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 하나에요.

소개

동적 계획법

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

이런 문제가 있어요

#AL튜비튜

/<> 10870번 : 피보나치 수 5 – Bronze 2

문제

- F(n) = F(n-1) + F(n-2) (n >= 2)
- n번째 피보나치 수를 구하는 문제

제한 사항

- 입력 범위는 0 <= n <= 20

이런 문제가 있어요

/<> 10870번 : 피보나치 수 5 – Bronze 2

문제

- F(n) = F(n-1) + F(n-2) (n >= 2)
- n번째 피보나치 수를 구하는 문제

→ n부터 시작하면 계속 전 단계 함수를 호출

제한 사항

- 입력 범위는 0 <= n <= 20

재귀함수로 풀면 안되나?

피보나치 수 5

- F(n) = F(n-1) + F(n-2) (n >= 2)
- n번째 피보나치 수를 구하는 문제
- n의 범위 <= 20

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

✓ 완전 가능!

다른 문제도 있어요

피보나치 수 7

- $F(n) = F(n-1) + F(n-2)$ $(n >= 2)$
- n번째 피보나치 수를 구하는 문제
- n의 범위 <= 1,000,000

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

✓ 완전 가능?

재귀함수로..?

피보나치 수 7

- F(n) = F(n-1) + F(n-2) (n >= 2)
- n번째 피보나치 수를 구하는 문제
- n의 범위 <= 1,000,000

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

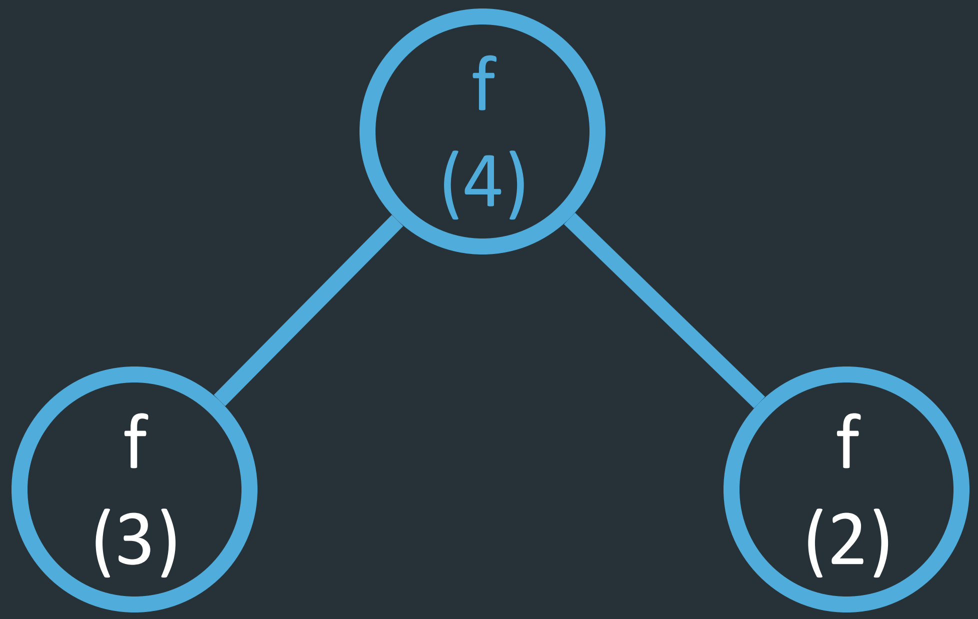
✓ 절대 불가능

→ 재귀로 풀기엔 n의 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

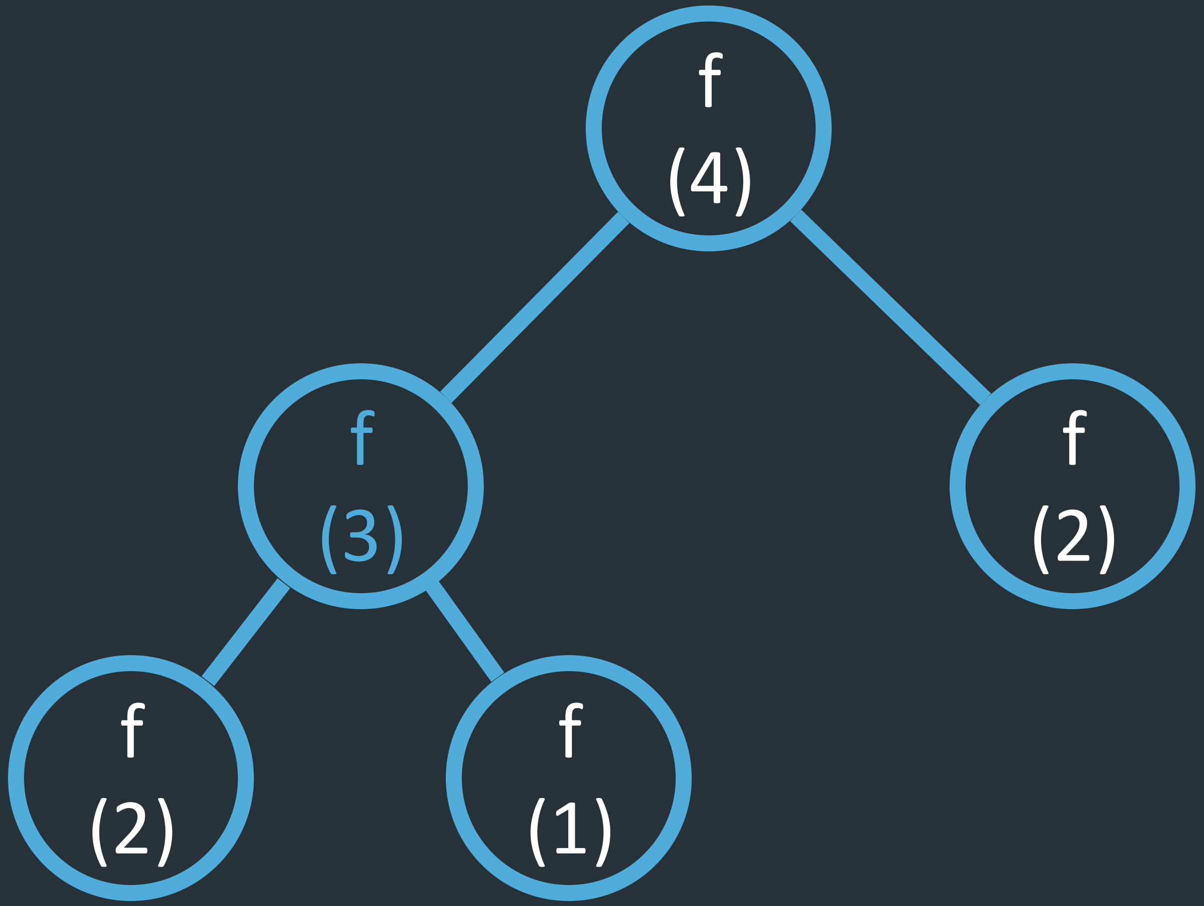



```
int f(int n) {
    if (n <= 1)
        return n;
    return f(n - 1) + f(n - 2);
}
```

재귀함수는

- n = 4

[함수 호출 수]
f(4): 1
f(3): 1
f(2): 2
f(1): 1
f(0): 0

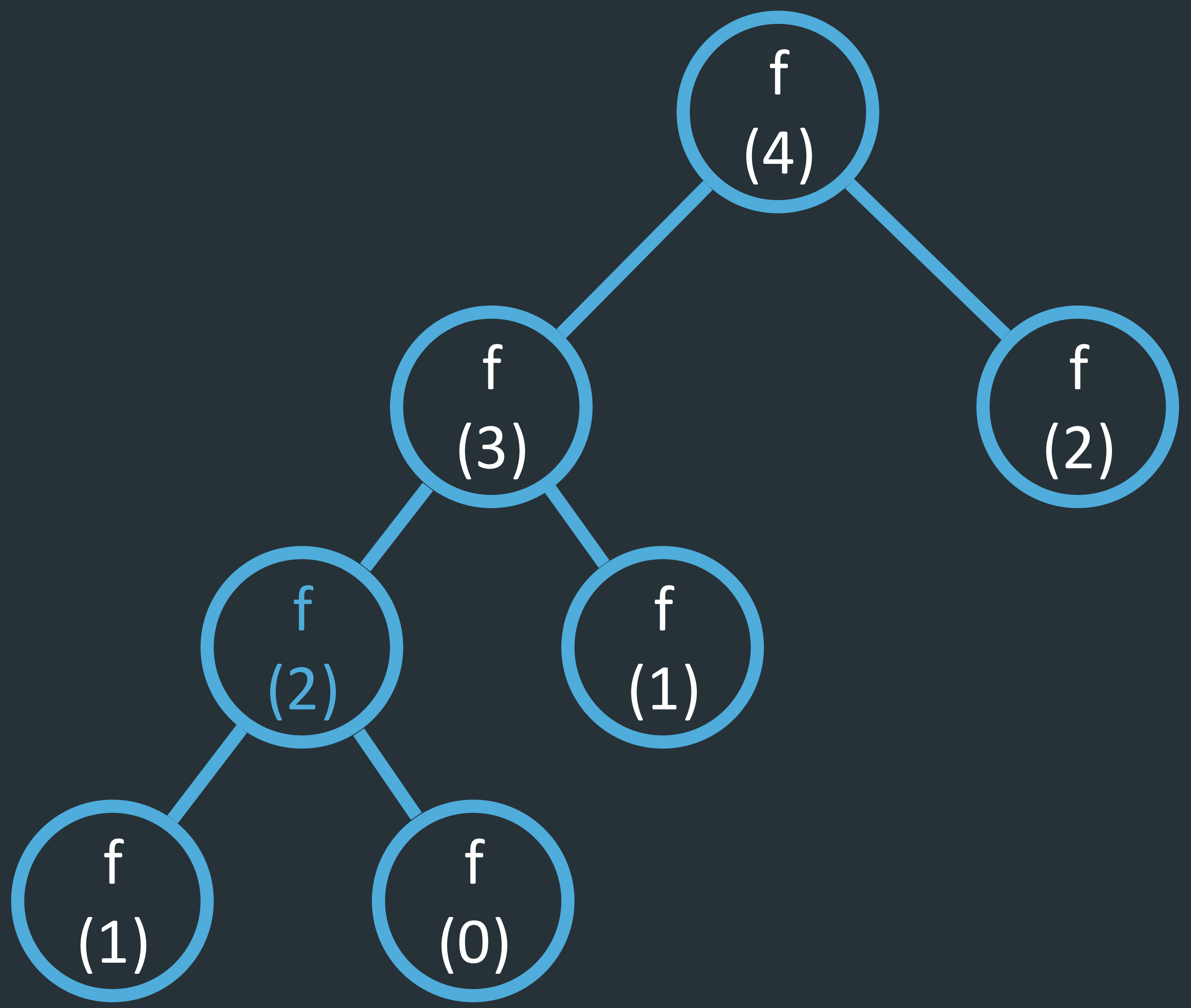



```
int f(int n) {
    if (n <= 1)
        return n;
    return f(n - 1) + f(n - 2);
}
```

재귀함수는

- n = 4

[함수 호출 수]
f(4): 1
f(3): 1
f(2): 2
f(1): 2
f(0): 1

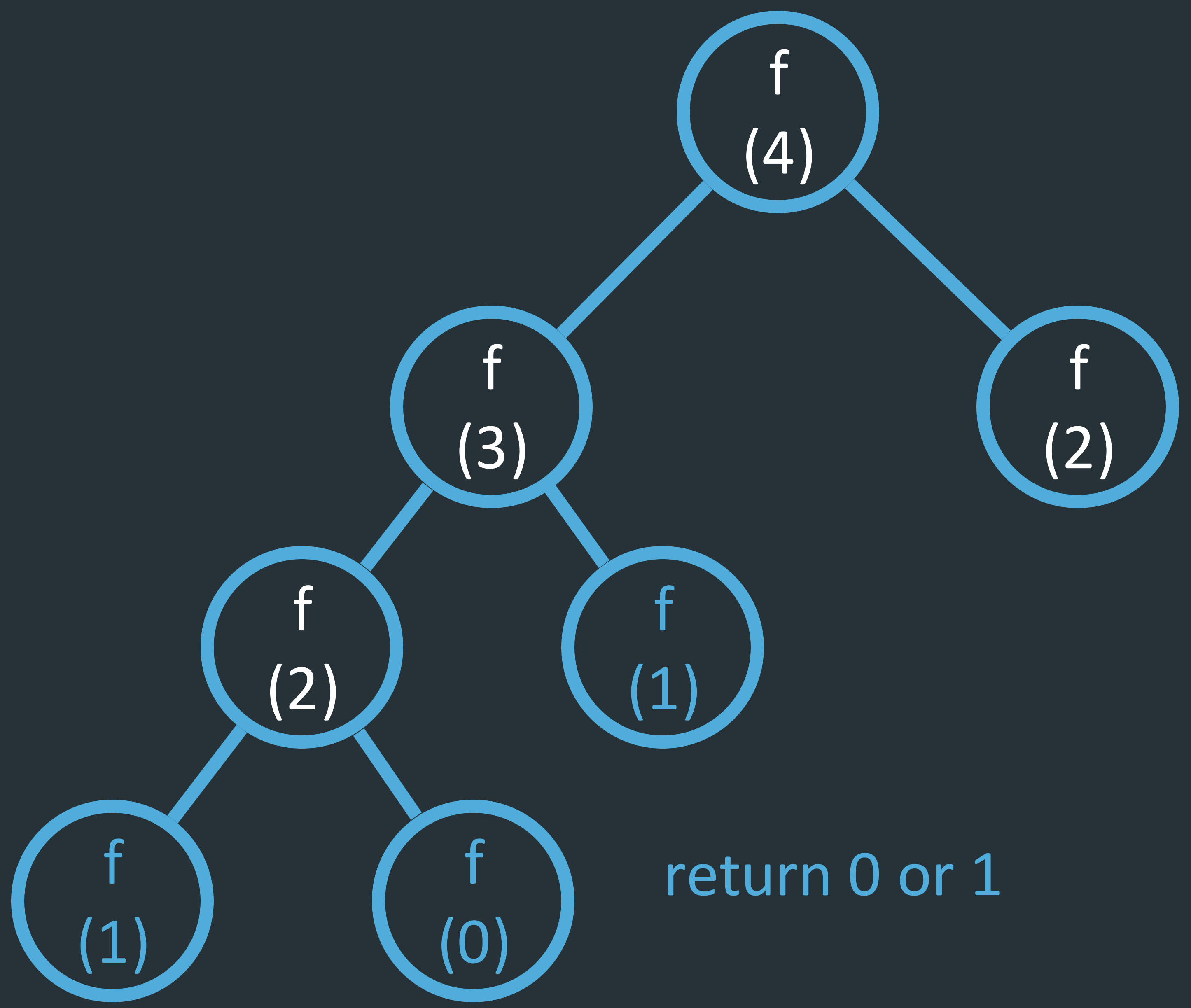



```
int f(int n) {
    if (n <= 1)
        return n;
    return f(n - 1) + f(n - 2);
}
```

재귀함수는

- n = 4

[함수 호출 수]
f(4): 1
f(3): 1
f(2): 2
f(1): 2
f(0): 1

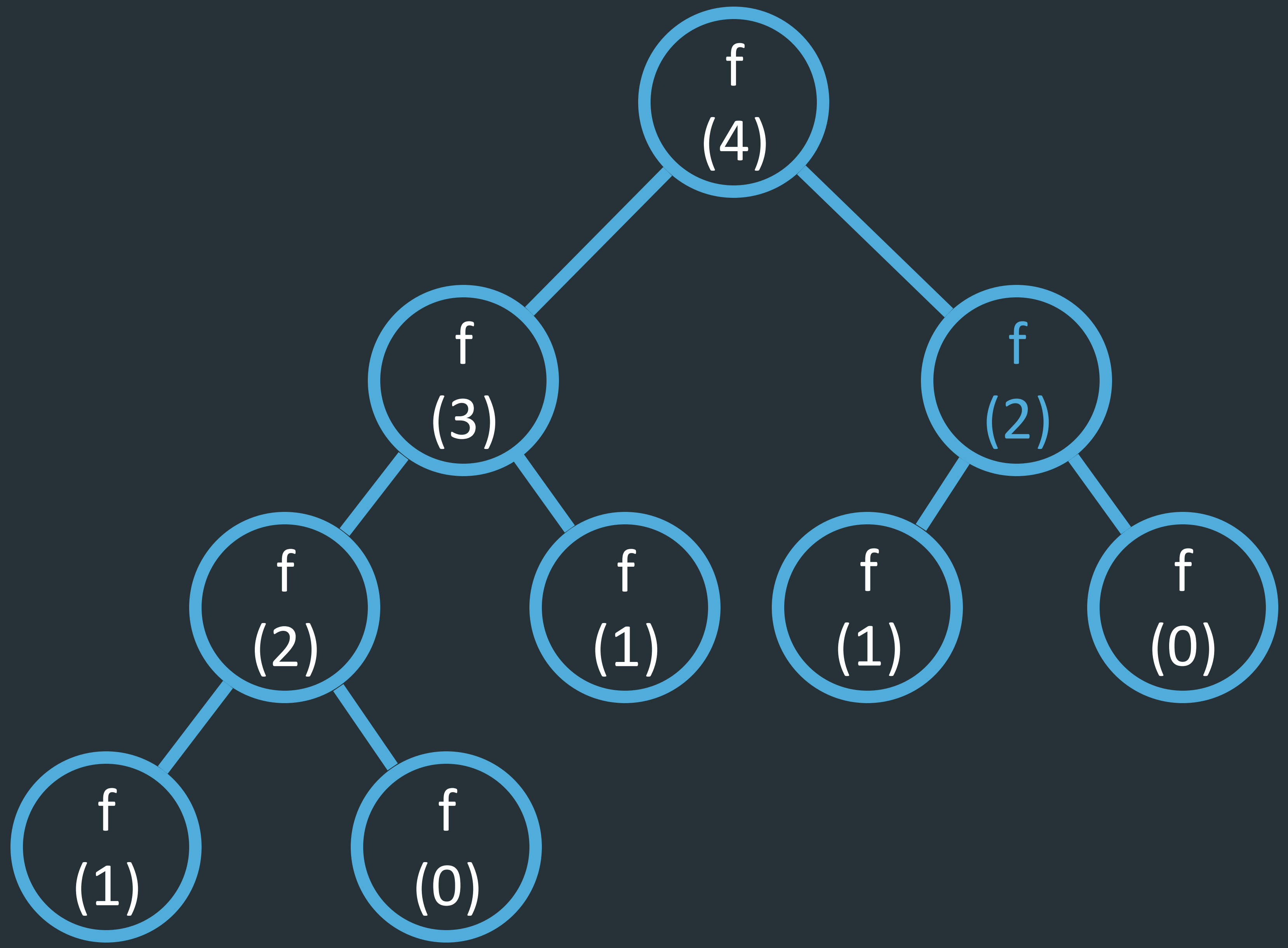



```c
int f(int n) {
    if (n <= 1)
        return n;
    return f(n - 1) + f(n - 2);
}
```

재귀함수는

- n = 4

[함수 호출 수]
f(4): 1
f(3): 1
f(2): 2
f(1): 3
f(0): 2

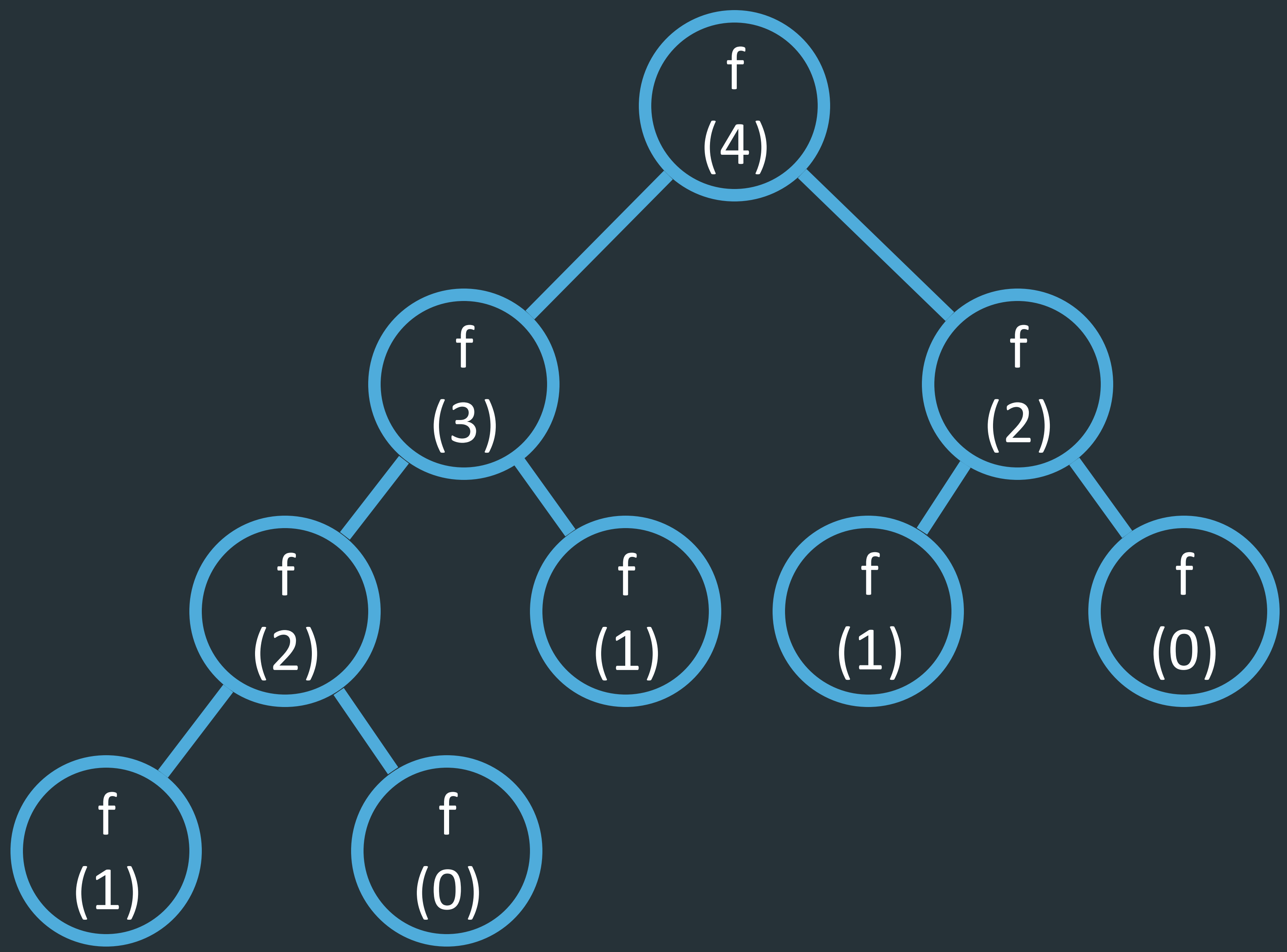



```
int f(int n) {
    if (n <= 1)
        return n;
    return f(n - 1) + f(n - 2);
}
```

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);
}
```

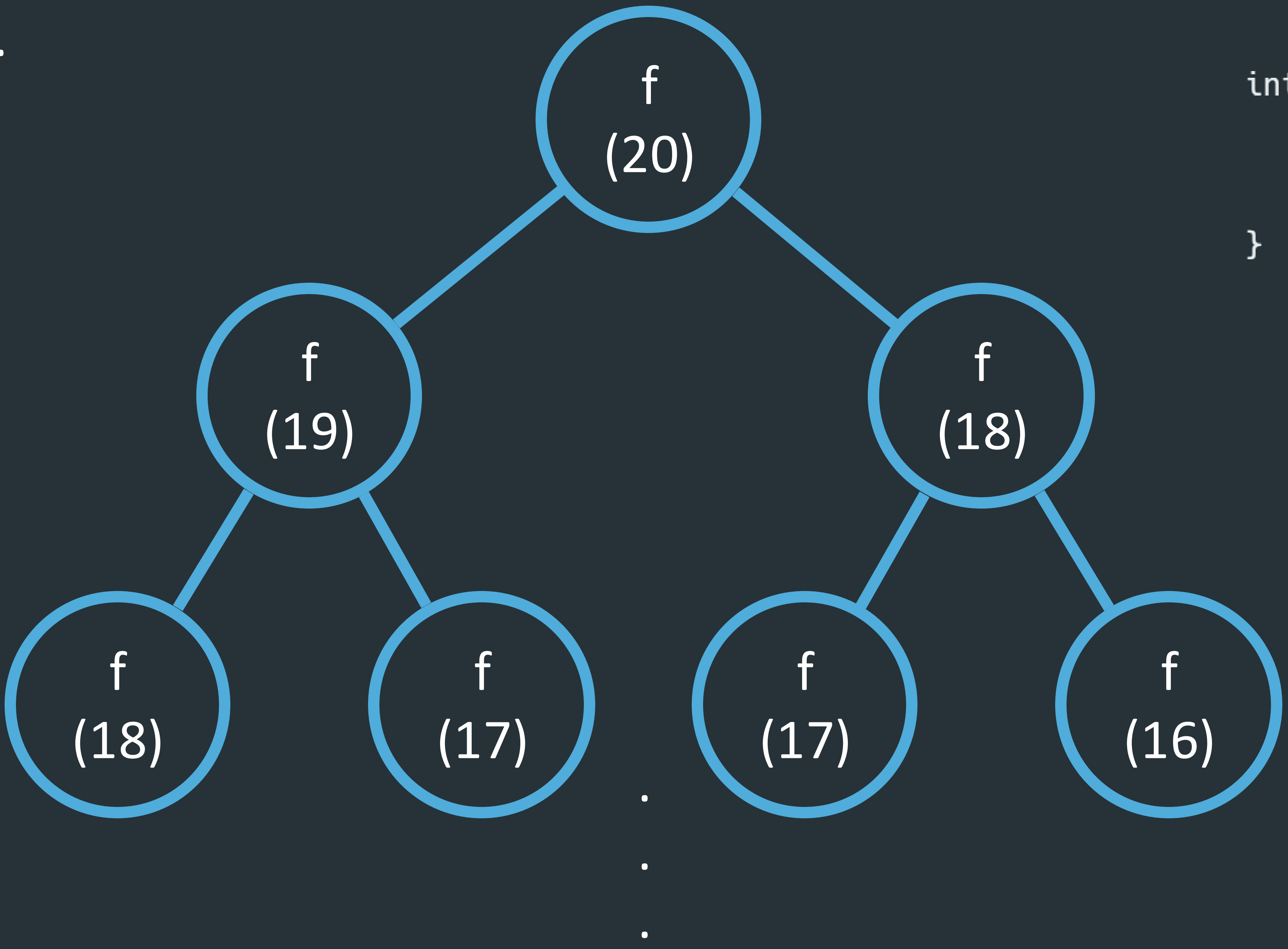
à 같은 함수를 여러 번 호출하는 경우가 많다!
à 즉, 한 번 계산한 값을 또 계산하게 됨

재귀함수는

- 당장 n = 20 이여도..

[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);
}
```

à N이 커지면? 함수 호출이 훨씬 많이 일어남
à 그렇다면, 이미 구한 답을 또 계산할 필요가 있을까?

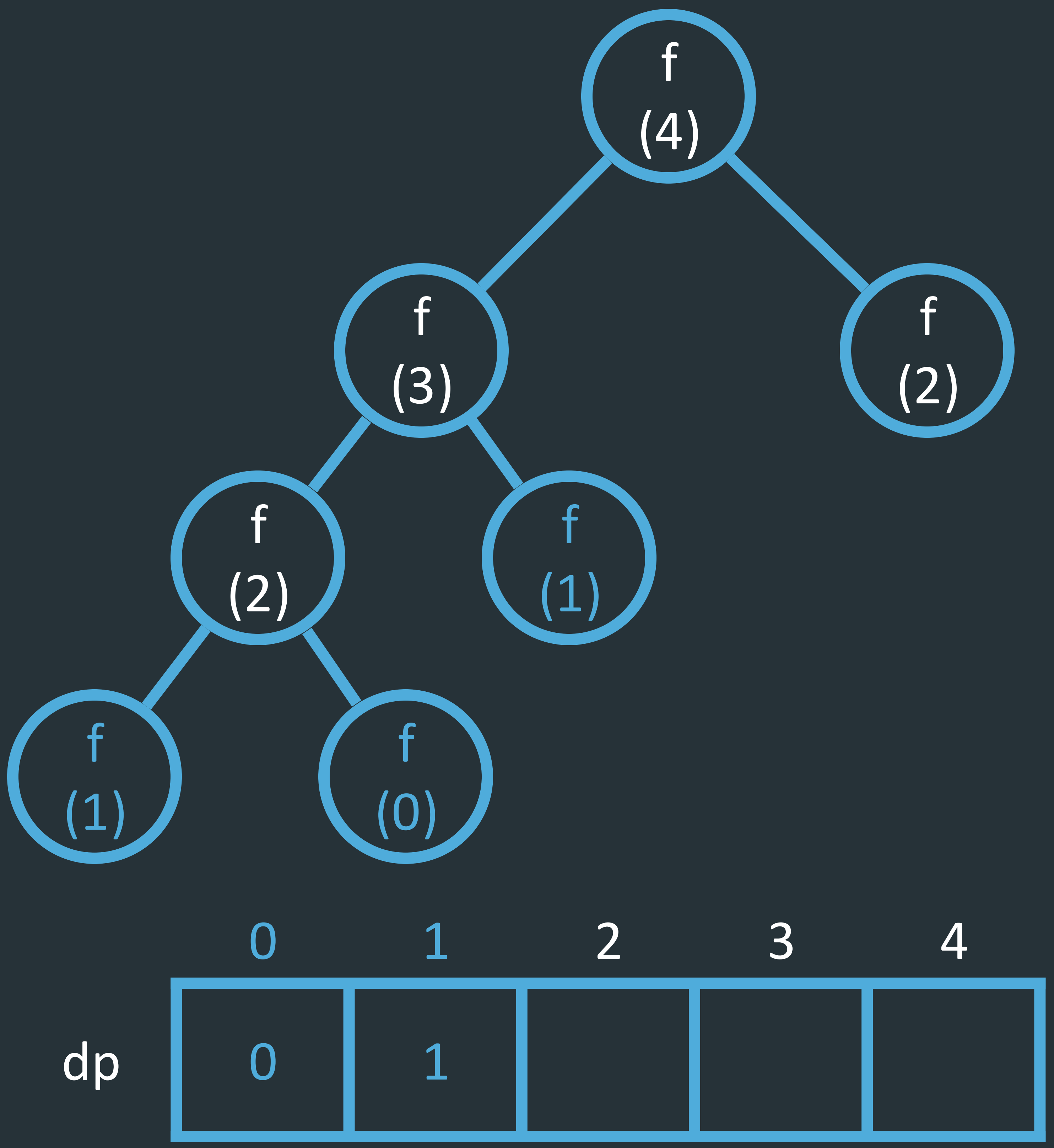
동적 계획법은

Memoization

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

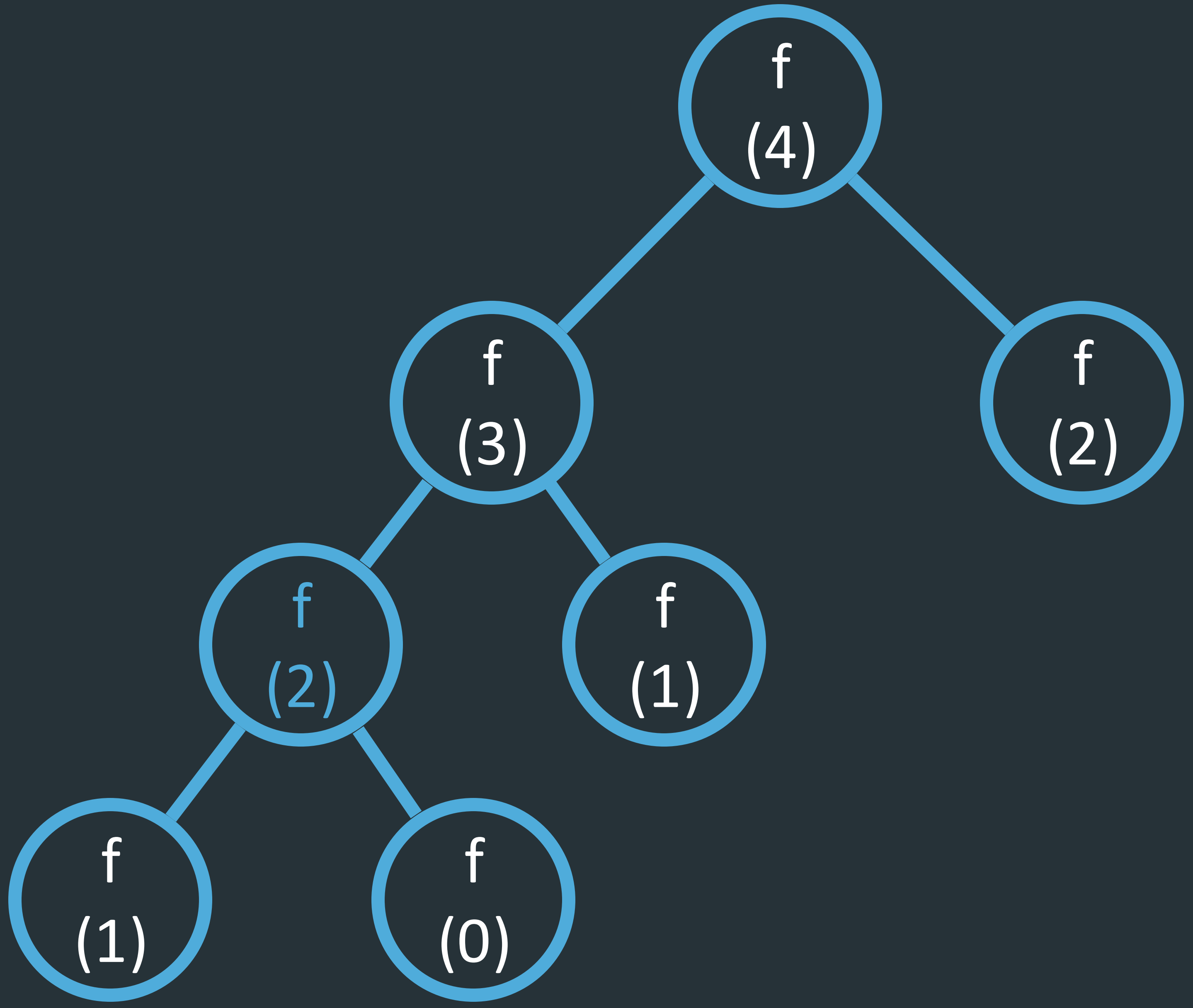
피보나치 수 문제에 적용하면

- n = 4



피보나치 수 문제에 적용하면

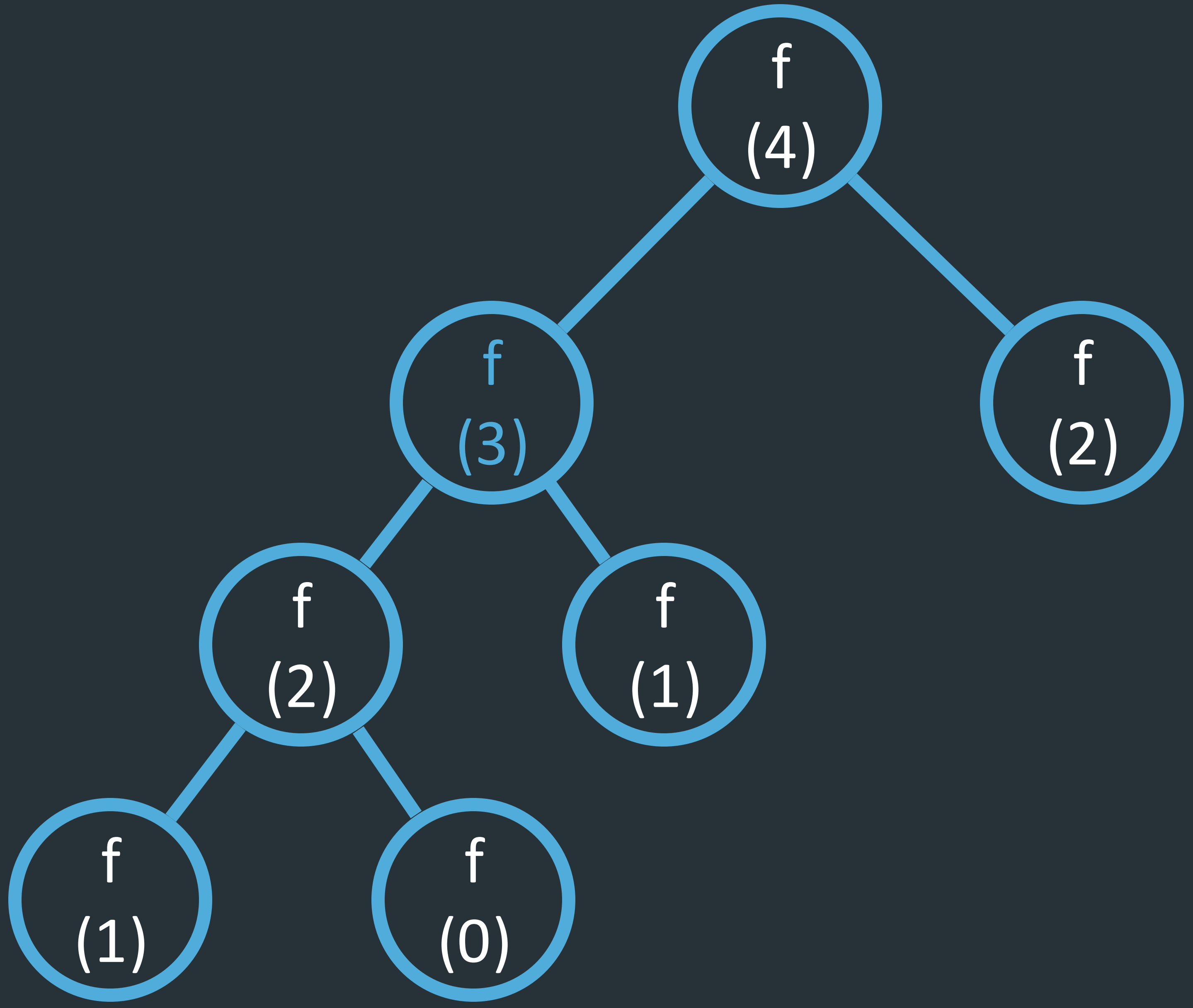
- n = 4



	0	1	2	3	4
dp	0	1	1		

피보나치 수 문제에 적용하면

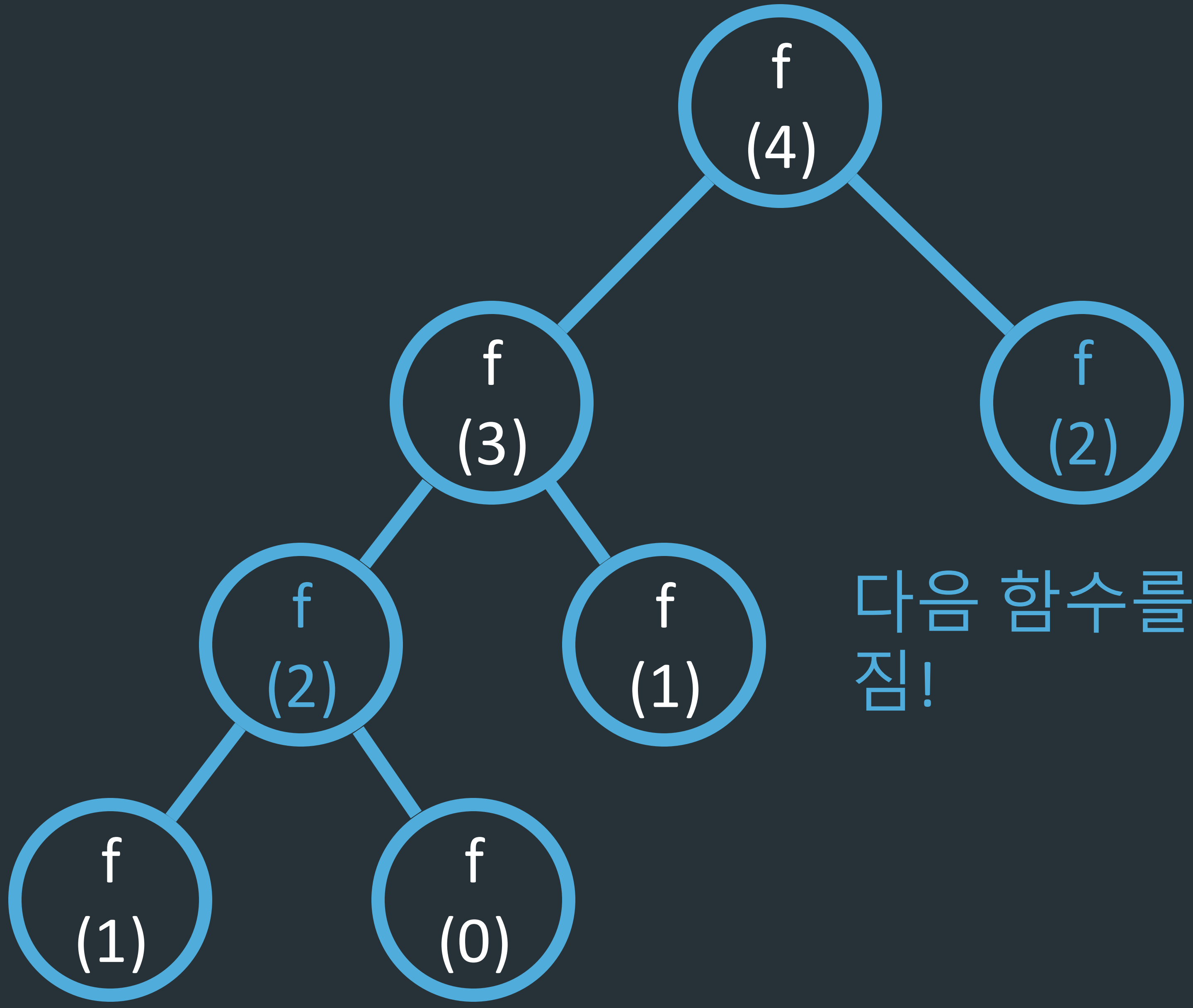
- n = 4



	0	1	2	3	4
dp	0	1	1	2	

피보나치 수 문제에 적용하면

- n = 4

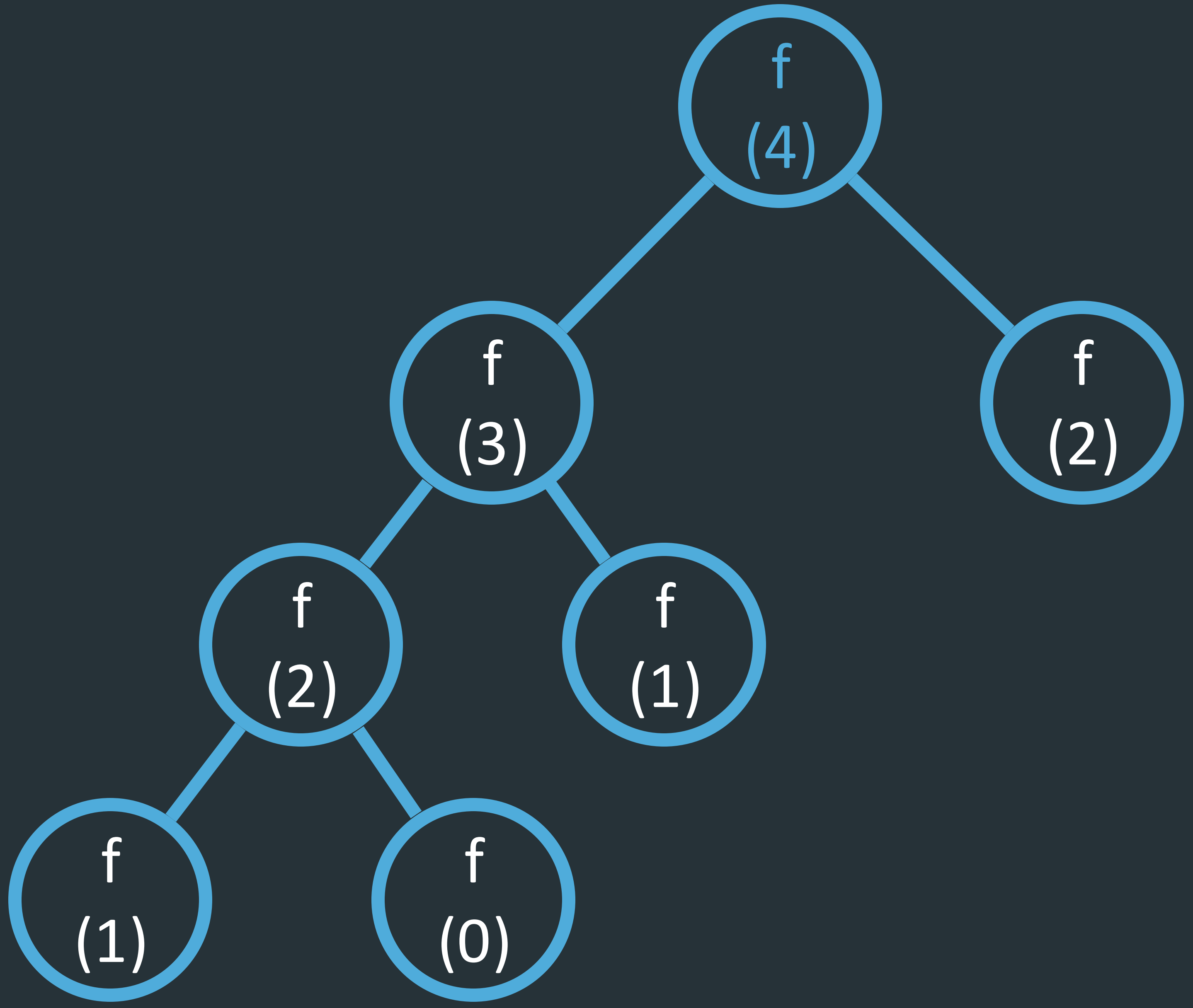


다음 함수를 호출할 필요가 없어 짐!

	0	1	2	3	4
dp	0	1	1	2	

피보나치 수 문제에 적용하면

- n = 4



	0	1	2	3	4
dp	0	1	1	2	3

피보나치 수 문제에 적용하면

단순 재귀함수

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

- 보통 n <= 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);
}
```

- n의 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);
}
```

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

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

다르게 구현할 수도 있어요

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

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

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

5928 KB	4 ms

- 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

어떻게 풀죠

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]

알고스팟 : 외발뛰기

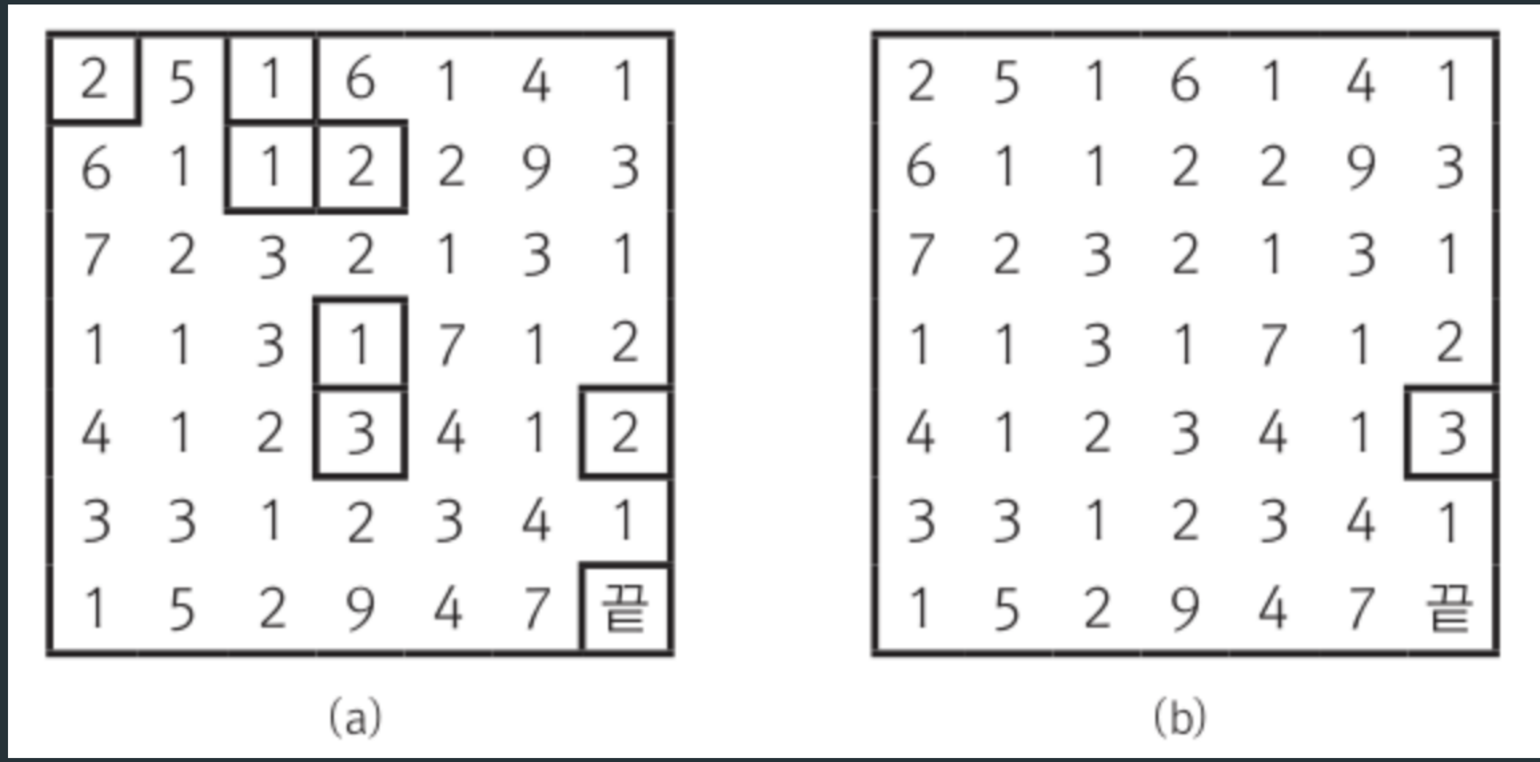
문제

- n*n 크기의 격자에 각 1부터 9 사이의 정수를 쓴 상태로 시작 하는 땅따먹기 게임
- 맨 왼쪽 윗 칸 에서 시작
- 각 칸에 적혀 있는 숫자만큼 오른쪽이나 아래 칸으로 움직일 수 있으며, 중간에 게임판 밖으로 벗어나면 안됨
- 왼쪽 위의 시작점에서 오른쪽 아래의 시작점에 도달할 수 있는지 구하는 문제

제한 사항

- 입력 케이스 C <= 50
- 격자 크기 n 2<=n<=100

예제



- (a)는 해당 경로를 통해 도달 가능
- (b)는 도달 불가능

예제

풀이 1: 재귀를 이용 (완전 탐색)

- 1. 시작점과 끝점을 매개변수로 받는 함수생성
- 2. 갈 수 있다면 다시 재귀 반복

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

예제 – 풀이 1 (완전 탐색)

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

예제 – 풀이 1 (완전 탐색)

#알튜비튜

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

예제 – 풀이 1 (완전 탐색)

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

범위를 벗어났기 때문에 더이
상 탐색 불가능

2	5	1	6	1	4	1
6	1	1	2	2	9	3
7	2	3	2	1	3	1
1	1	3	1	7	1	2
4	1	2	3	4	1	2
3	3	1	2	3	4	1
1	5	2	9	4	7	끝

예제 – 풀이 1 (완전 탐색)

풀이 1: 재귀를 이용 (완전 탐색)

- 1. 시작점과 끝점을 매개변수로 받는 함수생성
- 2. 갈 수 있다면 다시 재귀 반복

예제 – 풀이 2 (dp)

풀이 2: dp 사용

- 풀이1에서 두 번 이상 계산 될 수 있는 부분은?
- 각 칸에 도달하는 경우는 두 번 이상 계산될 수 있음

→ 이미 해당 칸이 도달 가능하다고 계산되면, 첫 번째 경우로만 계산함

예제 – 풀이 2 (dp)

- 왜 dp가 더 효율적일 수 있나요?

1	1	2	3
1	1	4	2
1	1	1	1
1	1	1	끝

1	1	2	3
1	1	4	2
1	1	1	1
1	1	1	끝

주황 칸에 도달할 수 있는 방법은 2가지
→ 이후 칸에 대해서 둘 중 하나의 경우만 탐색을 하면 탐색하는 경우의 수를 줄일 수 있음

예제 – 풀이 2 (dp)

```cpp
int result = dp(0, 0); // 0,0 시작점에서 시작하여 목적지에 도달할 수 있는지 확인
if (result)
    cout << "YES" << endl; // 도달 가능
else
    cout << "NO" << endl; // 도달 불가
```

```cpp
int board[101][101];
int check[101][101];
int n;

int dp(int x, int y)
{
    if (x >= n || y >= n)
        return 0;
    if (x == n - 1 && y == n - 1)
        return 1;

    if (check[x][y] != -1)
        return check[x][y];

    check[x][y] = (dp(x + board[x][y], y) || dp(x, y + board[x][y]));

    return check[x][y];
}
```

#알튜비튜

14501번: 퇴사 – Silver 3

문제

- N+1 되는 날 퇴사하기 위해 최대한 많은 상담을 하는 문제
- 각각의 상담은 상담을 완료하는데 걸리는 기간 T_i와 상담을 했을 때 받을 수 있는 금액 P_i로 이루어져있음
- 상담을 적절히 했을 때, 백준이가 얻을 수 있는 최대 수익을 구하는 문제

제한 사항

- 입력 케이스 N ($1 \le N \le 15$)
- ($1 \le T_i \le 5$, $1 \le P_i \le 1{,}000$)

예제

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 최대 이익: 1일, 4일, 5일 상담 → 10 + 20 + 15 = 45

예제 – 풀이 1 (완전 탐색)

풀이 1: 완전 탐색

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 일을 기준으로 해당 일에 상담을 할 수 있는지 없는지 전부 탐색한 후, 최댓값을 갱신하는 방법

예제 – 풀이 1 (완전 탐색)

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 1. 7개 중 가능한 조합을 모두 찾기: 7c1,7c2,7c3,7c4,7c5,7c6,7c7
- 2. 해당 조합이 가능한지에 대해 검토(ex 1일, 2일 ,3일 조합은 안됨)

→ 입력 n개에 대한 시간 복잡도 : 2^n

예제 – 풀이 2 (dp)

풀이 2: dp (Bottom-up)

- 1일차부터 시작하여, 해당 상담 진행 일자 + 상담 진행 시 걸리는 시간 보다 큰 시간에 대해 모두 갱신

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 1일에 상담 진행할 때: 상담에 3일이 걸리기 때문에, 4,5,6,7일에 상담 가능
  → 1일 dp 배열: [0, 0, 0, 0, 0, 10, 10, 10, 10]

- 2일에 상담 진행할 때: 상담에 5일이 걸리기 때문에, 7일에 상담 가능
  → 2일 dp 배열: [0, 0, 0, 0, 10, 10, 10, 20]

- 1일 상담, 2일 상담 둘 다 7일에 상담이 가능함
  그러나, 1일, 2일 상담 모두를 진행할 수는 없음
  → 1일과 2일 상담 중 이득이 큰 쪽을 선택

- dp[7] = max(10, 20)
  즉, 2일차 dp[7] = max(dp[7], 20)
  → max(기존 dp[7], 새로 갱신된 Pi)

예제 – 풀이 2 (dp)

- 1일차부터 시작하여, 해당 상담 진행 일자 + 상담 진행 시 걸리는 시간 보다 큰 시간에 대해 모두 갱신

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 3일에 상담 진행할 때: 상담에 1일이 걸리기 때문에, 4,5,6,7일에 상담 가능
  → 3일 dp 배열: [0, 0, 0, 0, 10, 10, 10, 20]

- 4,5,6일의 경우:
  1일 상담, 4일 상담 → 기존에 갱신되었던 dp[4]
  3일 상담, 4일 상담 → dp[4] = max(dp[4], dp[3] + 10)

- 7일의 경우:
  1일 상담, 7일 상담 → 이경우 아까 2일 상담보다 이익이 적었으므로, 2일 상담에서 이미 걸러짐
  2일 상담, 7일 상담 → 기존에 갱신되었던 dp[7]
  3일 상담, 7일 상담 → dp[7] = max(dp[7], dp[3] + 10)

예제 – 풀이 2 (dp)

- 1일차부터 시작하여, 해당 상담 진행 일자 + 상담 진행 시 걸리는 시간 보다 큰 시간에 대해 모두 갱신

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 4일에 상담 진행할 때: 상담에 1일이 걸리기 때문에 ,5,6,7일에 상담 가능
  → 4일 dp 배열: [0, 0, 0, 0, 10, 30, 30, 30]

- 5일에 상담 진행할 때: 상담에 2일이 걸리기 때문에 ,7일 상담 가능
  → 5일 dp 배열: [0, 0, 0, 0, 10, 30, 30, 45]

예제 – 풀이 2 (dp)

- 1일차부터 시작하여, 해당 상담 진행 일자 + 상담 진행 시 걸리는 시간 보다 큰 시간에 대해 모두 갱신

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 6일에 상담 진행할 때: 상담에 4일이 걸리기 때문에 다른 날 상담 불가능, 6+4는 7을 넘어가기 때문에 갱신x
  → 6일 dp 배열: [0, 0, 0, 0, 10, 30, 30, 45]

- 7일에 상담 진행할 때: 상담에 2일이 걸리기 때문에 다른 날 상담 불가능
  → 7일 dp 배열: [0, 0, 0, 0, 10, 30, 30, 45]

예제 – 풀이 2 (dp)

- 1일차부터 시작하여, 해당 상담 진행 일자 + 상담 진행시 걸리는 시간 보다 큰 시간에 대해 모두 갱신

```
void dpBottomUp(vector<int> dp, vector<pair<int, int> > li, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + li[i].first; j <= n; j++) {
            if (dp[j] < dp[i] + li[i].second) { //만약에 기존 값보다 크다면 새로 dp[j] 값 갱신
                dp[j] = dp[i] + li[i].second;
            }
        }
    }
    cout << dp[n] << endl;
}
```

예제 – 풀이 3 (Top-down)

풀이 3: dp (Top-down)

- N일차부터 시작하여, 역순으로 dp 진행 i일에 상담 진행했을 때, 얻는 이익을 i번째에 저장

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 7일에 상담 시작: 2일이 걸리므로 7을 넘어감 → 상담 불가
  [0, 0, 0, 0, 0, 0, 0, 0]

- 6일에 상담 시작: 4일이 걸리므로 7을 넘어감 → 상담 불가
  [0, 0, 0, 0, 0, 0, 0, 0]

예제 – 풀이 3 (Top-down)

- N일차부터 시작하여, 역순으로 dp 진행 i일에 상담 진행했을 때, 얻는 이익을 i번째에 저장

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 선택할 수 있는 것 2가지 : 상담을 하거나, 상담을 하지 않거나

- 5일에 상담 시작: 2일이 걸리므로 상담 가능
  [0, 0, 0, 0, 15, 0, 0, 0]
  → 5일의 경우 상담해도 7일 상담이 가능함. 즉, dp[5]는 7일에 상담 이익을 얻을 수 있음
  → dp[5] = max(dp[6], dp[7] + 15)

- 4일에 상담 시작: 1일이 걸리므로 상담 가능
  [0, 0, 0, 35, 15, 0, 0, 0]
  → 4일의 경우 상담해도 5,6,7일 상담이 가능함. 즉, dp[4]는 5,6,7일에 상담 이익을 얻을 수 있음
  → dp[4] = max(dp[5], dp[5] + 20)

예제 – 풀이 3 (Top-down)

- N일차부터 시작하여, 역순으로 dp 진행 i일에 상담 진행했을 때, 얻는 이익을 i번째에 저장

	1일	2일	3일	4일	5일	6일	7일
T_i	3	5	1	1	2	4	2
P_i	10	20	10	20	15	40	200

- 3일에 상담 시작: 1일이 걸리므로 상담 가능
  [0, 0, 45, 35, 15, 0, 0, 0]
  → 3일의 경우 상담해도 4,5,6,7일 상담 가능함. 즉, dp[3]은 4,5,6,7일에 상담 이익을 얻을 수 있음
  → dp[3] = max(dp[4], dp[4] + 10)

- 2일에 상담 시작: 5일이 걸리므로 상담 가능
  [0, 45, 45, 35, 15, 0, 0, 0]

- 1일에 상담 시작: 3일이 걸리므로 상담 가능
  [45, 45, 45, 35, 15, 0, 0, 0]

예제 – 풀이 3 (Top-down)

- N일차부터 시작하여, 역순으로 dp 진행 i일에 상담 진행했을 때, 얻는 이익을 i번째에 저장

```
void dpTopDown(vector<int> dp, vector<pair<int, int> > li, int n) {
    for (int i = n-1; i >= 0; i--) {
        if (i + li[i].first > n) // i일에 상담하는 것이 n을 넘기면 상담을 할 수 없음
            dp[i] = dp[i + 1];
        else // 상담을 하는 것과 하지 않는 것 중 큰 값을 선택
            dp[i] = max(dp[i + 1], li[i].second + dp[i + li[i].first]);
    }

    cout << dp[0] << endl;
}
```

14002번: 가장 긴 증가하는 부분 수열 4 – Gold 4

문제

- 수열 A가 주어졌을 때, 가장 긴 증가하는 부분 수열을 구하는 프로그램을 작성
- 수열 A의 크기 N
- 수열 A를 이루고 있는 A_i

제한 사항

- 입력 케이스 N (1 ≤ N ≤ 1,000)
- $(1 \le A_i \le 1{,}000)$

예제

예제 입력

```
6
10 20 10 30 20 50
```

A = {10, 20, 10, 30, 20, 50}

예제 출력

```
4
10 20 30 50
```

A = {10, 20, 10, 30, 20, 50}

예제

풀이: dp 사용   A = {10, 20, 10, 30, 20, 50}

- 1. dp[i]에 인덱스 i번째에 해당하는 값을 넣는다고 생각
- 2. 인덱스 i 이전에 값에 대해 탐색, j<i일때, 만약 A[i]보다 A[j]가 작다면, dp[i] = dp[j] + 1
- 3. 부분 수열 출력을 위해 길이가 가장 긴 부분 수열이 될 때마다, 가장 마지막 index를 갱신해 줌

예제

A = {10, 20, 10, 30, 20, 50}

i = 0

A[0] = 10

dp[0] = 1 (10 하나 들어가는 수열)

dp: [1, 0, 0, 0, 0, 0]
→ max_index = 0

i = 1

A[1] = 20

dp[1] = 1 (20 하나 들어가는 수열)

인덱스 1번 이전 인덱스 (0에 대해 탐색)
A[0] < A[1] → dp[1] = max(dp[1], dp[0] + 1)

dp: [1, 2, 0, 0, 0, 0]
→ max_Index = 1

예제

A = {10, 20, 10, 30, 20, 50}

i = 2

A[2] = 10

dp[2] = 1 (10 하나 들어가는 수열)

인덱스 2번 이전 인덱스 (0,1에 대해 탐색)
→ A[0]=10, A[1]로 10보다 작지 않기 때문에 갱신x

dp: [1, 2, 1, 0, 0, 0]

i = 3

A[3] = 30

dp[3] = 1 (30 하나 들어가는 수열)

인덱스 2번 이전 인덱스 (0,1,2에 대해 탐색)
dp[3] = max(dp[0]+1, dp[1]+1, dp[2]+1, dp[3])

dp: [1, 2, 1, 3, 0, 0]
→ max_index = 3

예제

A = {10, 20, 10, 30, 20, 50}

i = 4

A[4] = 20

dp: [1, 2, 1, 3, 2, 0]

i = 5

A[5] = 50

dp: [1, 2, 1, 3, 2, 4]
→ max_index = 5

예제

```
for(int i = 0; i < n; i++)
    {
        cin >> a[i];
        len = 0;

        for(int j = 0; j < i; j++)
        {
            if(a[i] > a[j])
                len = max(dp[j], len);
        }
        dp[i] = len + 1;

        if(tmp < dp[i])
        {
            tmp = dp[i];
            idx = i;
        }
    }
```

dp 갱신 부분

```
for(int i = idx;i >= 0;i--)
{
    if(tmp == dp[i])
    {
        arr.push_back(a[i]);
        tmp--;
    }
}
```

수열 출력을 위한 배열 만들기

마무리

#알튜비튜

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!

이것도 알아보세요!

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

과제

필수

/<> 2579번: 계단 오르기 - Silver 3
/<> 11053번 : 가장 긴 증가하는 부분 수열 – Silver 2
/<> 20923번 : 숫자 할리갈리 게임 – Silver 1

도전

/<> 2240번 : 자두나무 – Gold 5
/<> 12865번 : 평범한 배낭 - Gold 5

과제 마감일

과제제출 마감 ~ 4월 2일 화요일 18:59

추가제출 마감 ~ 4월 4일 목요일 23:59