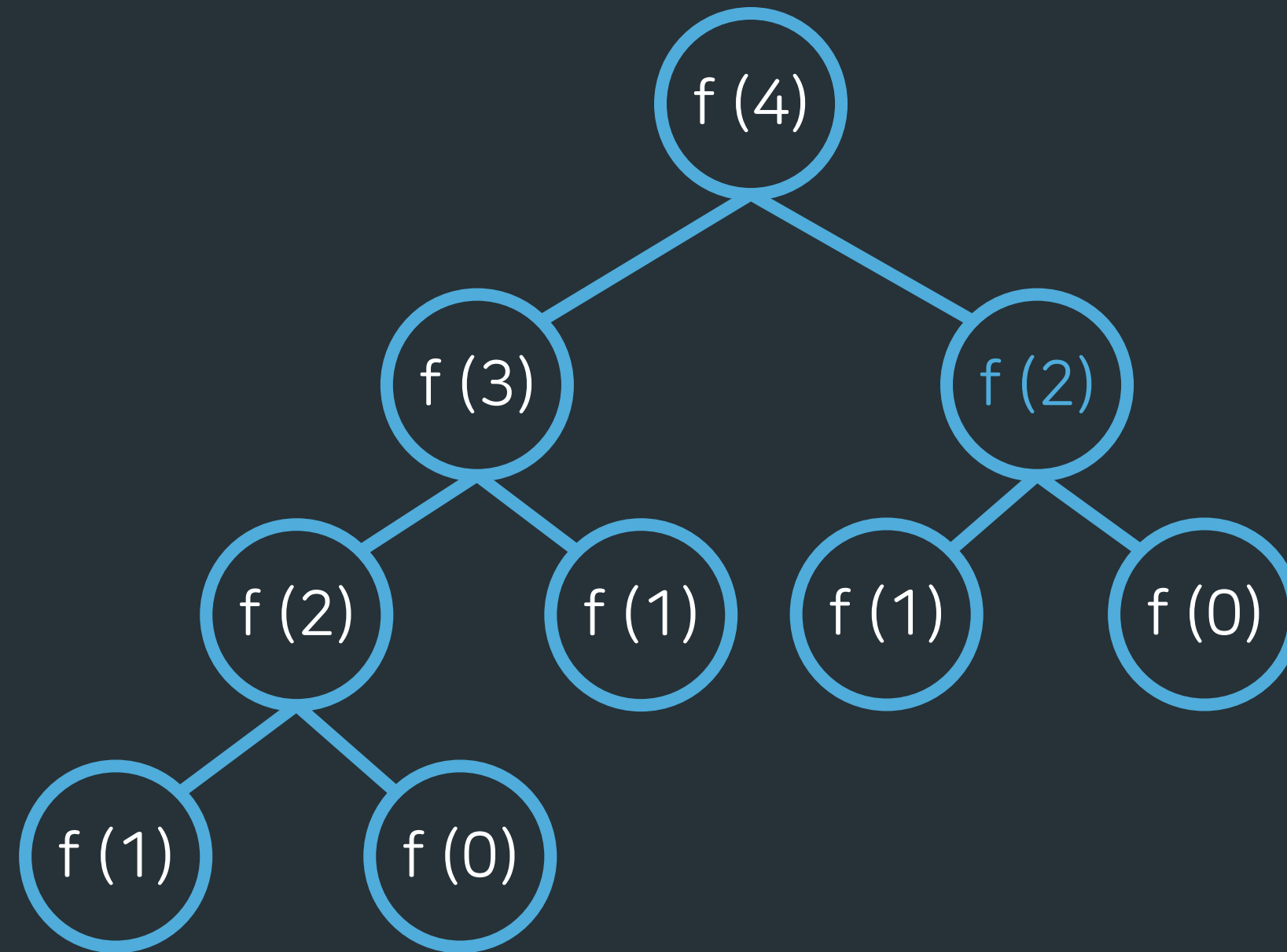
$f(4)$: 1

$f(3)$: 1

$f(2)$: 2

$f(1)$: 3

$f(0)$: 2



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

재귀함수는

● $n = 4$

[함수 호출 수]

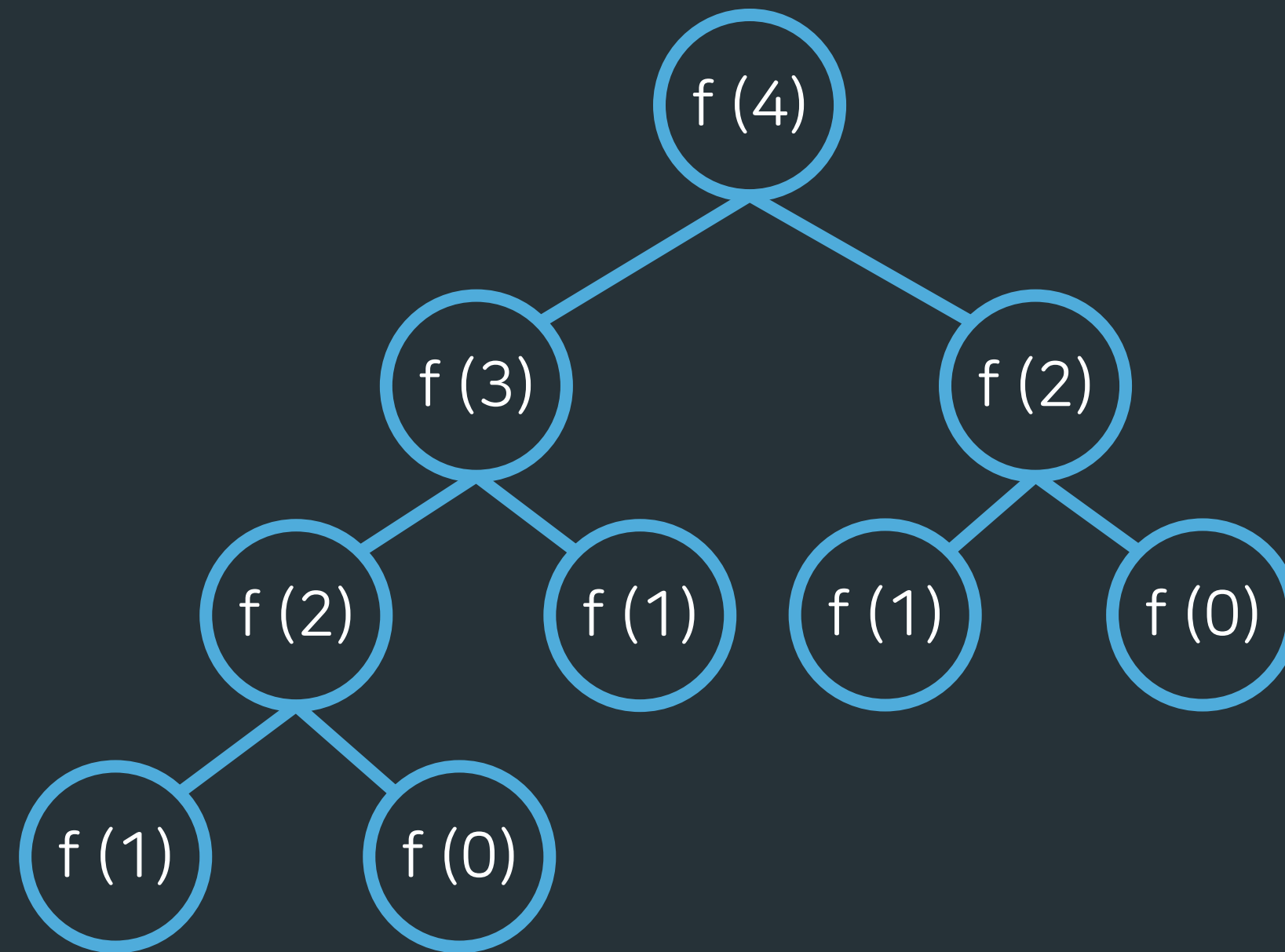
$f(4)$: 1

$f(3)$: 1

$f(2)$: 2

$f(1)$: 3

$f(0)$: 2



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

→ 같은 함수를 여러 번 호출하는 경우가 많다!
→ 즉, 한 번 계산한 값을 또 계산하게 됨

재귀함수는

- 당장 $n = 20$ 이어도..

[함수 호출 수]

$f(0)$: 4181

$f(1)$: 6765

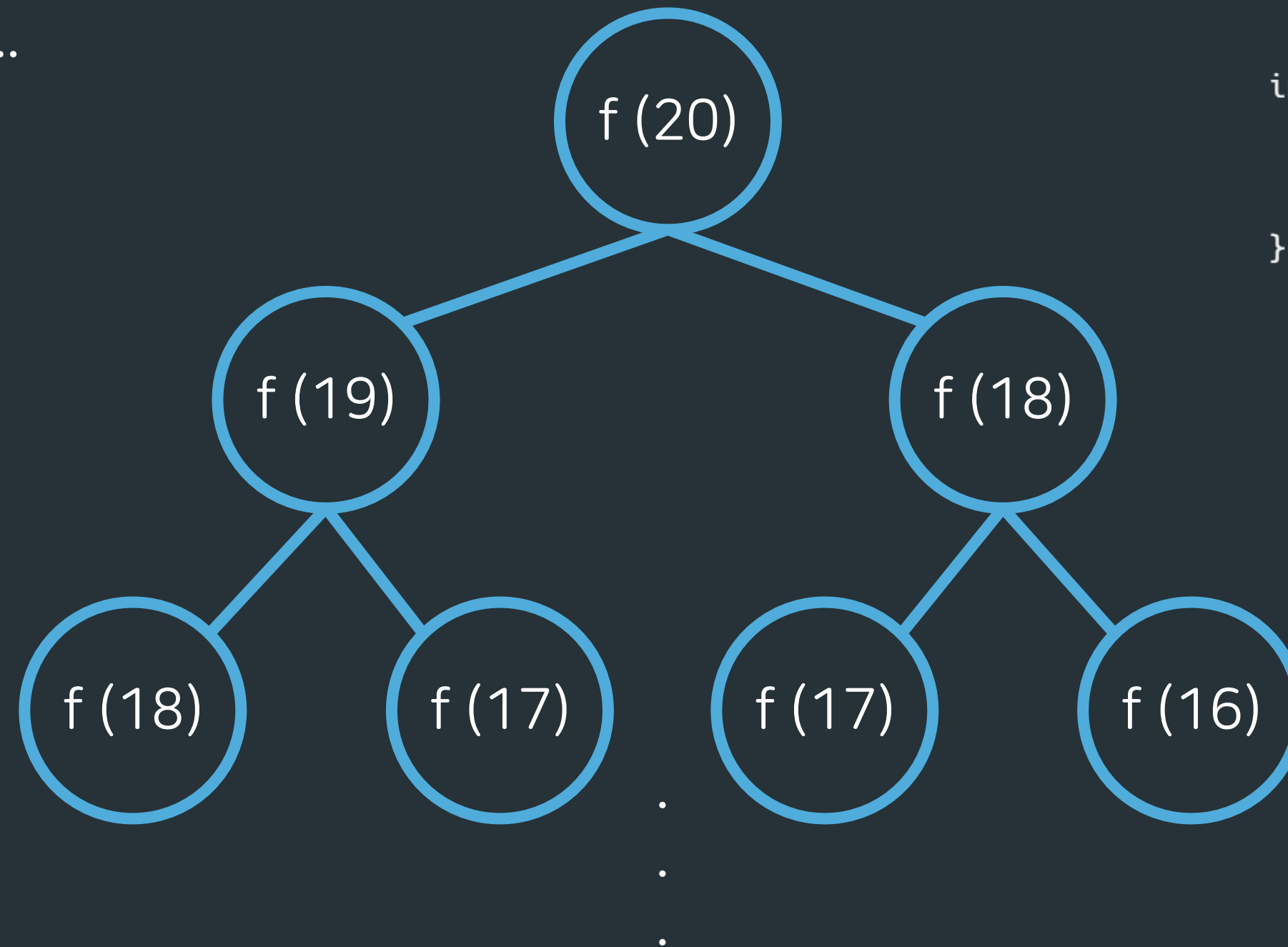
$f(2)$: 4181

$f(3)$: 2584

$f(4)$: 1597

·
·
·

*실제 값입니다



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

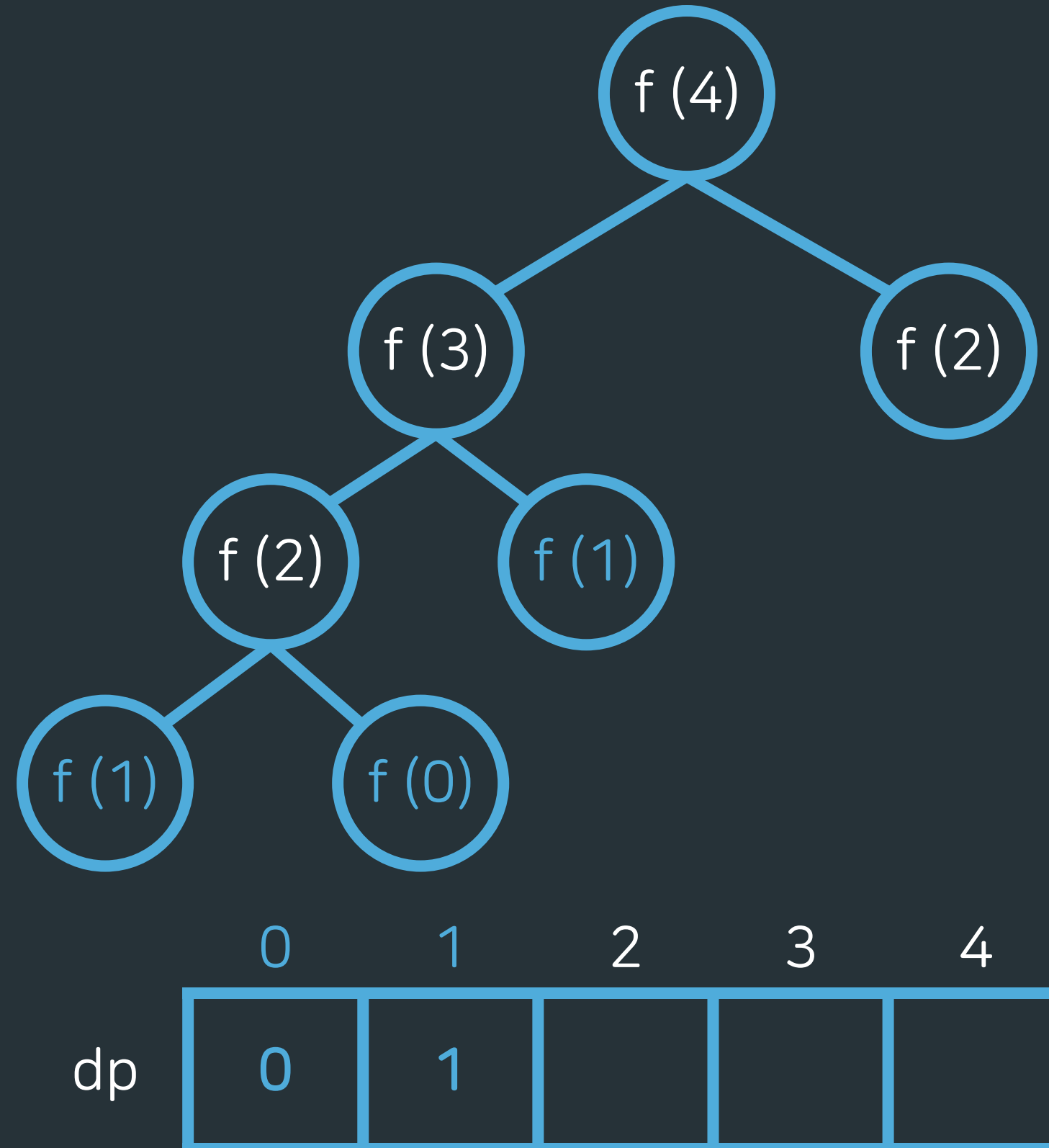
- N 이 커지면? 함수 호출이 훨씬 많이 일어남
- 그렇다면, 이미 구한 답을 또 계산할 필요가 있을까?

Memoization

- 이전에 구해둔 값을 저장해서 중복 계산을 방지
- 이전 범위의 답을 구하면, 바로 배열에 저장해 놓자!
- 시간과 공간면에서 모두 효율적!

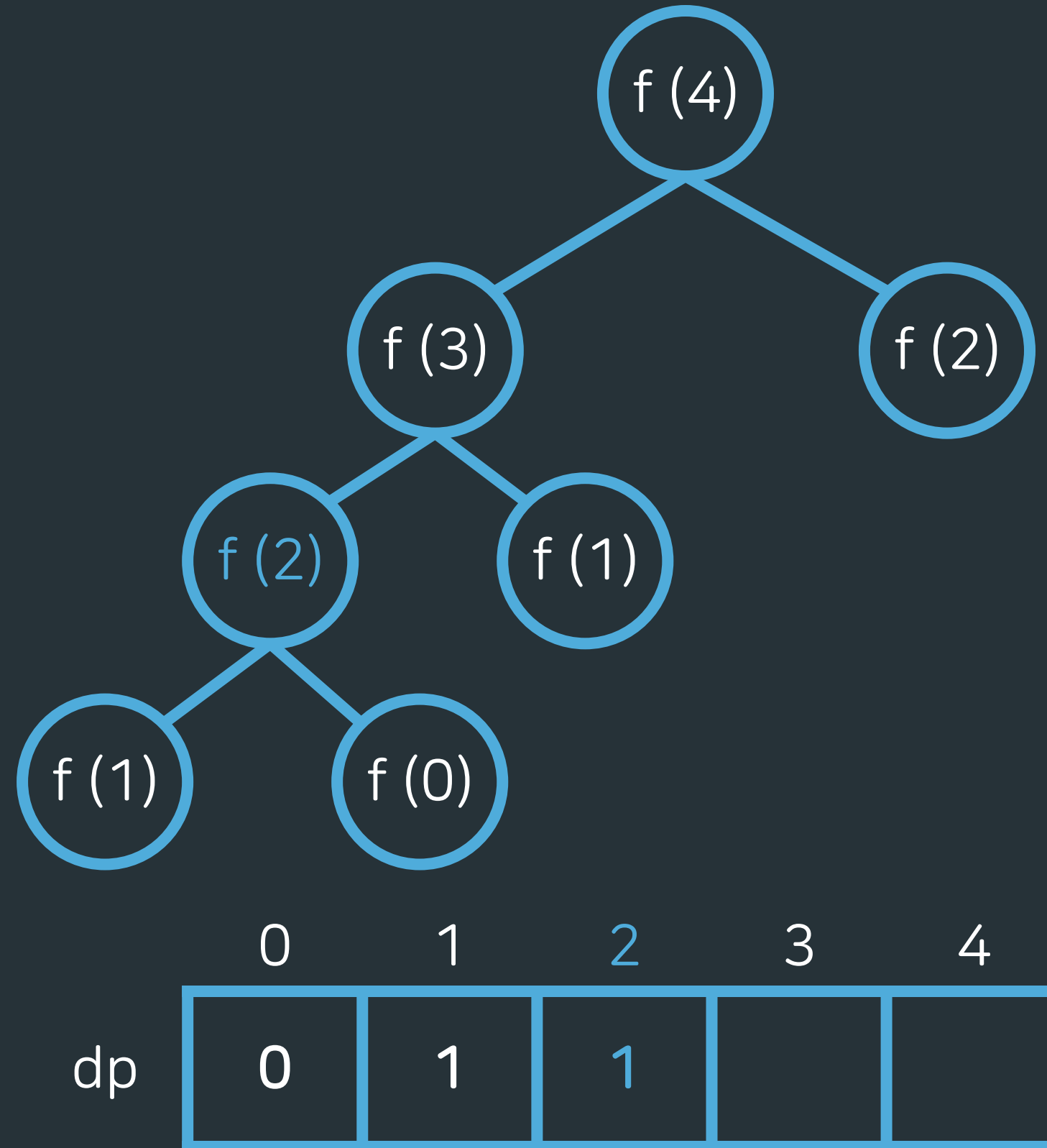
피보나치 수 문제에 적용하면

- $n = 4$



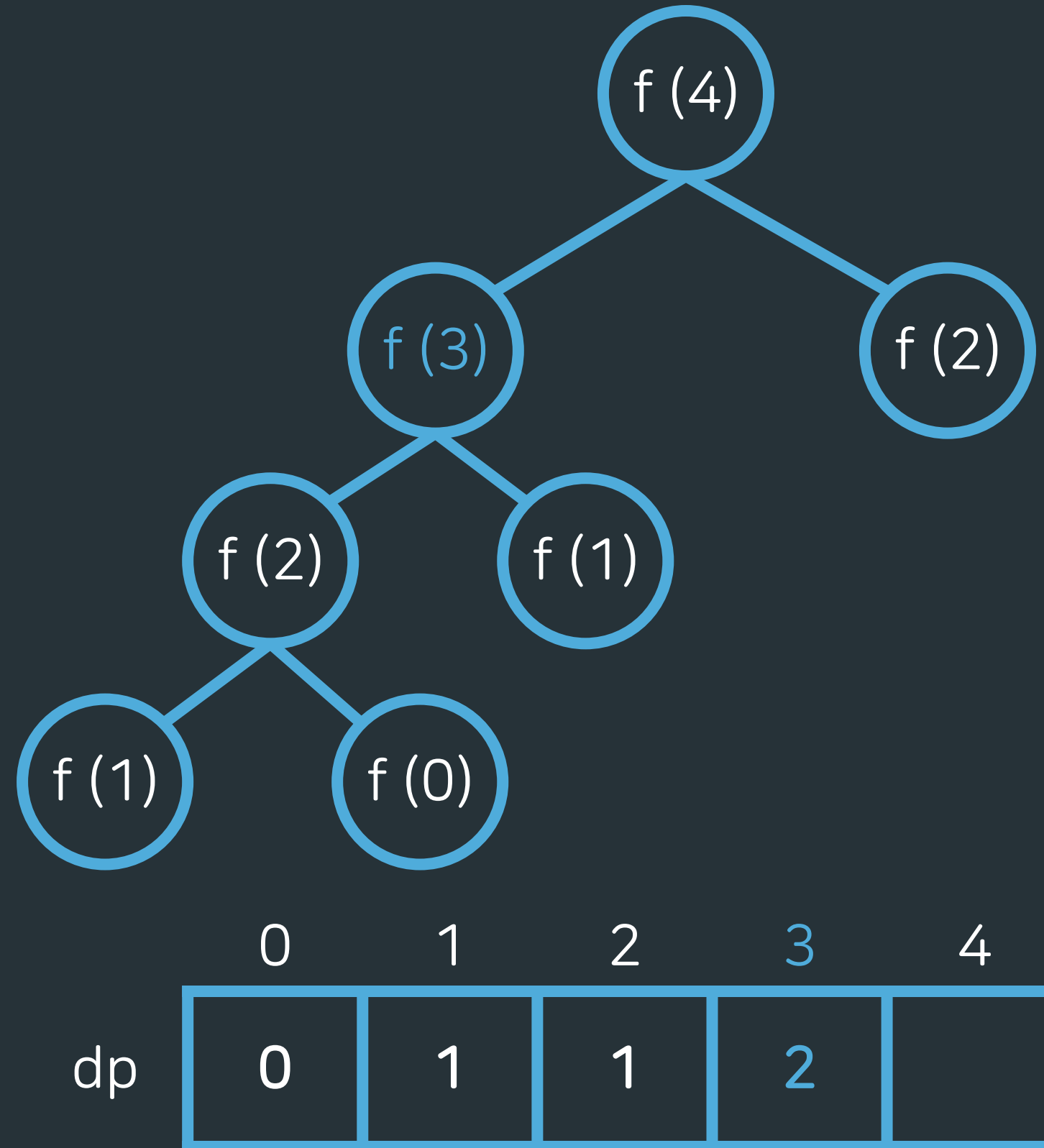
피보나치 수 문제에 적용하면

● $n = 4$



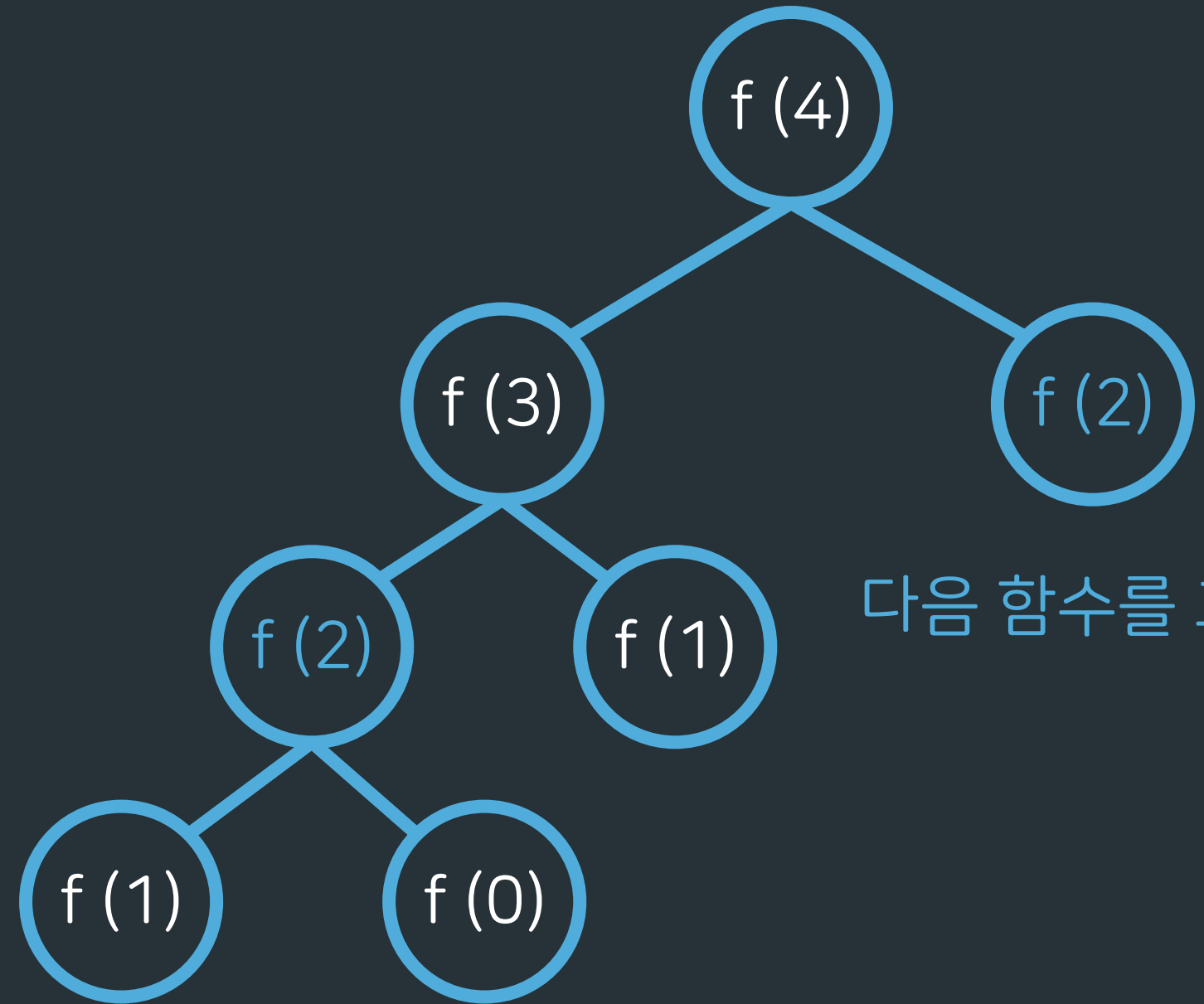
피보나치 수 문제에 적용하면

- $n = 4$



피보나치 수 문제에 적용하면

- $n = 4$

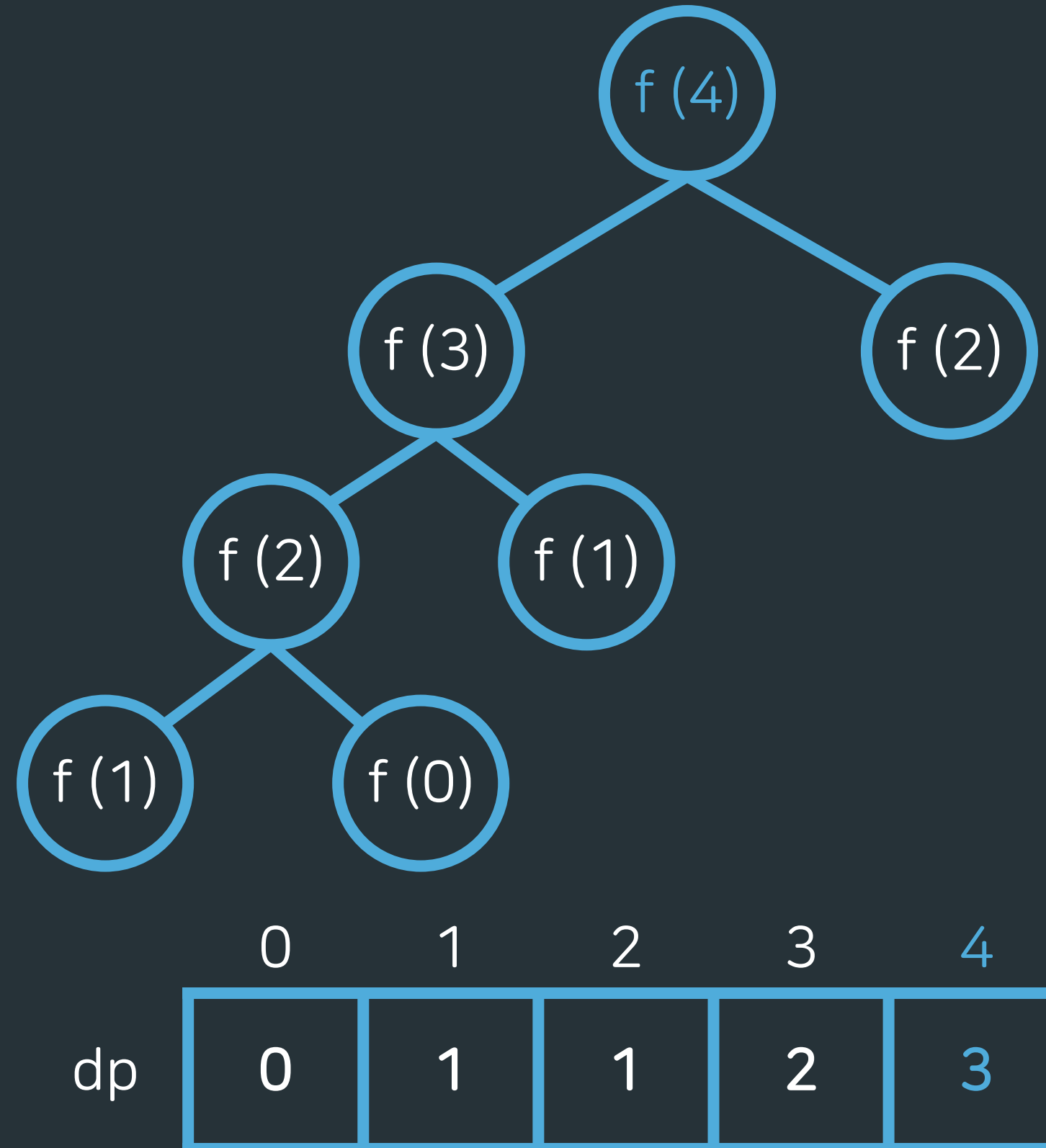


다음 함수를 호출할 필요가 없어짐!

	0	1	2	3	4
dp	0	1	1	2	

피보나치 수 문제에 적용하면

- $n = 4$



단순 재귀함수

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

- 보통 $n \leq 20$ 까지만 가능
- 그 이상은 시간초과



동적 계획법

```
int f(int n) {  
    if (n <= 1)  
        return n;  
    if (dp[n]) //dp[n]의 값이 존재한다면  
        return dp[n]; //함수 호출x 이미 계산한 값 리턴  
    return dp[n] = f(n - 1) + f(n - 2);  
}
```

- n 의 범위 클 때 활용
- 훨씬 효율적인 풀이

다르게 구현할 수도 있어요

동적 계획법

```
int f(int n) {  
    if (n <= 1)  
        return n;  
    if (dp[n]) //dp[n]의 값이 존재한다면  
        return dp[n]; //함수 호출x 이미 계산한 값 리턴  
    return dp[n] = f(n - 1) + f(n - 2);  
}
```



0번 인덱스부터 시작해서 미리
배열에 이전 범위의 답을 저장하면
어떨까?

- Top-down 방식 (n부터)
- 구하려 하는 문제를 작은 문제로 호출하며 탐색
- 재귀함수를 활용

Top-down vs Bottom-up

```
int f(int n) {  
    if (n <= 1)  
        return n;  
    if (dp[n]) //dp[n]의 값이 존재한다면  
        return dp[n]; //함수 호출x 이미 계산한 값 리턴  
    return dp[n] = f(n - 1) + f(n - 2);  
}
```

37052 KB

32 ms



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

5928 KB

4 ms

- Top-down 방식 (n부터)
- 구하려 하는 문제를 작은 문제로 호출하며 탐색
- 재귀함수를 활용

- Bottom-up 방식 (0부터)
- 이미 알고 있는 작은 문제부터 원하는 문제까지 탐색
- Top-down 방식보다 속도 빠름!

어떨 때 동적 계획법을 적용하지?

동적 계획법

- 주어진 문제를 부분 문제로 나누었을 때, 부분 문제의 답을 통해 주어진 문제의 답을 도출할 수 있을 때
- 부분 문제의 답을 여러 번 구해야 할 때
- 즉, 한 번 계산한 값을 다시 사용해야 할 때

점화식

- 인접한 항들 사이의 관계식
- 동적 계획법 문제를 풀 때는, 점화식을 미리 세우고 풀면 좋다!
- 이전 값들을 통해 DP(현재)를 정의하자

(ex) 피보나치 수 문제: $DP[i] = DP[i - 1] + DP[i - 2]$

/<> 1932번 : 정수 삼각형 - Silver 1

문제

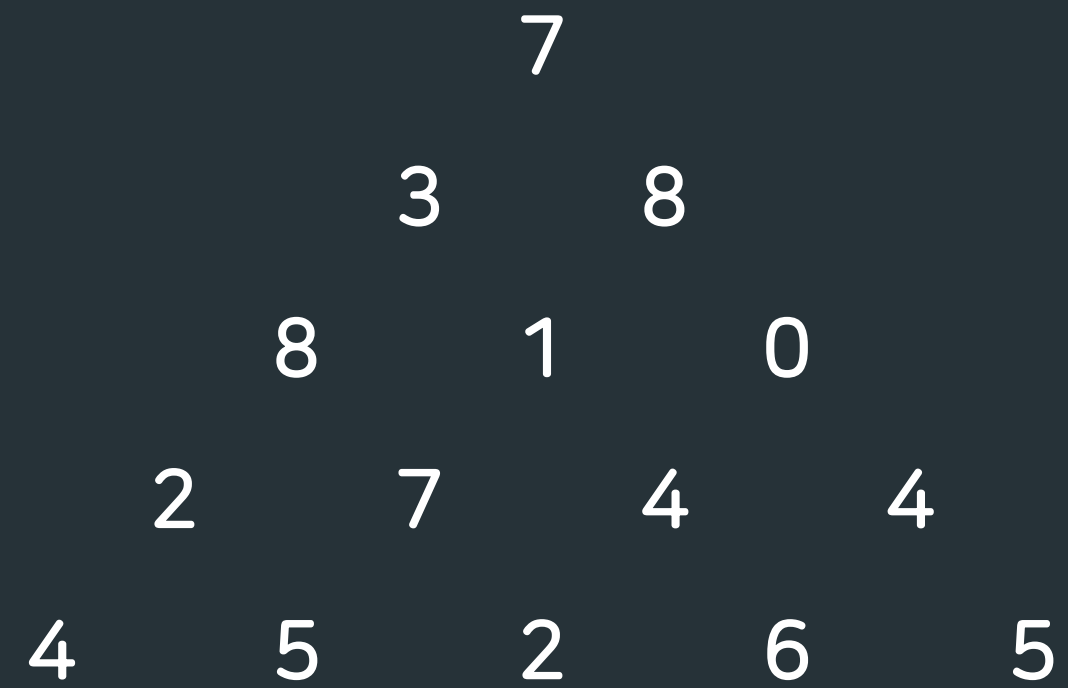
- 정수 삼각형이 주어졌을 때, 맨 위층부터 시작해서 아래층으로 내려오면서 이제까지 선택된 수의 합이 최대가 되는 경로를 구해라
- 현재 층에서 대각선 왼쪽 또는 대각선 오른쪽으로만 이동 가능

제한 사항

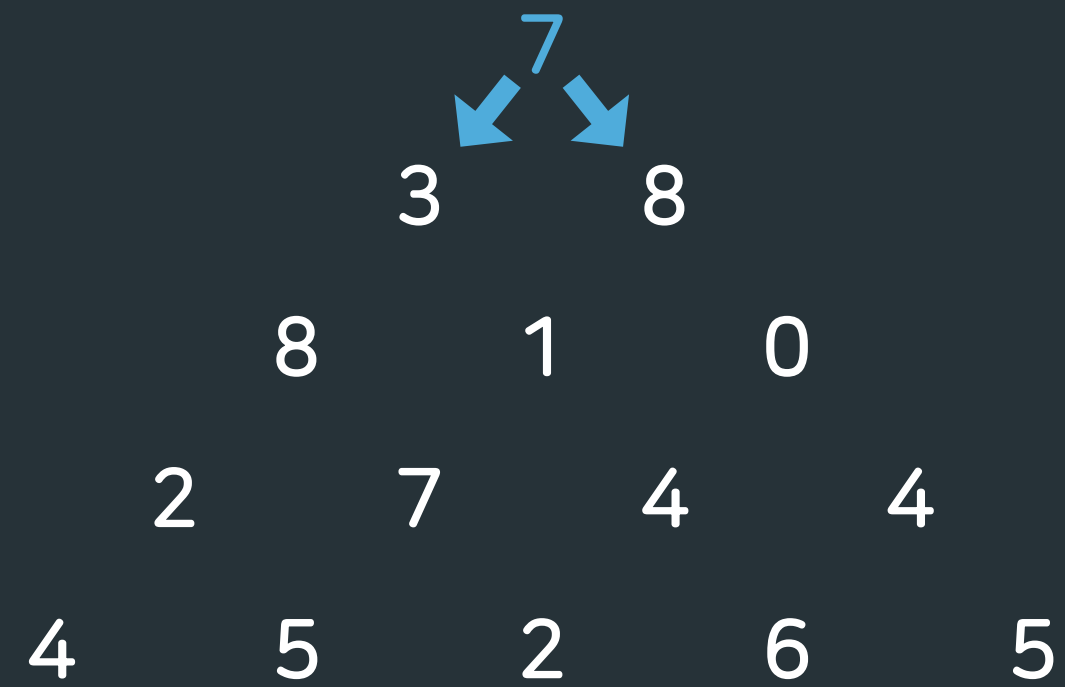
- 삼각형 크기 ≤ 500
- 삼각형 이루는 정수 $0 \sim 9,999$

→ 피보나치 수 문제랑 조금 비슷해 보이지 않나요?

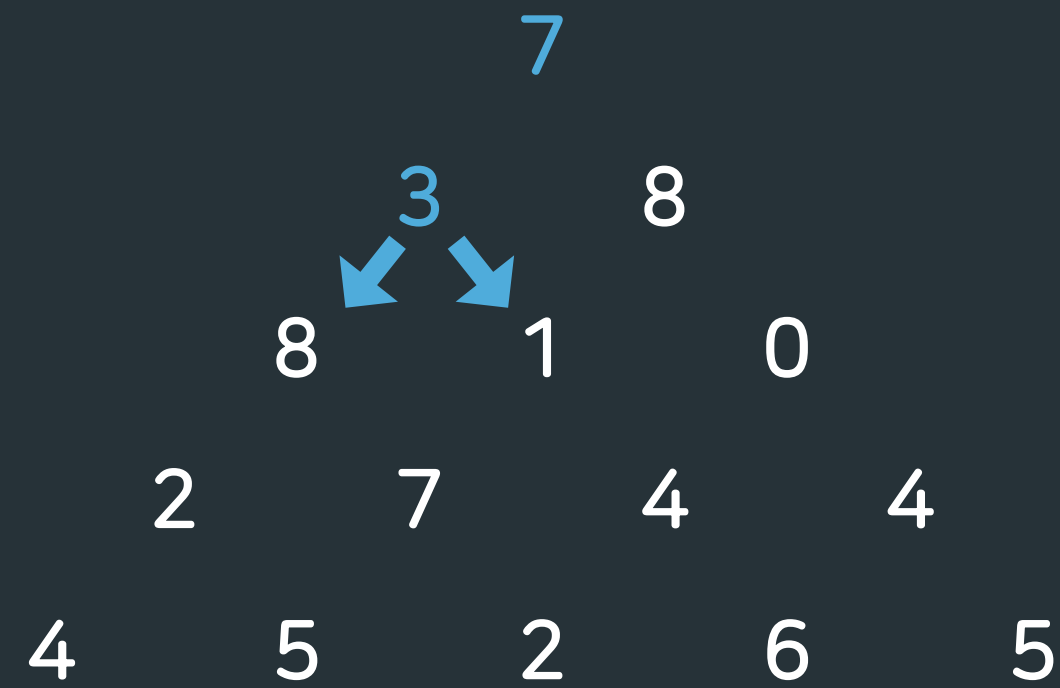
- 크기가 5인 정수 삼각형



- 크기가 5인 정수 삼각형



- 크기가 5인 정수 삼각형



- 크기가 5인 정수 삼각형



- 크기가 5인 정수 삼각형



재귀함수?

- 크기가 5인 정수 삼각형

제한 사항

- 삼각형 크기 ≤ 500



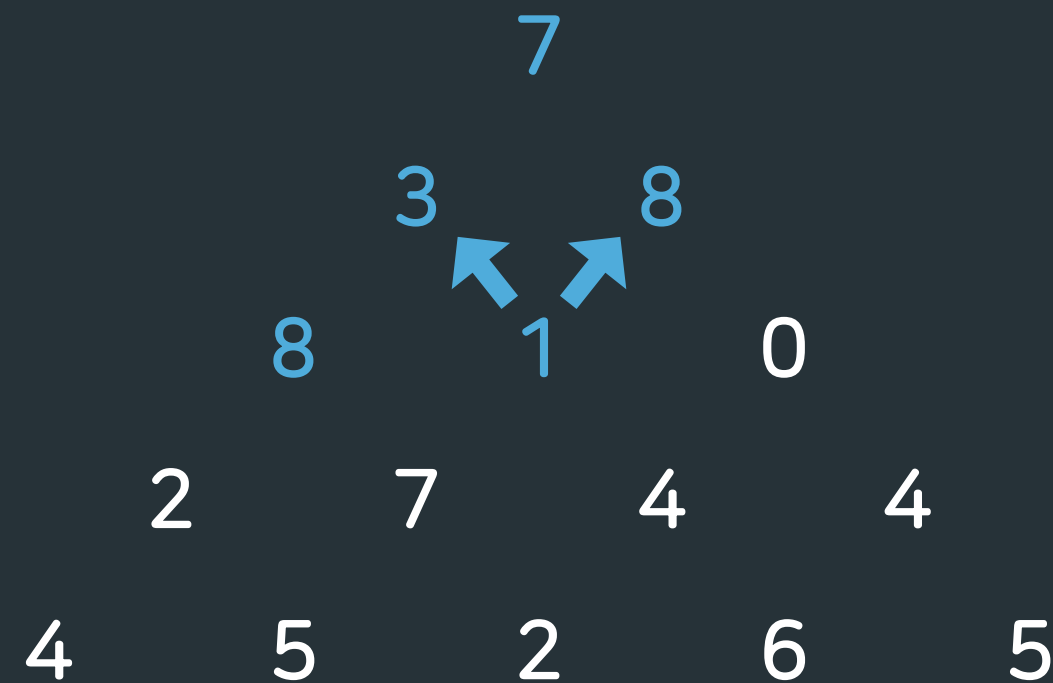
Bottom-up 방식의 DP!

→ 현재 위치의 최대 경로값은 어떻게 알지?

- 크기가 5인 정수 삼각형

제한 사항

- 삼각형 크기 ≤ 500



Bottom-up 방식의 DP!

→ 문제에서 위에서부터 내려가는거라 했으므로, 현재 위치에서 왼쪽 위 대각선, 오른쪽 위 대각선 중 어느 경로를 택해야 최대 경로인지 구하자!

인덱스 어떻게..?

- 문제에선 이렇게 주어져요.

7				
3	8			
8	1	0		
2	7	4	4	
4	5	2	6	5

즉, 현재 인덱스를 (i, j) 라 하면

- 왼쪽 위 대각선: $(i-1, j-1)$
- 오른쪽 위 대각선: $(i-1, j)$

→ $(1, 1)$ 인덱스부터 현재 인덱스에서 최대 경로값을 배열에 저장하며 풀자!

→ 2차원 배열 필요

```
dp[1][1] = triangle[1][1];
for (int i = 2; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        dp[i][j] = max(dp[i - 1][j - 1], dp[i - 1][j]) + triangle[i][j];
    }
}
```

물론 다른 방법도 가능해요

Bottom - up

3980 KB

40 ms

```
dp[1][1] = triangle[1][1];
for (int i = 2; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        dp[i][j] = max(dp[i - 1][j - 1], dp[i - 1][j]) + triangle[i][j];
    }
}
```

Top - down

3980 KB

44 ms

```
int f(int row, int col) {
    if (row == 1)
        return triangle[1][1];
    if (row == 0 || col == 0) //정수 삼각형 값이 없는 곳
        return 0;
    if (dp[row][col] >= 0) //값이 이미 존재한다면
        return dp[row][col];
    return dp[row][col] = max(f(row - 1, col - 1), f(row - 1, col)) + triangle[row][col];
}
```

/<> 2579번 : 계단 오르기 - Silver 3

문제

- 계단은 한 번에 1칸 or 2칸 오를 수 있음
- 연속된 세 개의 계단을 모두 밟으면 안됨 (시작점은 포함 x)
- 마지막 도착 계단은 반드시 밟음
- 각 칸의 점수가 주어질 때, 얻을 수 있는 점수의 최댓값 구하는 문제

제한 사항

- 계단 개수 ≤ 300
- 점수 $\leq 10,000$

→ 각 계단마다의 최댓값을 구한 후 저장하며 풀면 되지 않을까?

예제 입력

```
6
10
20
15
25
10
20
```

예제 출력

```
75
```

문제

- 계단은 한 번에 1칸 or 2칸 오를 수 있음
- 연속된 세 개의 계단을 모두 밟으면 안됨 (시작점은 포함 x)
- 마지막 도착 계단은 반드시 밟음
- 각 칸의 점수(score)가 주어질 때, 얻을 수 있는 점수의 최댓값 구하는 문제

접근

- DP에는 현재 계단까지의 점수의 최댓값 저장
- 현재 계단은 1칸 or 2칸 전 계단에서 온 것

→ $DP[i] = \text{MAX}(DP[i - 1], DP[i - 2]) + \text{score}[i]$?

문제

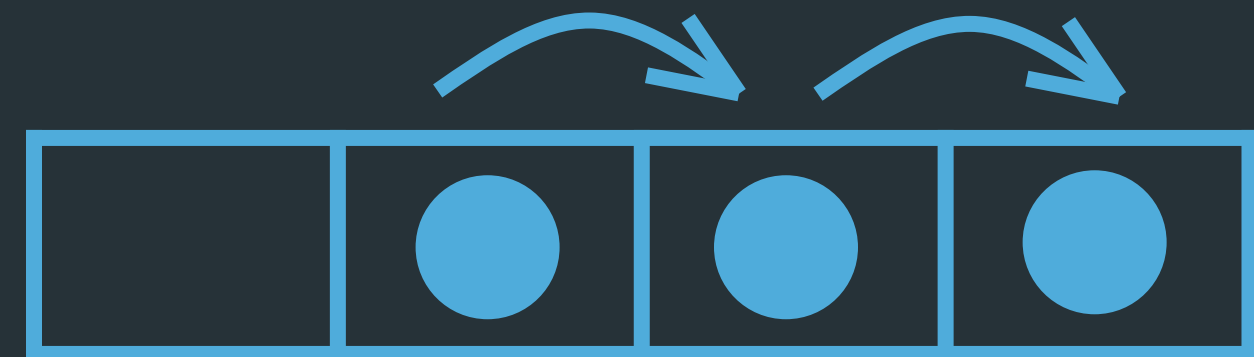
- 계단은 한 번에 1칸 or 2칸 오를 수 있음
- 연속된 세 개의 계단을 모두 밟으면 안됨 (시작점은 포함 x)
- 마지막 도착 계단은 반드시 밟음
- 각 칸의 점수(score)가 주어질 때, 얻을 수 있는 점수의 최댓값 구하는 문제

접근

- DP에는 현재 계단까지의 점수의 최댓값 저장
- 현재 계단은 1칸 or 2칸 전 계단에서 온 것

→ $DP[i] = \text{MAX}(DP[i - 1], DP[i - 2]) + \text{score}[i]$ (x)

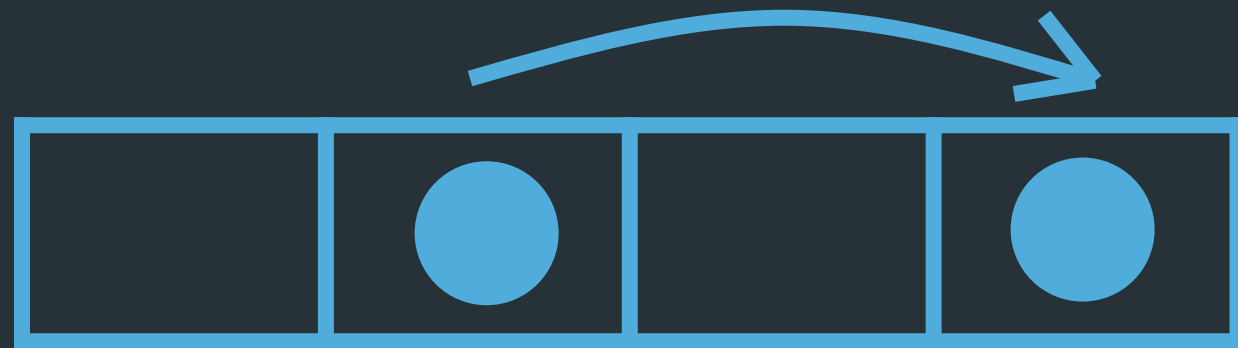
→ 이것만으론 연속 세 칸을 잡아낼 수 없음



연속 세 칸이므로 안됨!

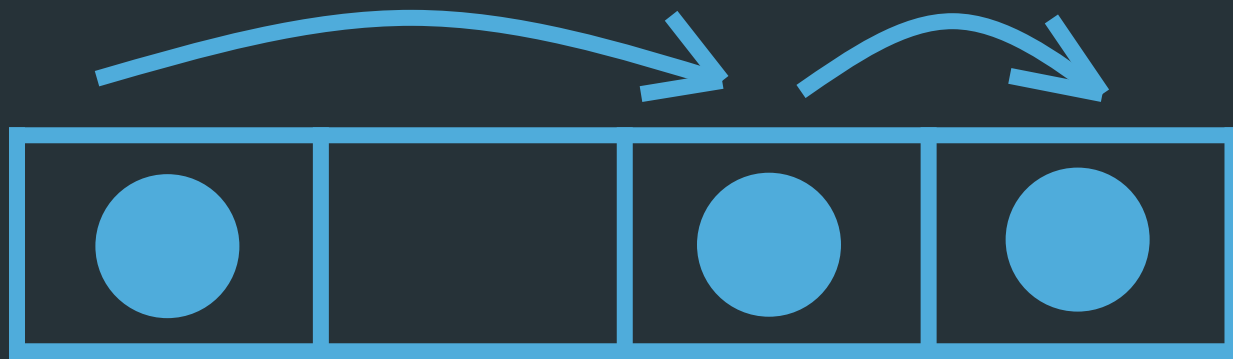
점화식을 세워보자

접근



- 두 칸 전에서 온 건 괜찮음
→ $DP[i - 2]$

접근



- 한 칸 전에서 온 값을 쓰고 싶다면, 3칸 전에서 2칸 이동 후 한 칸 전으로 온 경우 생각하면 됨!
→ $DP[i - 3] + \text{score}[i - 1]$

$$\therefore DP[i] = \text{MAX}(DP[i - 2], DP[i - 3] + \text{score}[i - 1]) + \text{score}[i]$$

/<> 11053번 : 가장 긴 증가하는 부분 수열 - Silver 2

문제

- 수열 A가 주어졌을 때, 가장 긴 증가하는 부분 수열의 길이를 구하는 문제

제한 사항

- 수열 A의 길이 범위는 $1 \leq \text{len}(A) \leq 1,000$

가장 긴 증가하는 부분 수열 (LIS)



[10, 20, 10, 30, 20, 50]

가장 긴 증가하는 부분 수열 (LIS)



[10, 20, 10, 30, 20, 50]

가장 긴 증가하는 부분 수열 (LIS)

접근


- 모든 부분 수열을 구해서 증가하는 부분 수열인지 검사하는 브루트 포스 접근

→ 시간 복잡도 $O(2^n)$ 이고, n 은 최대 1,000이므로 절대 불가능!

가장 긴 증가하는 부분 수열 (LIS)

접근

- 수열을 0번 인덱스부터 탐색하며 해당 인덱스로 끝나는 부분수열의 최대값을 계산해나가면 어떨까?



[10, 20, 10, 30, 20, 50]

가장 긴 증가하는 부분 수열 (LIS)

접근

- 수열을 0번 인덱스부터 탐색하며 해당 인덱스로 끝나는 “증가하는 부분수열”의 길이의 최댓값을 계산해나가면 어떨까?

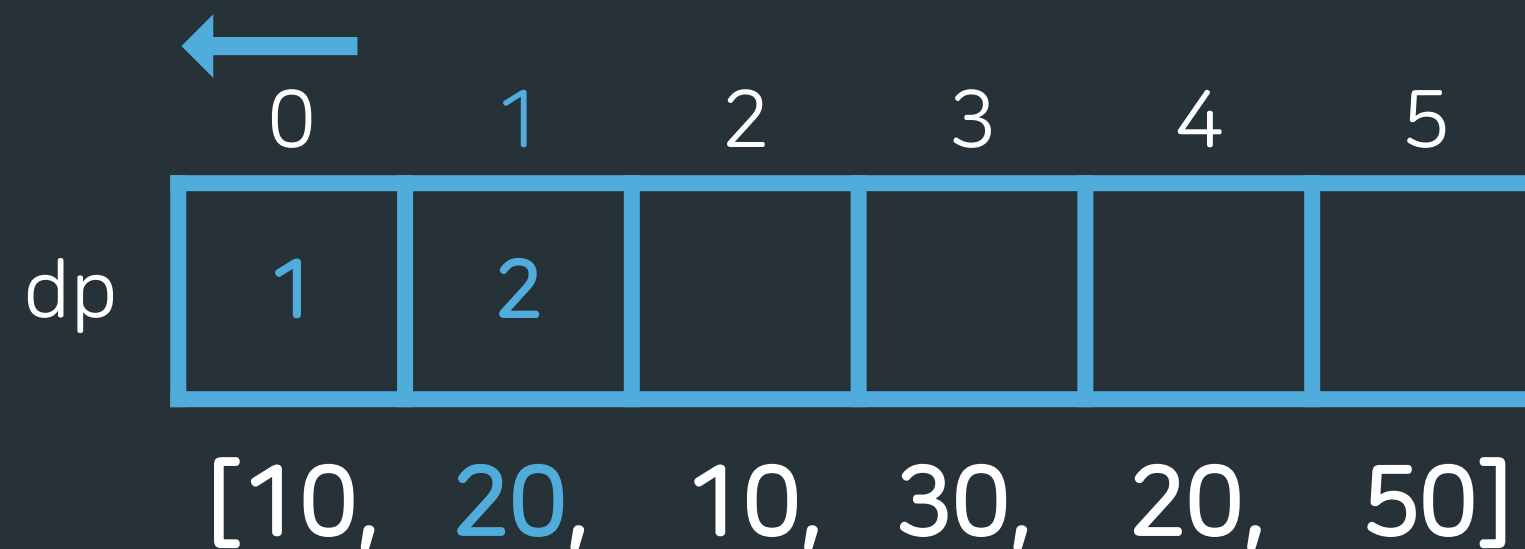
	0	1	2	3	4	5
dp	1	1	1	1	1	1
	[10,	20,	10,	30,	20,	50]

- 우선, 해당 인덱스의 수가 존재하므로
최소 길이는 1

가장 긴 증가하는 부분 수열 (LIS)

접근

- 수열을 0번 인덱스부터 탐색하며 해당 인덱스로 끝나는 “증가하는 부분수열”의 길이의 최댓값을 계산해나가면 어떨까?

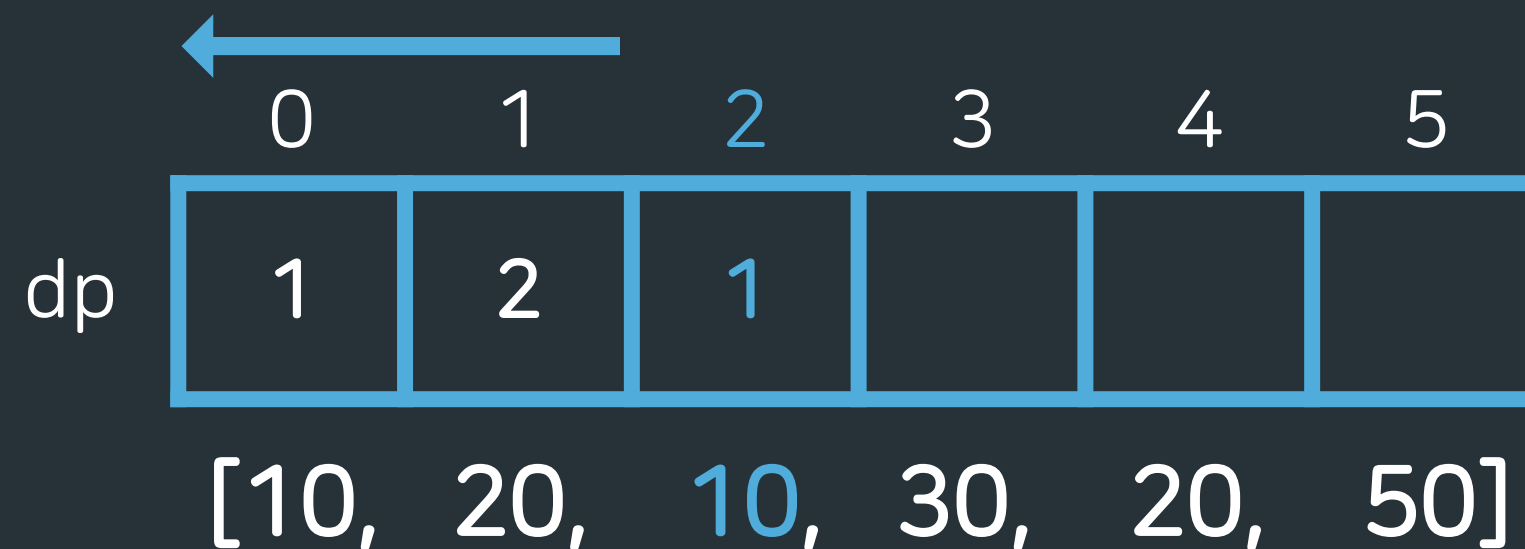


- “증가하는 부분수열”이므로 $10 < 20$ 검사 필요

가장 긴 증가하는 부분 수열 (LIS)

접근

- 수열을 0번 인덱스부터 탐색하며 해당 인덱스로 끝나는 “증가하는 부분수열”의 길이의 최댓값을 계산해나가면 어떨까?

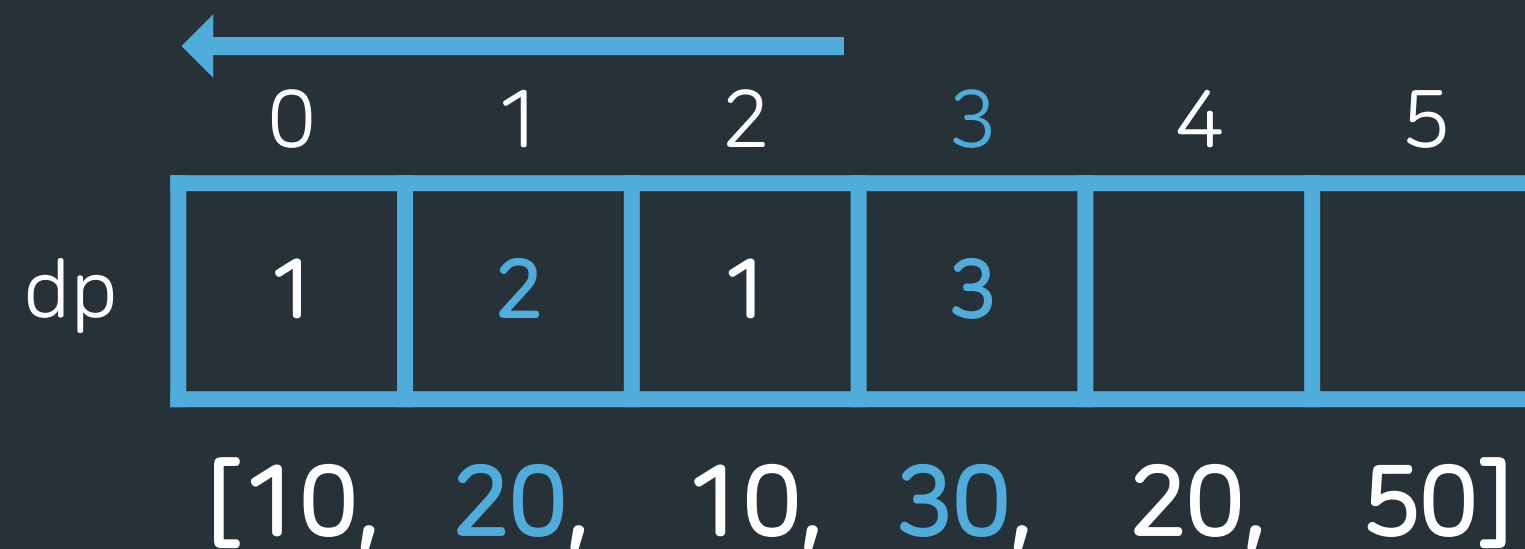


- 현재 수보다 작은 값이 없음 → 자기 자신 값 사용

가장 긴 증가하는 부분 수열 (LIS)

접근

- 수열을 0번 인덱스부터 탐색하며 해당 인덱스로 끝나는 “증가하는 부분수열”의 길이의 최댓값을 계산해나가면 어떨까?



- 현재 수의 값보다 작은 그 전 값들 중, 길이가 가장 긴 것(dp[i]값이 가장 큰 것)을 고르는 것이 핵심!
- 거기서 + 1을 해주면 현재 수로 끝나는 가장 긴 부분수열 길이!

가장 긴 증가하는 부분 수열 (LIS)

접근

- 수열을 0번 인덱스부터 탐색하며 해당 인덱스로 끝나는 “증가하는 부분수열”의 길이의 최댓값을 계산해나가면 어떨까?

	0	1	2	3	4	5
dp	1	2	1	3	2	4
	[10,	20,	10,	30,	20,	50]

$$\therefore DP[i] = \text{MAX}(\sum_{j=0}^{i-1} DP[j]) + 1 \text{ (단, } A[j] < A[i])$$

/<> 12865번 : 평범한 배낭 - Gold 5

문제

- 최대 무게(k)가 정해진 배낭에 물건을 넣는다.
- 각 물건은 무게(w)와 가치(v)가 있다.
- 배낭에 넣을 수 있는 물건들의 가치합의 최댓값을 구하는 문제

제한 사항

- 물품의 수 N ($1 \leq N \leq 100$)
- 배낭 무게 K ($1 \leq K \leq 100,000$)
- 물건 무게 W ($1 \leq W \leq 100,000$)
- 물건 가치 V ($0 \leq V \leq 1,000$)

예제 입력

```
4 7
6 13
4 8
3 6
5 12
```

예제 출력

```
14
```

Hint

1. 전 시간들에 배운 알고리즘으로 풀기엔 시간이 부족해보여요.
2. 부분 문제에 대한 정답을 어떻게 활용할 수 있을까요? 이 문제의 전체 정답은 최대 무게 K 일 때의 최대 가치합이죠. 그렇다면 부분 문제는 무엇일까요?

문제

- 최대 무게(k)가 정해진 배낭에 물건을 넣는다.
- 각 물건은 무게(w)와 가치(v)가 있다.
- 배낭에 넣을 수 있는 물건들의 가치합의 최댓값을 구하는 문제

접근

- 물품의 가능한 조합을 모두 구한 후, 무게가 K 이내이면서 가치합이 최대인 경우를 찾는 브루트 포스 접근
- $O(2^n)$ 이고, 물품의 수(n)가 최대 100이므로 절대 불가능!

문제

- 최대 무게(k)가 정해진 배낭에 물건을 넣는다.
- 각 물건은 무게(w)와 가치(v)가 있다.
- 배낭에 넣을 수 있는 물건들의 가치합의 최댓값을 구하는 문제

접근

- 물품의 가능한 조합을 구하는데, 중간에 무게가 K 를 초과하는 경우를 모두 쳐내며 가치합이 최대인 경우를 찾는 백트래킹 접근
- 웬지 가능해 보이지만, 이 풀이도 최악의 경우 K 를 초과하는 경우가 없으면 결국 브루트 포스와 동일. 즉, 불가능

문제

- 최대 무게(k)가 정해진 배낭에 물건을 넣는다.
- 각 물건은 무게(w)와 가치(v)가 있다.
- 배낭에 넣을 수 있는 물건들의 가치합의 최댓값을 구하는 문제

접근

- 오늘 배운 동적 계획법을 활용해 보자. 이전에 구한 답을 활용?
 - K 이전의 무게들에 대한 정답(가치합의 최댓값)을 저장하며 풀면 어떨까?
 - 무게를 인덱스로!

K이전 무게들에 대한 정답은 어떻게 계산?


접근

- K 까지의 무게를 인덱스로 나타냄
- 현재 물품을 배낭에 넣는 경우 or 안 넣는 경우 중 최대값을 저장하자
- 배낭에 넣으려면?
 - 현재 물품 무게만큼 배낭에 추가되는 것! 그런데 현재 인덱스가 배낭의 최대 무게인데?
 - $[\text{현재 배낭 무게} - \text{물품 무게}]$ 인 배낭 무게에서의 최대 가치값 + 현재 물품 가치값
- 배낭에 안 넣는 경우는?
 - 현재 배낭 무게에 저장된 정답을 그대로 사용하면 됨!

점화식을 세워봅시다

- $n = 4$
- $k = 7$
- product(물품) $w = 6, v = 13$
- 현재 물품을 배낭에 넣는 경우 or 안 넣는 경우 중 최댓값을 저장하자

	0	1	2	3	4	5	6	7
dp[1]	0	0	0	0	0	0	13	13



- 배낭에 넣는 경우: (현재 배낭 무게 - 물품 무게)를
인덱스로 가지는 값 + 물품 가치 $\rightarrow dp[0][0] + 13$
- 배낭에 안 넣는 경우: 0

점화식을 세워봅시다

- $n = 4$
- $k = 7$
- product(물품) $w = 4, v = 8$
- 현재 물품을 배낭에 넣는 경우 or 안 넣는 경우 중 최댓값을 저장하자

	0	1	2	3	4	5	6	7
dp[1]	0	0	0	0	0	0	13	13
dp[2]	0	0	0	0	8	8	13	13




● 배낭에 넣는 경우: $dp[1][6 - 4] + 8 = 8$

● 배낭에 안 넣는 경우: $dp[1][6] = 13$

점화식을 세워봅시다

- $n = 4$
- $k = 7$
- product(물품) $w = 3, v = 6$
- 현재 물품을 배낭에 넣는 경우 or 안 넣는 경우 중 최댓값을 저장하자

	0	1	2	3	4	5	6	7
dp[1]	0	0	0	0	0	0	13	13
dp[2]	0	0	0	0	8	8	13	13
dp[3]	0	0	0	6	8	8	13	14



- 배낭에 넣는 경우: $dp[2][7 - 3] + 6 = 14$
- 배낭에 안 넣는 경우: $dp[2][7] = 13$

점화식을 세워봅시다

- $n = 4$
- $k = 7$
- product(물품) $w = 5, v = 12$
- 현재 물품을 배낭에 넣는 경우 or 안 넣는 경우 중 최댓값을 저장하자

	0	1	2	3	4	5	6	7
dp[1]	0	0	0	0	0	0	13	13
dp[2]	0	0	0	0	8	8	13	13
dp[3]	0	0	0	6	8	8	13	14
dp[4]	0	0	0	6	8	12	13	14

점화식

- 현재 물품을 배낭에 넣는 경우 or 안 넣는 경우 중 최댓값을 저장하자
- ∴ $DP[i][j] = \text{MAX}(DP[i-1][j - \text{product}[i].w] + \text{product}[i].v, DP[i-1][j])$
(단, $\text{product}[i].w \leq j$)

왜 2차원 DP?

- 1차원 DP로 하면 안 되는 이유는?
→ 그 전 물품까지의 정보만 사용해야 하기 때문

왜 2차원 DP?

- 1차원 DP로 하면 안 되는 이유는?
→ 그 전 물품까지의 정보만 사용해야 하기 때문
- 지금처럼 증가하면서 검사할 때 1차원 DP를 사용하게 되면?
→ 해당 물품을 또 사용하는 경우 생길 수 있음!
→ 따라서 2차원을 사용하며 각 물품을 행으로 구분해서 중복 방지

1차원 DP도 가능

- 어떻게 하면 1차원 DP로 풀이가 가능할까
- 해당 물품을 여러 번 사용하는 걸 방지하기 위해 2차원을 사용함
- 그렇다면.. 지금처럼 증가하는게 아니라 무게를 감소하며 계산하면 어떨까?

기존 증가 방식

dp



감소 방식

dp



뒤에서부터 채워지므로 중복 사용 방지

/<> 9251번 : LCS – Gold 5

문제

- LCS(최장 공통 부분 수열): 두 수열의 공통 부분 수열 중 가장 긴 것
- 두 문자열의 LCS의 길이를 구하는 문제

제한 사항

- 수열 최대 1000글자

ACAYKP

CAPCAK

ACAYKP

CAPCAK

어떻게 구하지?

→
ACAYKP
CAPCAK
↑

- C에 대한 A, AC, ACA, ACAY, ACAYK, ACAYKP의 LCS를 저장 (이전 부분의 답 저장)

→ 가능한 이전 문자열들의 조합에 대한 공통 부분 수열의 최대 길이를 저장하며 풀자

이전의 답을 어떻게 사용?

	A	C	A	Y	K	P
C	0	0	0	0	0	0
A	0	0	0	0	0	0
P	0	0	0	0	0	0
C	0	0	0	0	0	0
A	0	0	0	0	0	0
K	0	0	0	0	0	0

이전의 답을 어떻게 사용?

- 두 문자가 서로 다른 곳은
공통 부분 문자열에 속하지
않으므로 그 전의 길이
최댓값 그대로 가져옴

	→					
	A	C	A	Y	K	P
C	0	0	0	0	0	0
A	0	0	0	0	0	0
P	0	0	0	0	0	0
C	0	0	0	0	0	0
A	0	0	0	0	0	0
K	0	0	0	0	0	0

이전의 답을 어떻게 사용?

- 두 문자가 서로 같은 곳은
공통 부분 문자열에
추가되므로 해당 문자들이
포함되기 전의 길이 + 1

	→					
	A	C	A	Y	K	P
C	0	1	0	0	0	0
A	0	0	0	0	0	0
P	0	0	0	0	0	0
C	0	0	0	0	0	0
A	0	0	0	0	0	0
K	0	0	0	0	0	0

문자가 서로 다를 때

- 두 문자가 서로 다른 곳은 공통 부분 문자열에 속하지 않으므로 그 전의 길이 최댓값 그대로 가져옴

→ 그 전은 어디? 위쪽 or 왼쪽
→ 둘 중 더 큰 쪽

	A	C	A	Y	K	P
C	0	1	1	1	1	1
A	1	1	0	0	0	0
P	0	0	0	0	0	0
C	0	0	0	0	0	0
A	0	0	0	0	0	0
K	0	0	0	0	0	0

(ex) C, AC 조합(위)을 가져오거나
CA, A 조합(왼쪽)을 가져옴

문자가 서로 같을 때

- 두 문자가 서로 같은 곳은
공통 부분 문자열에
추가되므로 해당 문자들이
포함되기 전의 길이 + 1

→ 그 전은 어디? 좌상향 대각선

		→					
		A	C	A	Y	K	P
C		0	1	1	1	1	1
A		1	1	2	0	0	0
P		0	0	0	0	0	0
C		0	0	0	0	0	0
A		0	0	0	0	0	0
K		0	0	0	0	0	0

(ex) C, AC 의 문자열에서
각각 A가 추가됨으로써
CA, ACA를 만듦

표를 채워서 익힌 뒤, 문제도 풀어봅시다.

	A	C	A	Y	K	P
C	0	1	1	1	1	1
A	1	1	2	2	2	2
P	1	1	2	2	2	3
C	1	2	2	2	2	3
A	1	2	3	3	3	3
K	1	2	3	3	4	4 ← 최종 LCS 길이

정리

- 이전의 답을 저장하고, 계속 사용하며 현재 답을 구하는 동적 계획법
- 입력 범위가 나름 커요 (보통 1,000 ~ 1,000,000) 이보다 더 크다면 그리디 고려
- 마지막 인덱스에서 내려가는 Top-down, 처음 인덱스부터 올라가는 Bottom-up 방식 존재
- 문제에 따라 1차원 혹은 2차원 테이블(DP 배열) 사용
- 점화식만 세우면 구현은 쉬움!
- LIS,ナップ색, LCS는 동적 계획법으로 푸는 대표적 문제 & 방식
- 따라서 세 유형의 풀이는 다른 동적 계획법 문제에서 많이 응용됨

이것도 알아보세요!

- Top-down 방식과 Bottom-up 방식 두 가지로 모두 풀어보고 시간을 비교해보아요

3문제 이상 선택

- /<> 11727번 : 2xn 타일링 2 – Silver 3
- /<> 9095번 : 1, 2, 3 더하기 – Silver 3
- /<> 11053번 : 가장 긴 증가하는 부분 수열 – Silver 2
- /<> 9084번 : 동전 – Gold 5
- /<> 9251번 : LCS – Gold 5
- /<> 11057번 : 오르막 수 – Silver 1

코드리뷰 0 마감 ~ 4월 4일 월요일 낮 12시

코드리뷰 X 마감 ~ 4월 4일 월요일 밤 12시 (4일에서 5일로 넘어가는 자정)

추가제출 마감 ~ 4월 5일 화요일 밤 12시 (5일에서 6일로 넘어가는 자정)