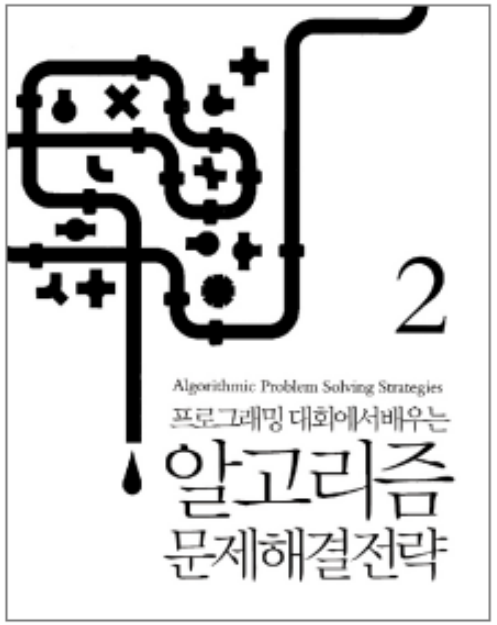
동적계획법
Dynamic Programming

DP의 메모이제이션 기법과 문제풀이 팁에 대한 소개

솔직하게, 동적계획법은 처음 접할 때 어렵습니다

구종만의 <알고리즘 문제해결 전략> 책에서도 160페이지에 걸쳐서 동적계획법을 설명

가장 어려워하는 유형으로 많이 뽑히기도 합니다



다이나믹 프로그래밍 (DP)

Dynamic	역동적인, 동적인, 활발한
Programming	프로그래밍

다이나믹 프로그래밍 (DP)

역동적인 프로그래밍(?)

Dynamic

역동적인, 동적인, 활발한

Programming

프로그래밍

뭔지 모르겠지만 멋있다

나도 어렵고 멋있는 문제 푼다

다이나믹 프로그래밍 (DP)

여담

Dynamic 역동적인, 동적인, 활발한

Programming 프로그래밍

그런데 실제로 DP 고안자인 Richard E. Bellman은

`Dynamic`이라는 단어가 멋있어서 사용했다고 합니다

다이나믹 프로그래밍 (DP)

주어진 큰 문제를 작은 조각의 문제들로 나누고
각 조각들의 답들로부터 원래 문제의 답을 도출

그 과정에서 발생되는 중복된 연산을 제거하여
문제풀이에 사용되는 연산(계산)량을 최소화

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

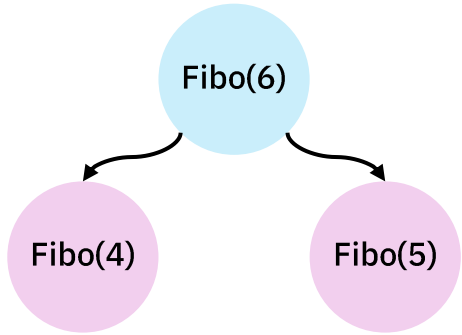
Fibo(6)

```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

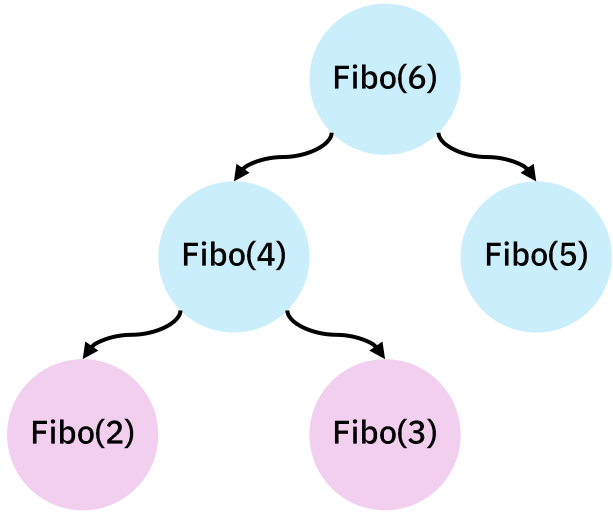
```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```



예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

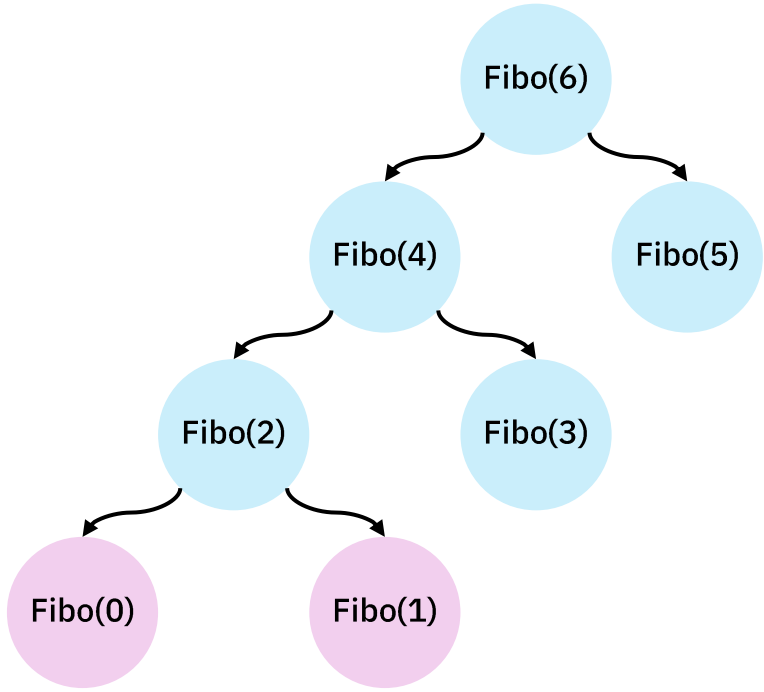
```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```



예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

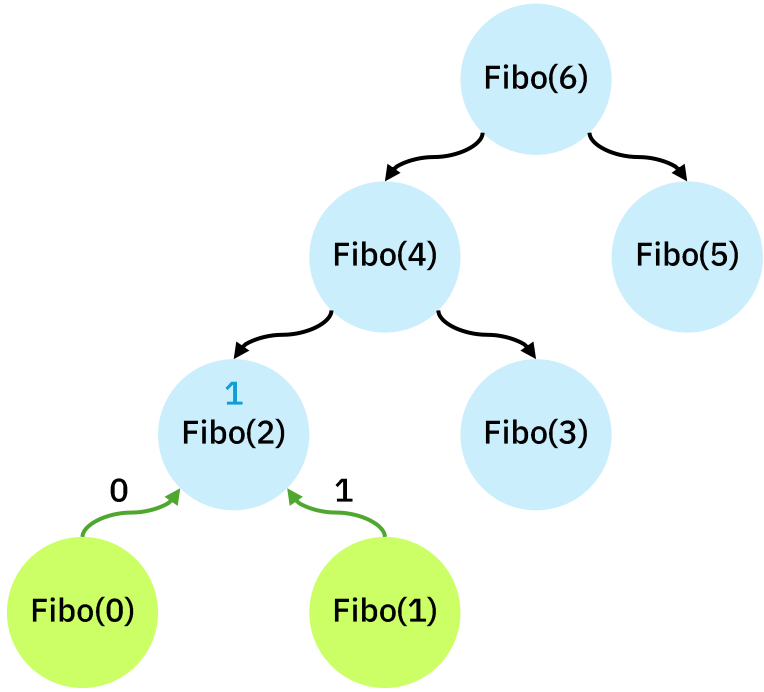
```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```



예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```

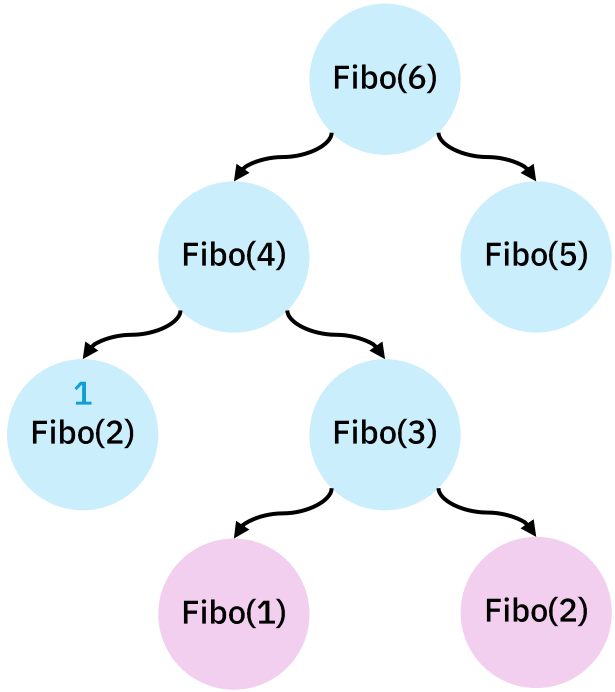


Base condition 에 의해 1 반환

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

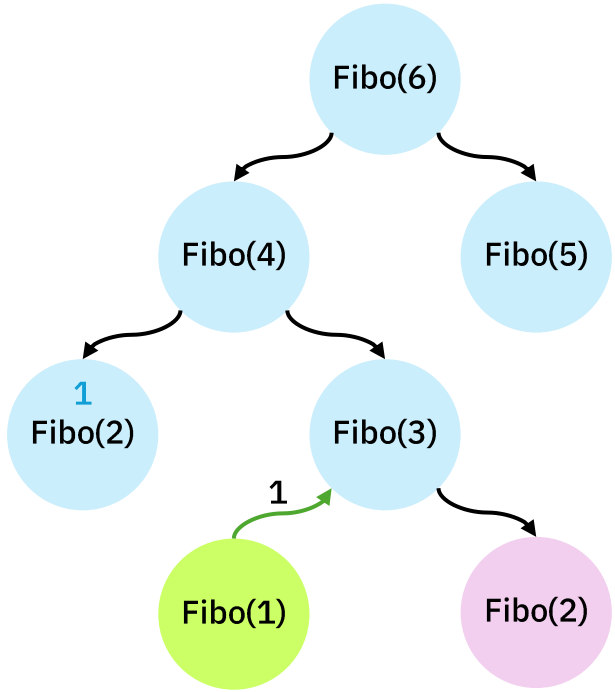
```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```



예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```

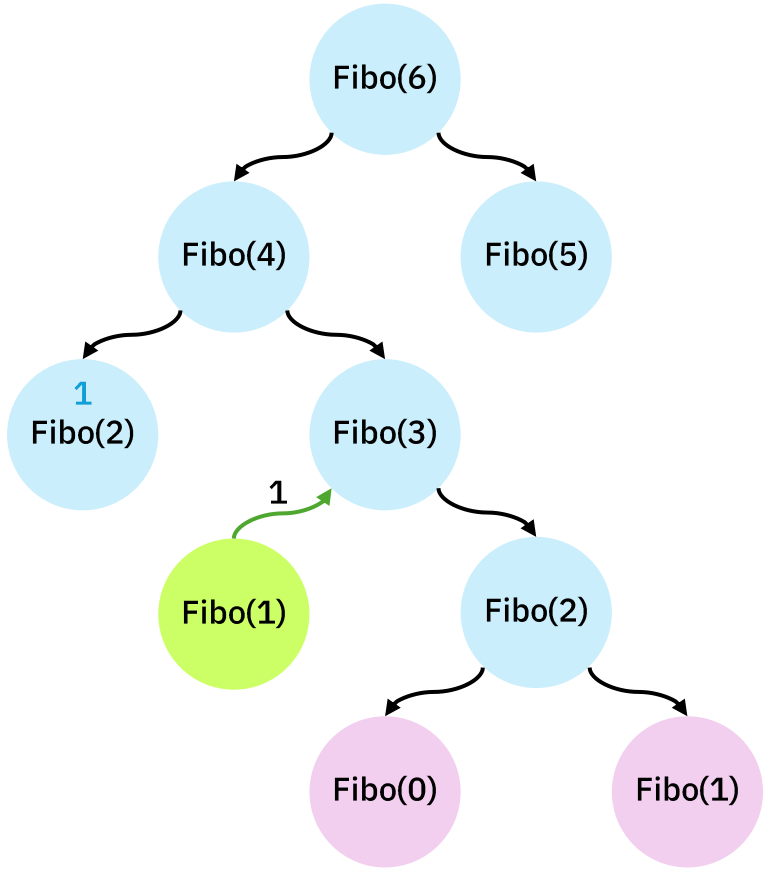


Base condition 에 의해 1 반환

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

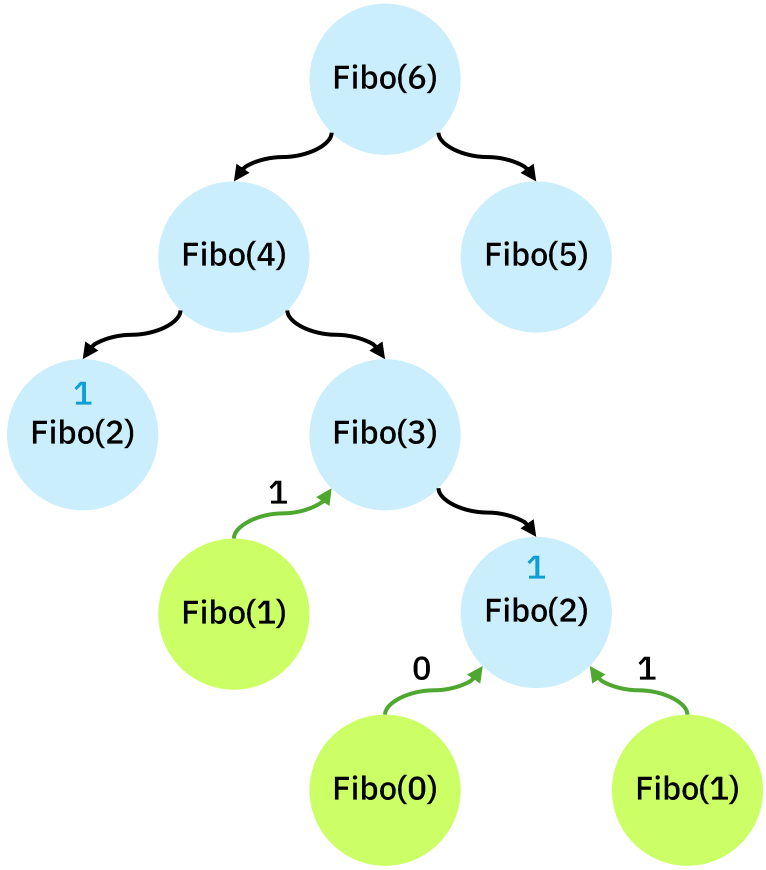
```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```



예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```

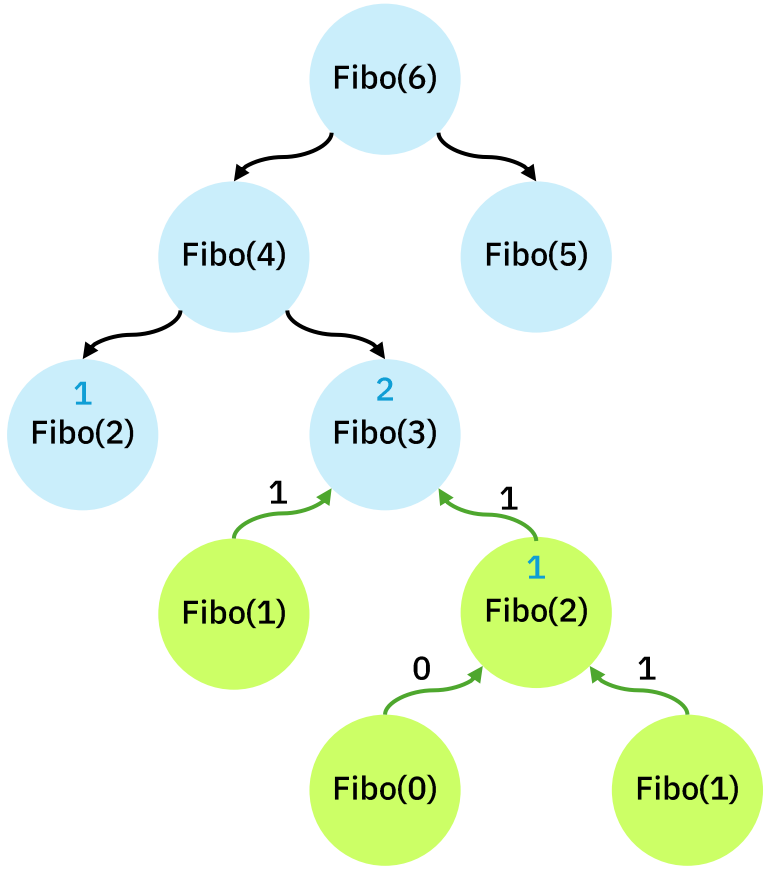


Base condition 에 의해 1 반환

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```

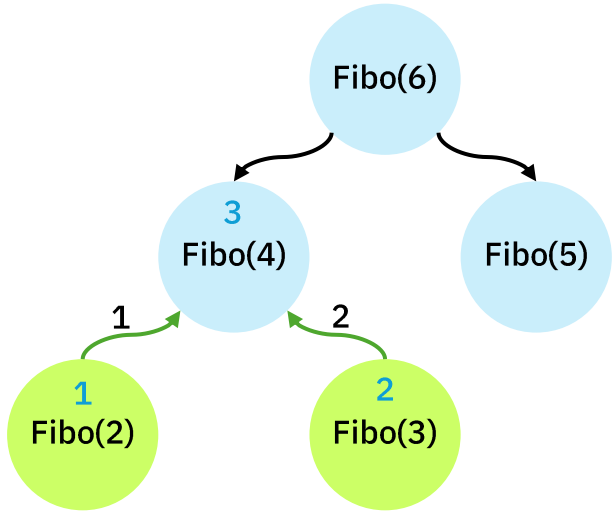


Base condition 에 의해 1 반환

예시: 피보나치 수 계산

6번째 Fibonacci 수 를 알고 싶다! (재귀 구현)

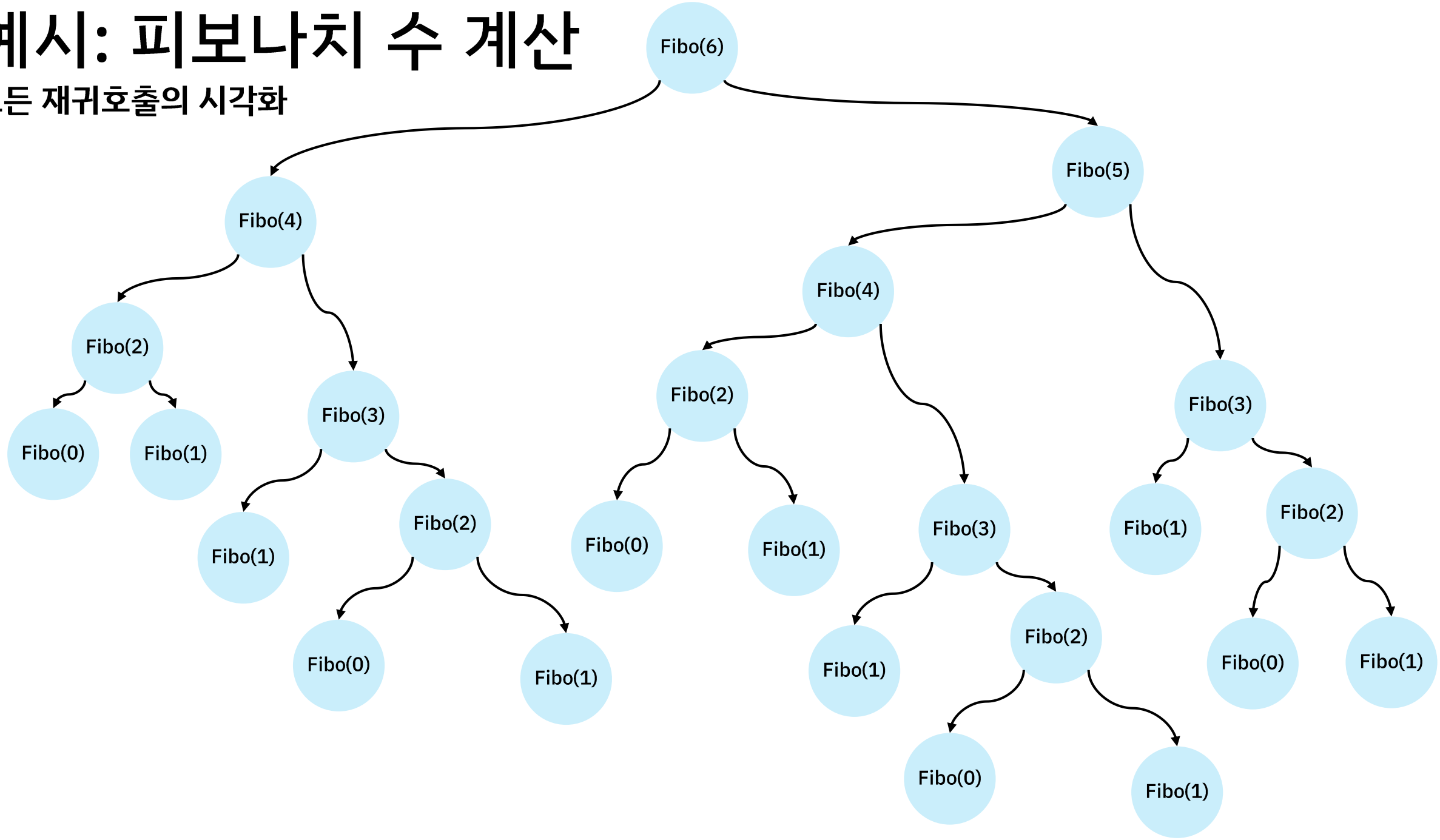
```python
def fibonacci(n):
  """
  n 번째 fibonacci 수를 반환하는 함수
  """

  # base condition: 첫번째와 두번째 피보나치수는 0
  if n <= 1:
    return 0

  # solver: 이전에 등장한 2개의 수의 합
  return fibonacci(n - 2) + fibonacci(n - 1)
```



이러한 구조로 계속 반복..

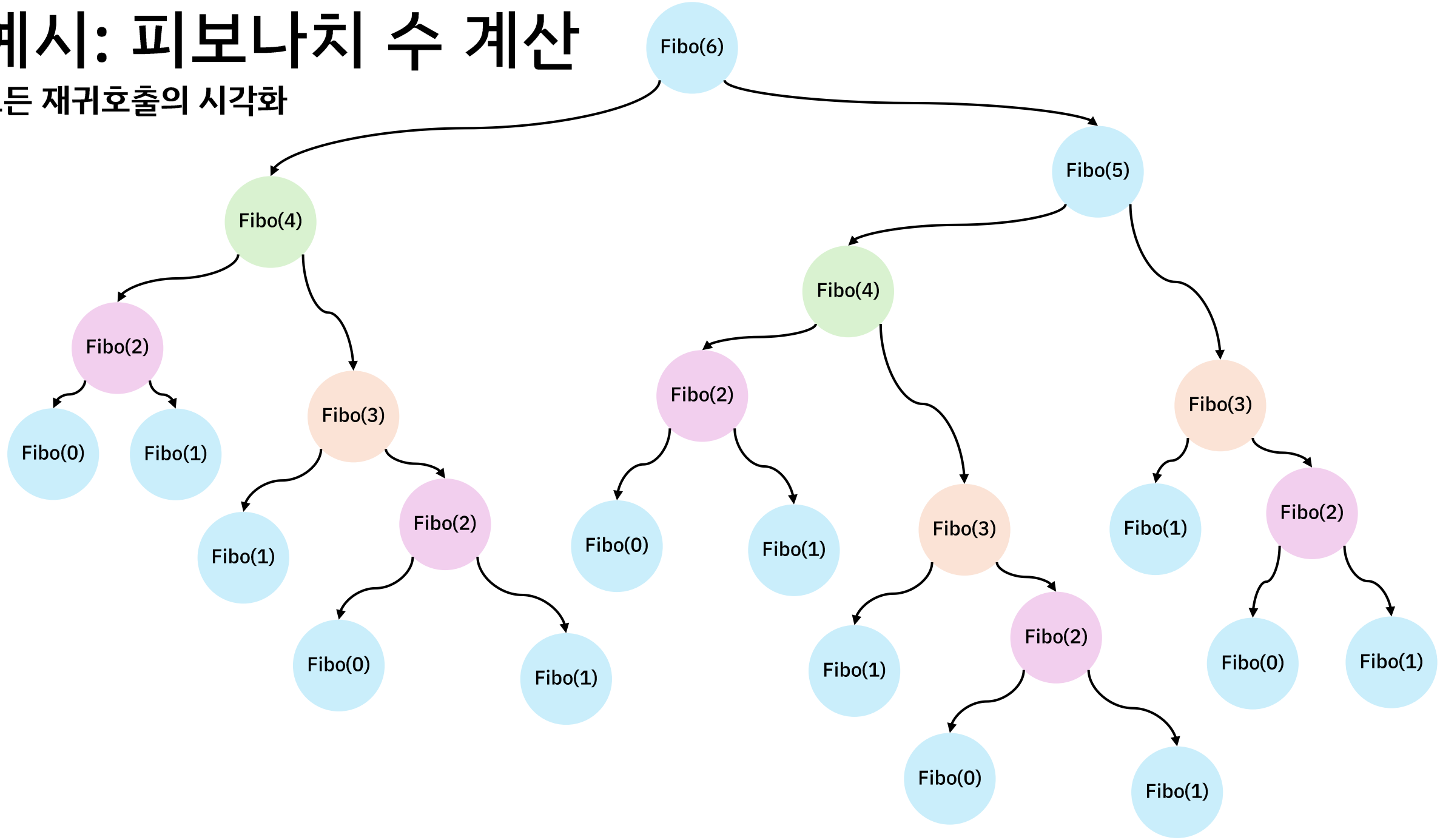
예시: 피보나치 수 계산

모든 재귀호출의 시각화



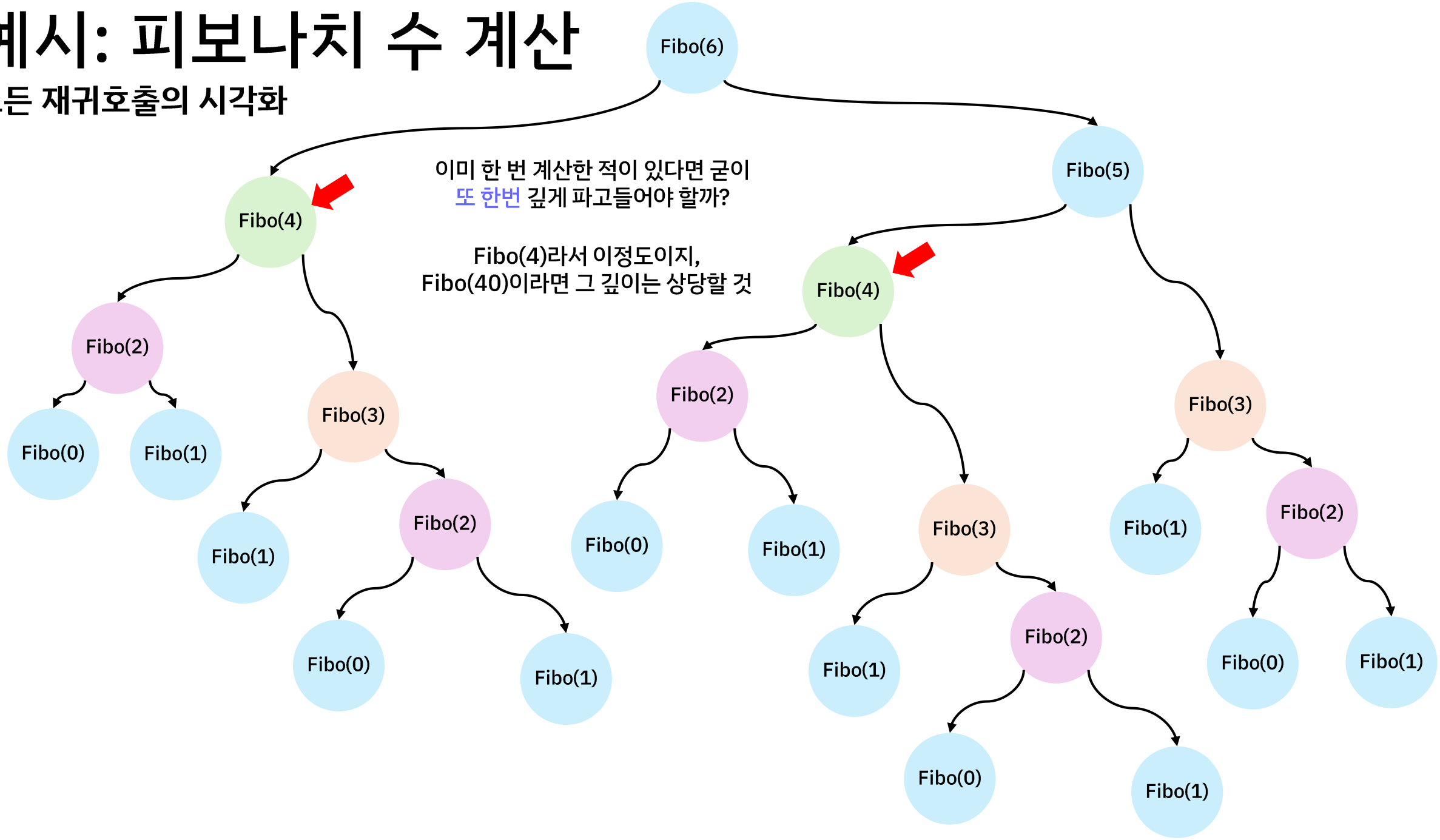
예시: 피보나치 수 계산

모든 재귀호출의 시각화



예시: 피보나치 수 계산

모든 재귀호출의 시각화



```python
import time

# 결과 값 기록을 위한 리스트
memo = [-1 for _ in range(100)]

# 일반적인 피보나치 호출
def fibonacci(n):
    if n <= 1:
        return 0

    return fibonacci(n - 2) + fibonacci(n - 1)

# 동적계획법이 적용된 피보나치 호출
def fibonacci2(n):
    if n <= 1:
        return 0

    if memo[n] != -1:
        return memo[n]

    memo[n] = fibonacci2(n - 2) + fibonacci2(n - 1)
    return memo[n]


# 계산시간 측정
start_time = time.time()
fibonacci(40)
print(f"Fibonacci Execution: Took {time.time() -
start_time: .5f} seconds.")

# 계산시간 측정
start_time = time.time()
fibonacci2(40)
print(f"DP Fibonacci Execution: Took {time.time() -
start_time: .5f} seconds.")
```

Fibonacci Execution: Took 30.97598 seconds.
DP Fibonacci Execution: Took 0.00002 seconds.

개선

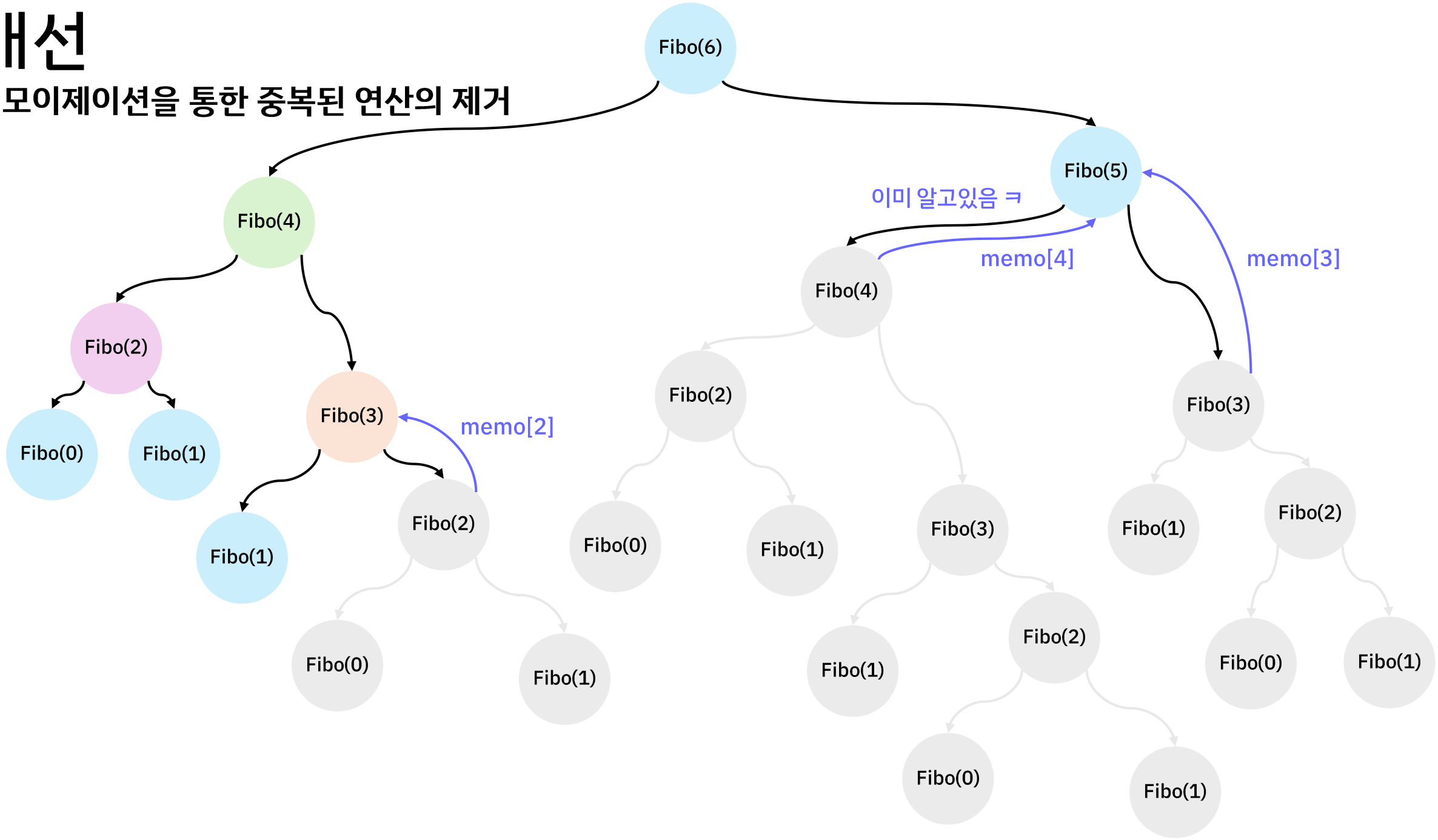
메모이제이션을 통한 중복된 연산의 제거

각 연산의 결과를 memo(ization) 리스트에 저장
연산의 결과가 이미 존재한다면, 바로 반환

```python
# 결과 값 기록을 위한 리스트
memo = [-1 for _ in range(100)]

# 일반적인 피보나치 호출
def fibonacci(n):
    if n <= 1:
        return 0

    return fibonacci(n - 2) + fibonacci(n - 1)

# 동적계획법이 적용된 피보나치 호출
def fibonacci2(n):
    if n <= 1:
        return 0

    if memo[n] != -1:
        return memo[n]

    memo[n] = fibonacci2(n - 2) + fibonacci2(n - 1)
    return memo[n]
```

개선

메모이제이션을 통한 중복된 연산의 제거



정리해서, 동적계획법으로 문제를 풀려면..

1. 재귀를 기반으로 한 완전탐색 풀이를 생각해본다.
2. 중복된 문제를 한 번만 계산하도록 메모이제이션을 적용해본다.

c.f. 재귀를 기반으로 하지 않고, for-loop iteration을 기반으로 한 동적계획법도 존재합니다.

스택 호출이 되는 재귀함수의 특성상 stack overflow 가 발생할 수 있지만,
지금까지 그런 경우는 많게는 1~2번, 코딩테스트 시험에서는 없었습니다.

유형화 되어 있는 동적계획법 종류들

1. LIS; Longest Increasing Subsequence (가장 긴 증가하는 부분 수열)
2. LCS; Longest Common Subsequence (최장 공통 부분 수열)
3. Knapsack (배낭 문제)
4. TSP; Traveling Salesman Problem (외판원 순회)

TIP

1. 재귀호출을 기반으로 하는 동적계획법 풀이를 추천드립니다. (코드가 직관적이고, 디버깅이 용이합니다)
2. Memoization 리스트를 생성할 때, 차원은 1차원이 아니라 N차원으로 확장될 수 있습니다.
   차원의 수를 지정하기 힘들다면, 특정 노드(상태)를 특정할 수 있는 최소 매개변수의 개수로 설정하면 좋습니다.
   보통, 재귀함수의 인자의 개수와 동일하게 차원을 설정하게 됩니다.
3. Fibonacci 처럼 큰 문제를 작은 문제로 나누는 것이 직관적이지 않은 경우가 대다수 입니다.
   때문에 이것을 모델링하는 점화식을 작성하는데 가장 큰 어려움을 겪는데,
   처음부터 최적화된 풀이를 떠올리기보다 brute-force 로 접근하여 점차 개선해나가는 것이 좋습니다 :)
4. 들어가면서 말했던 것처럼 동적계획법은 어렵습니다. 힘들지만 정말 많은 문제를 풀어보면서 감을 잡아야 합니다ㅜ.
5. 나는 동적계획법의 신이 되고 싶다: berlekamp-Massey 알고리즘을 찾아봅시다!
   코드는 매우 어렵지만, 놀랍게도 점화식을 알아서 찾아줍니다

그리디 알고리즘 vs. 동적계획법

문제를 해결하다보면, 특정 문제에 그리디 알고리즘을 사용할 지 DP를 적용시킬지 고민이 될 때가 많습니다.

이 선택의 가장 큰 핵심은 '증명'의 여부입니다.

자세히 말하자면, 그리디 알고리즘은 매 순간의 최선의 선택이 global optimal 한 해가 된다는 것인데,

각 순간의 최선의 선택이 결과적으로 최고의 선택임을 반드시 증명할 수 있어야만 합니다.

예제: #17485 진우의 달 여행 (Large)

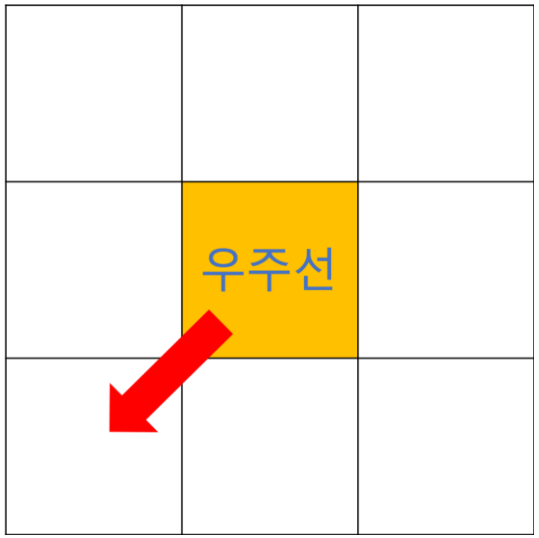
우주비행이 꿈이였던 진우는 음식점 '매일매일싱싱'에서 열심히 일한 결과 달 여행에 필요한 자금을 모두 마련하였다! 지구와 우주사이는 N X M 행렬로 나타낼 수 있으며 각 원소의 값은 우주선이 그 공간을 지날 때 소모되는 연료의 양이다.

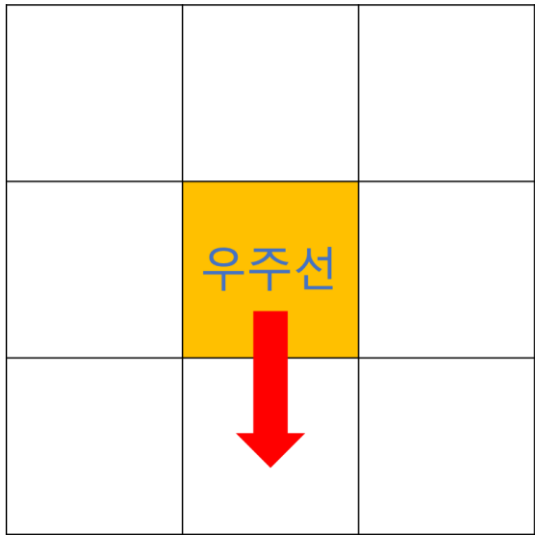
지구			
5	8	5	1
3	5	8	4
9	77	65	5
2	1	5	2
5	98	1	5
4	95	67	58
달			

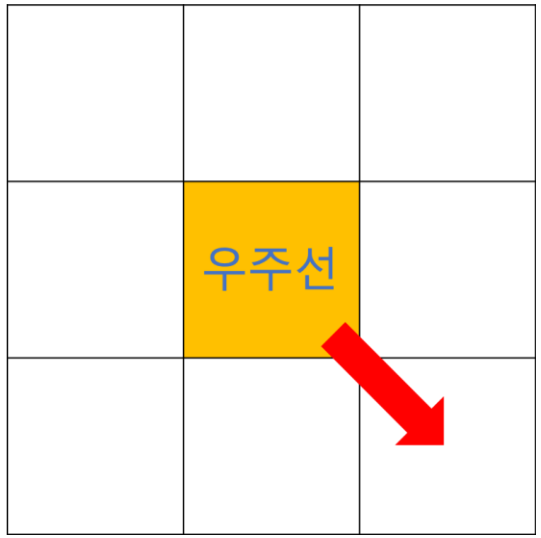
예제: #17485 진우의 달 여행 (Large)

진우는 여행경비를 아끼기 위해 조금 특이한 우주선을 선택하였다. 진우가 선택한 우주선의 특징은 아래와 같다.

1. 지구 -> 달로 가는 경우 우주선이 움직일 수 있는 방향은 아래와 같다.







예제: #17485 진우의 달 여행 (Large)

2. 우주선은 전에 움직인 방향으로 움직일 수 없다. 즉, 같은 방향으로 두 번 연속으로 움직일 수 없다.

진우의 목표는 연료를 최대한 아끼며 지구의 어느 위치에서든 출발하여 달의 어느 위치든 착륙하는 것이다.

최대한 돈을 아끼고 살아서 달에 도착하고 싶은 진우를 위해 달에 도달하기 위해 필요한 연료의 최소값을 계산해 주자.

첫줄에 지구와 달 사이 공간을 나타내는 행렬의 크기를 나타내는 N, M (2 ≤ N, M ≤ 1000)이 주어진다.

다음 N줄 동안 각 행렬의 원소 값이 주어진다. 각 행렬의 원소값은 100 이하의 자연수이다.

문제 접근하기

(완전탐색) 갈 수 있는 모든 방향 고려하기

(최단경로) 앞으로 최단경로를 구성하기 위해서 그 이전까지의 경로도 최단경로여야 한다

최단경로 조건을 만족하는 완전탐색 먼저 구현 또는 생각해보기

지구			
5	8	5	1
3	5	8	4
9	77	65	5
2	1	5	2
5	98	1	5
4	95	67	58
달			

문제 접근하기

(완전탐색) 갈 수 있는 모든 방향 고려하기

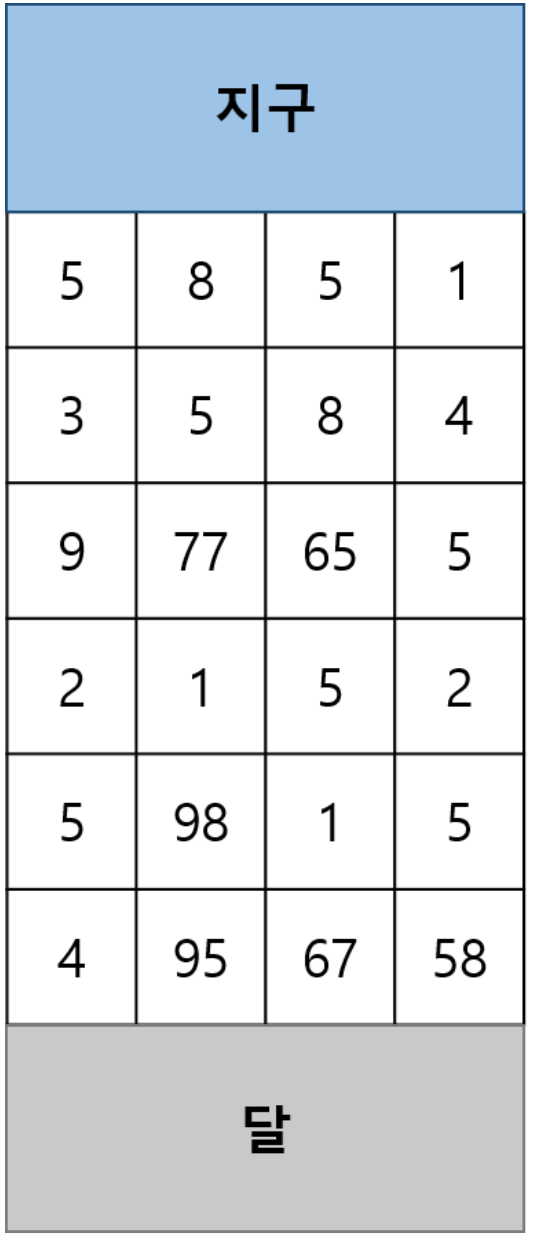
(최단경로) 앞으로 최단경로를 구성하기 위해서 그 이전까지의 경로도 최단경로여야 한다

최단경로 조건을 만족하는 완전탐색 먼저 구현 또는 생각해보기

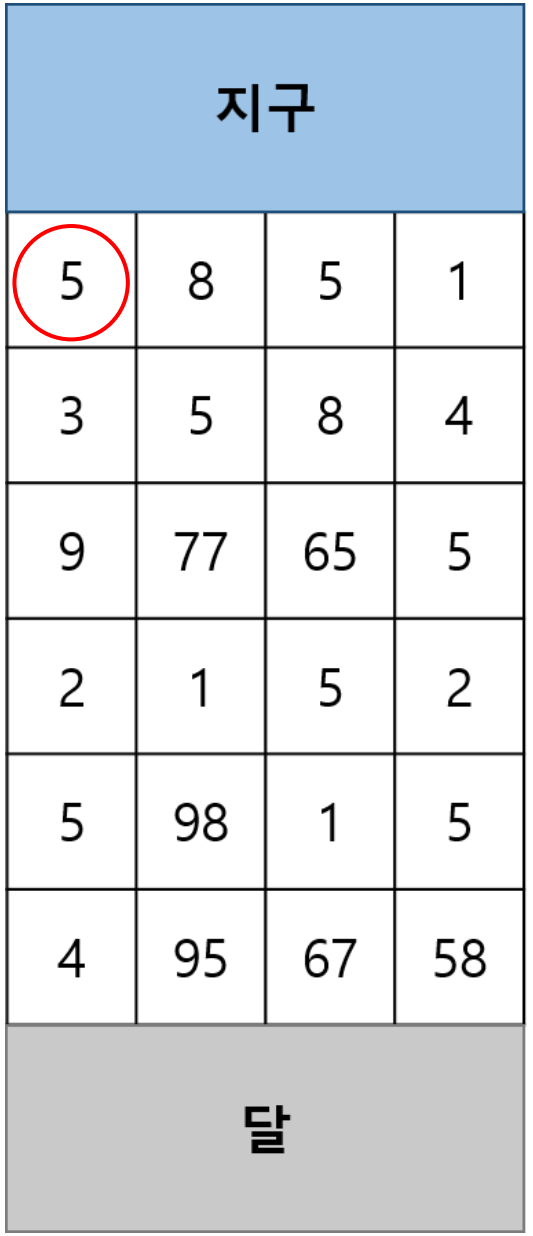
Greedy ? : 현재 바로 앞에 가장 적은 수를 선택한다고 끝까지 최소임이 보장되는가?

지구			
5	8	5	1
3	5	8	4
9	77	65	5
2	1	5	2
5	98	1	5
4	95	67	58
달			

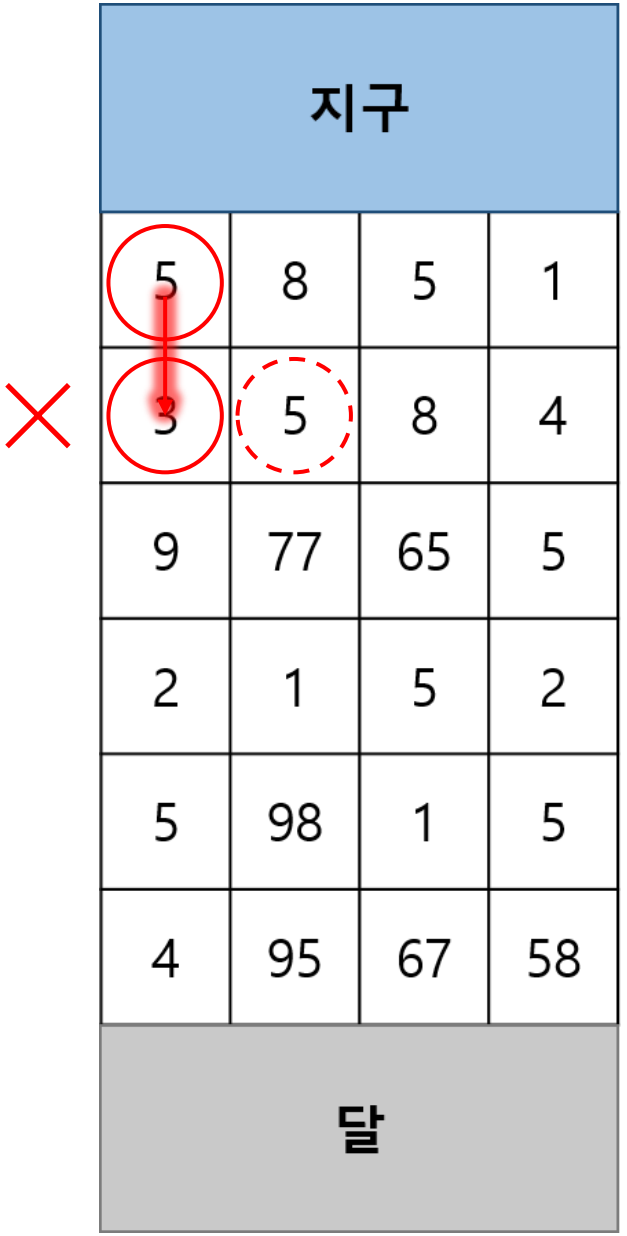
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



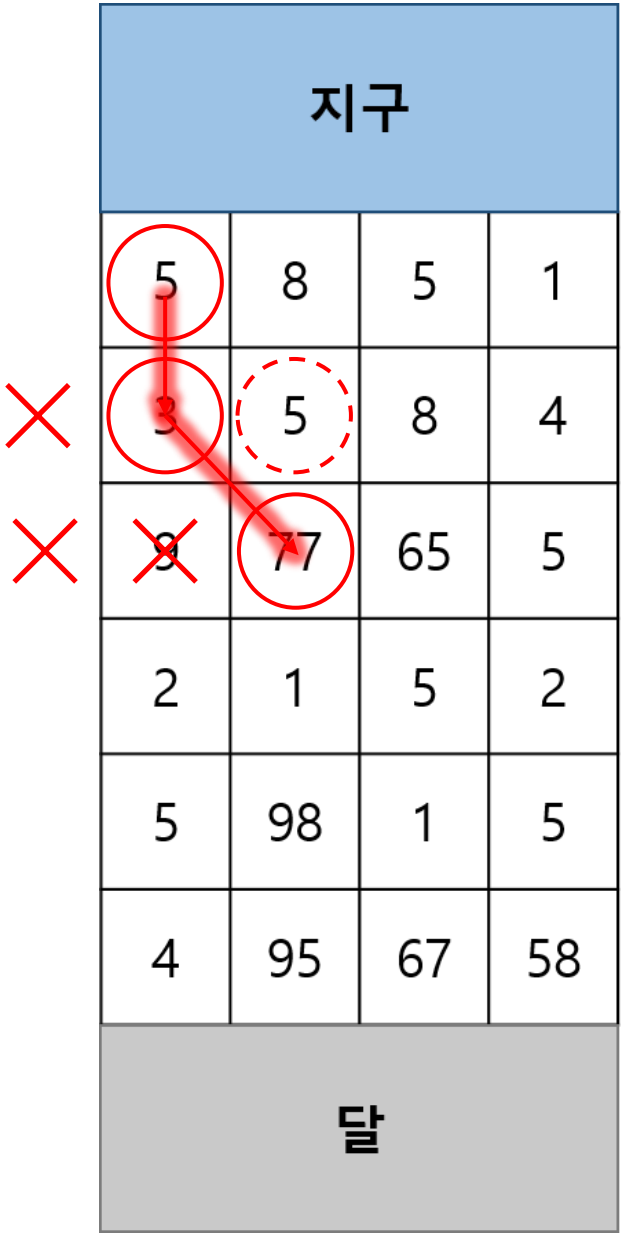
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



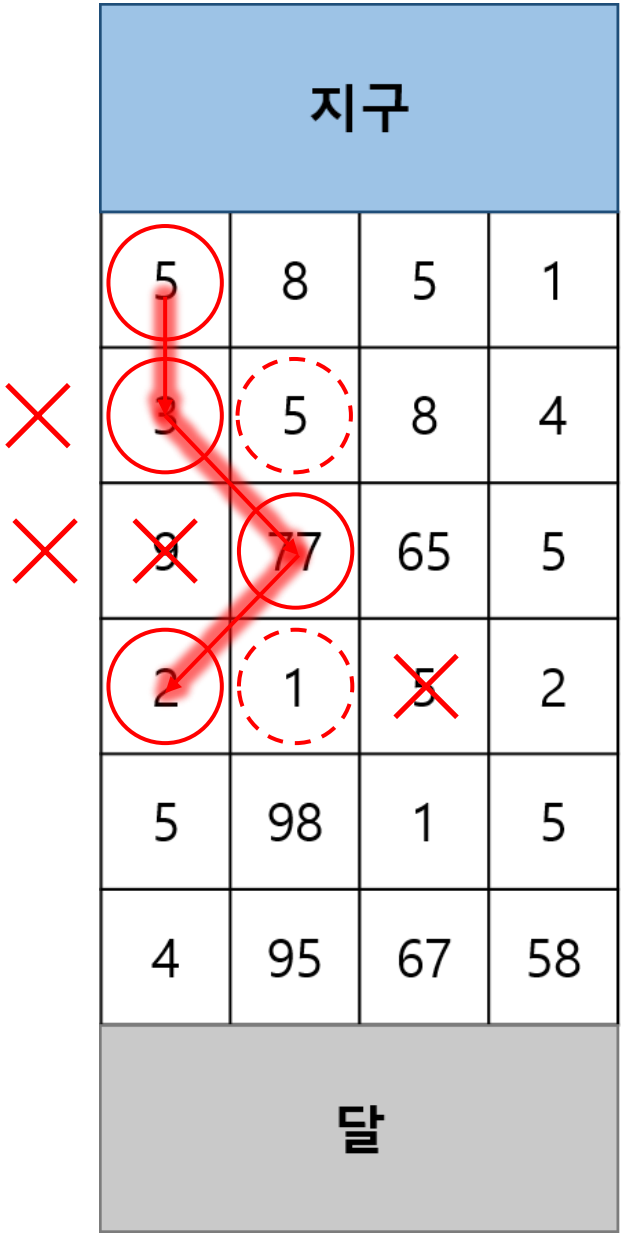
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



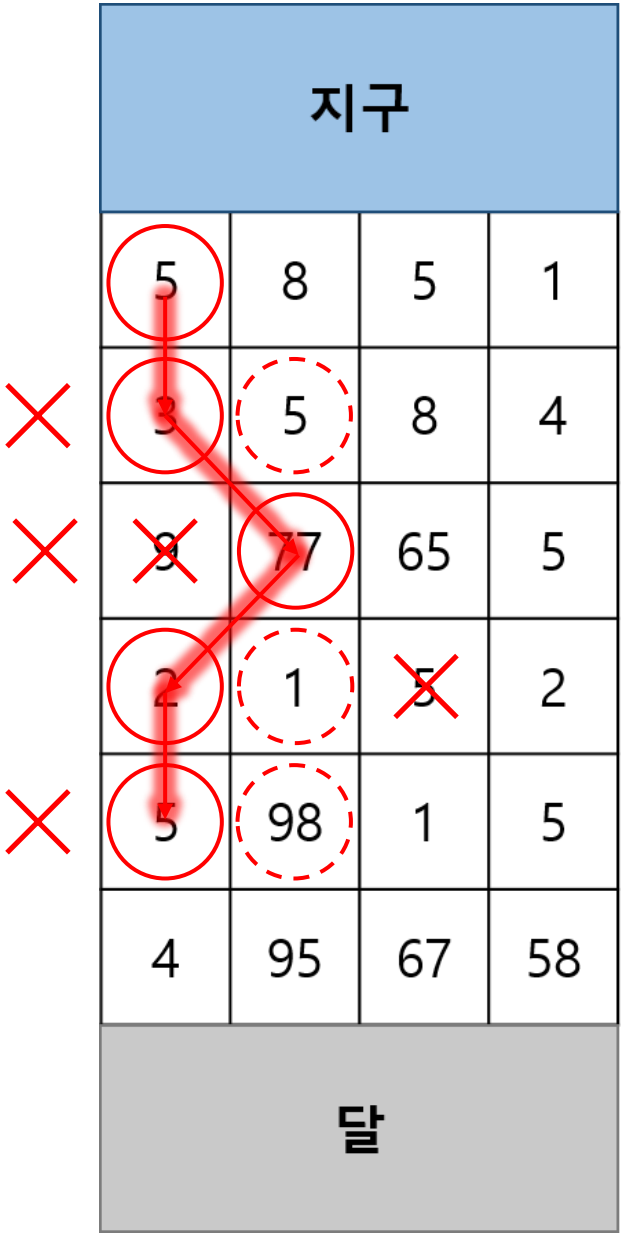
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



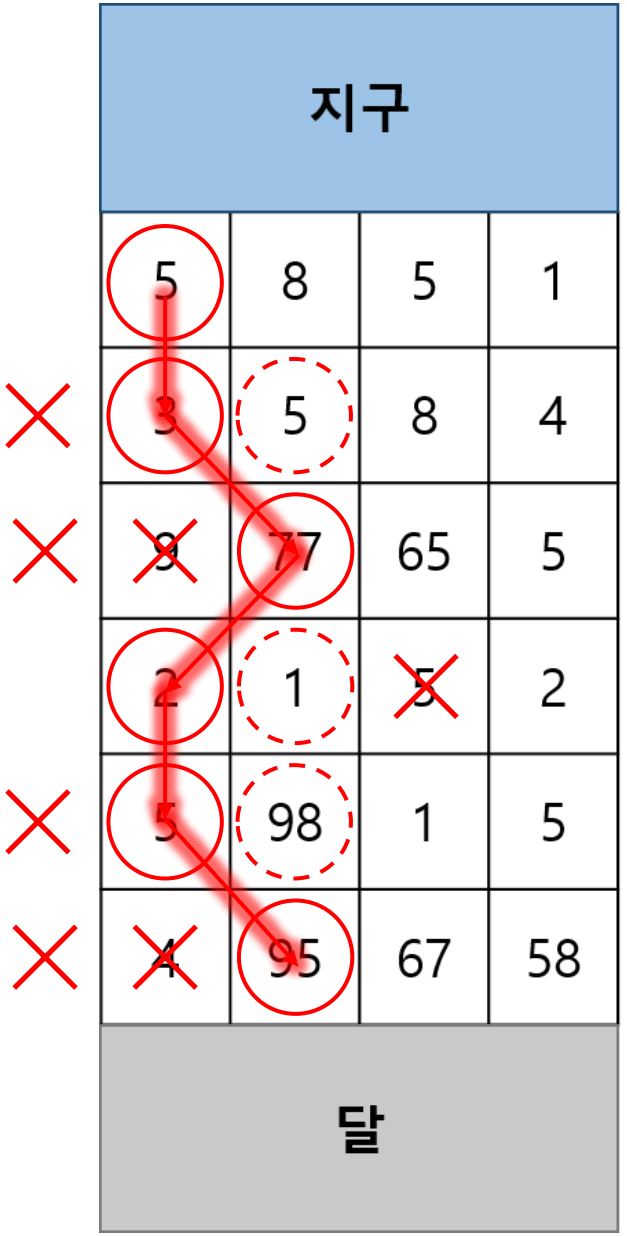
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



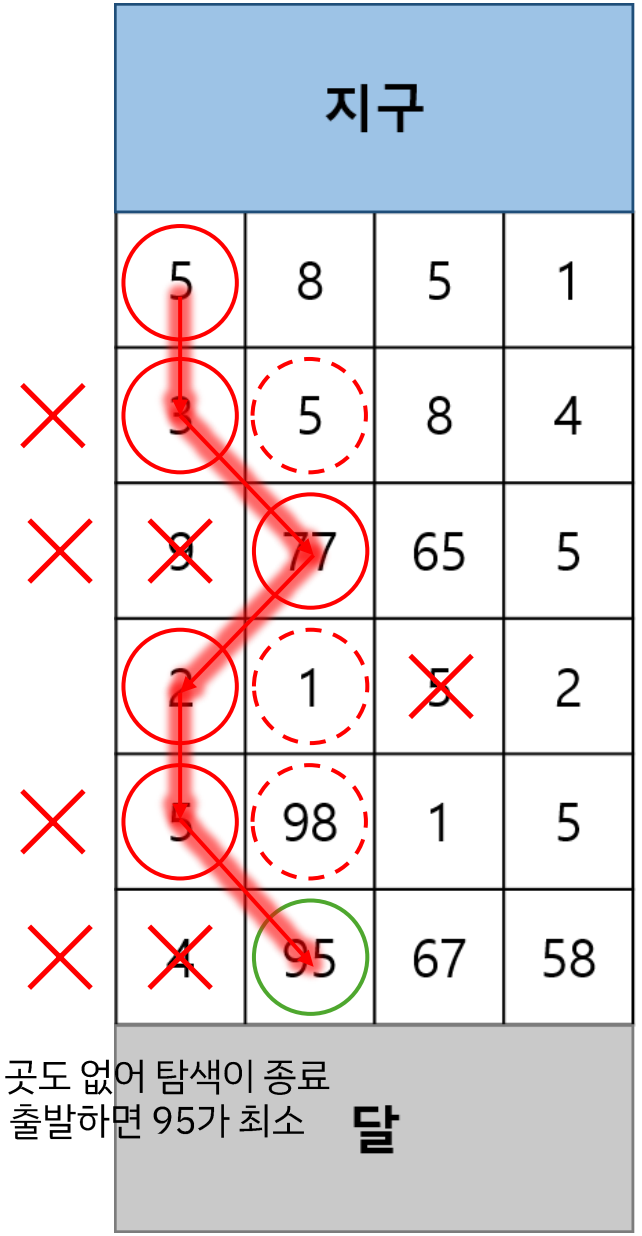
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```



앞으로 갈 곳도 없어 탐색이 종료
=여기서 출발하면 95가 최소

```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```

현재 위치에서 오른쪽 아래로 가게 되었을 때,
가능한 모든 경우의 수를 탐색하였으므로
오른쪽 아래로 간다면 5+95를 반환

앞으로 갈 곳도 없어 탐색이 종료
=여기서 출발하면 95가 최소

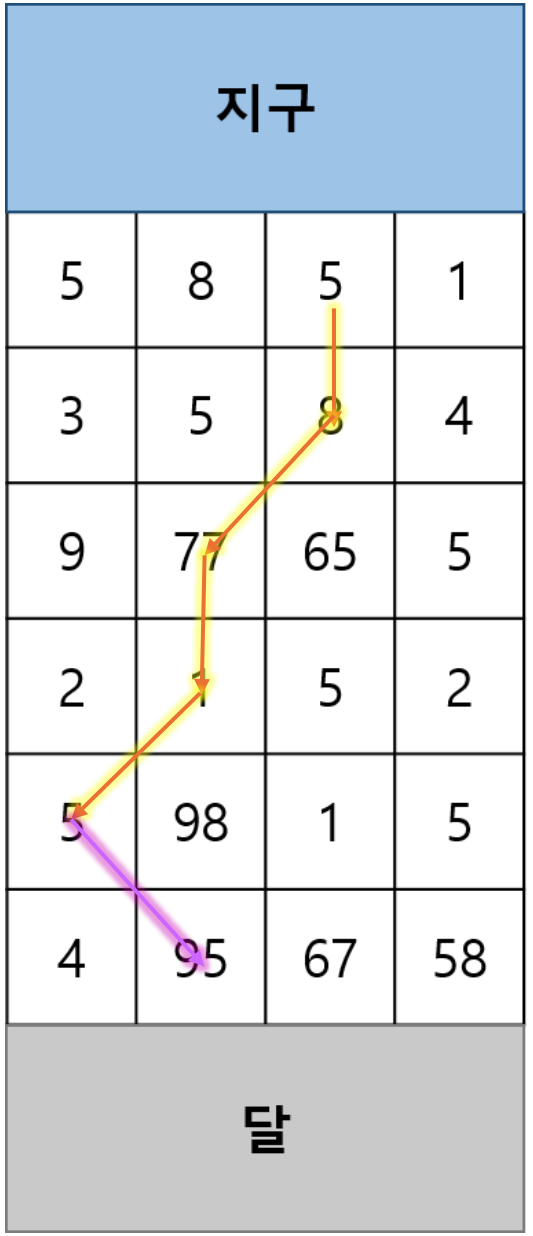
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```

이 경로를 다른 탐색과정 중에서도
확인할 수도 있어 중복이 발생합니다

앞으로 갈 곳도 없어 탐색이 종료
=여기서 출발하면 95가 최소

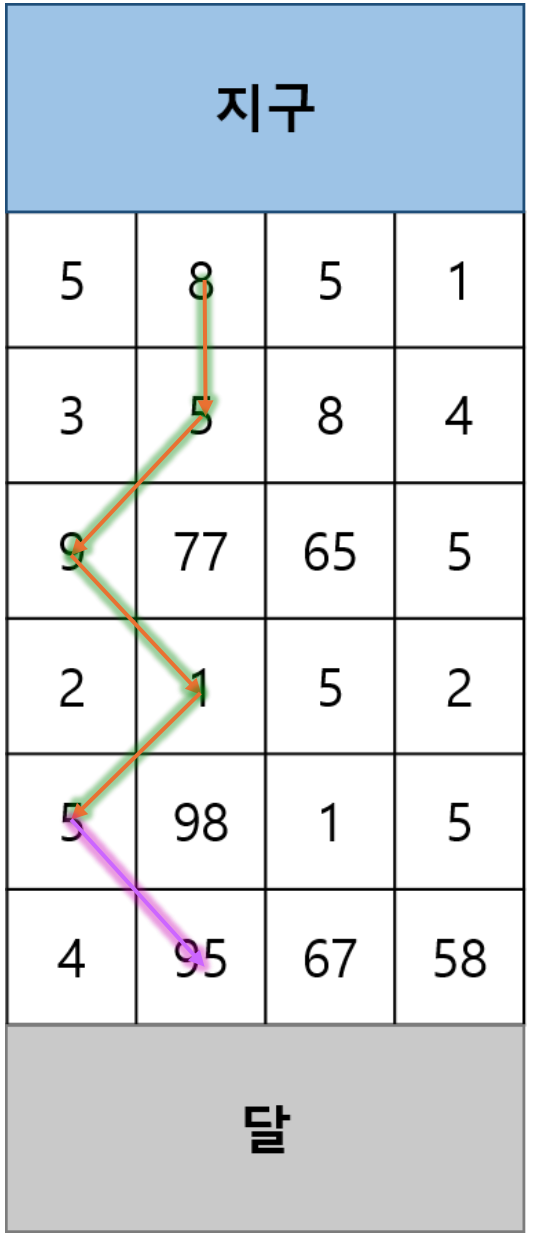
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```

이 경로를 다른 탐색과정 중에서도
확인할 수도 있어 중복이 발생합니다



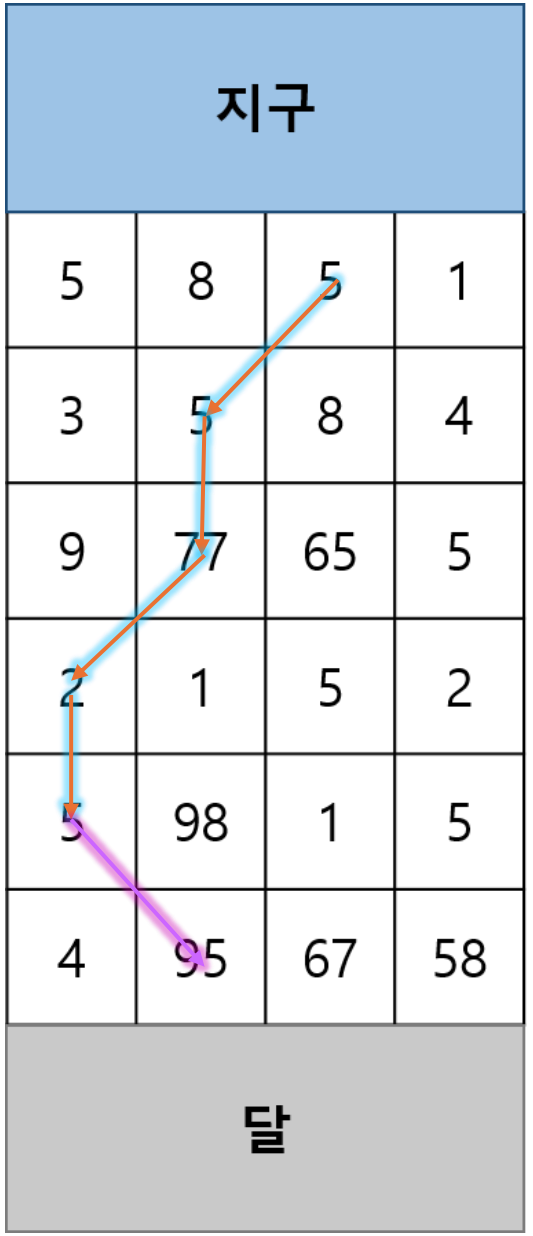
```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```

이 경로를 다른 탐색과정 중에서도
확인할 수도 있어 중복이 발생합니다



```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))

    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))

    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```

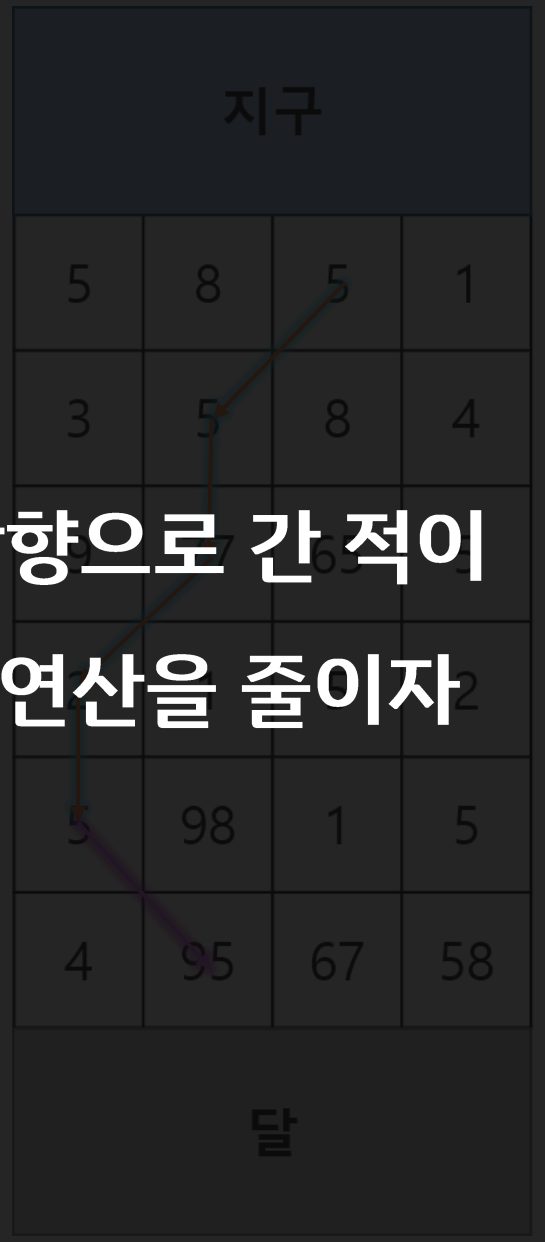
이 경로를 다른 탐색과정 중에서도
확인할 수도 있어 중복이 발생합니다



```python
def solve(grid, row, col, prev_dir=None):
    """
    grid에서 row, col에서 시작했을 때의 최소 cost를 반환해준다.
    """

    # base condition: 마지막 행에 도달
    if row == len(grid) - 1:
        return grid[row][col]

    # 갈 수 있는 방향의 목록
    directions = []

    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and prev_dir != -1:
        directions.append((row + 1, col - 1, -1))
    # 아래로 이동 (0)
    if prev_dir != 0:
        directions.append((row + 1, col, 0))
    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and prev_dir != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = 1e9
    for next_row, next_col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction)
        min_cost = min(min_cost, cost)

    return min_cost


# 최상단 행에서 시작하여 최저 비용을 계산
min_cost = 1e9
for col in range(len(grid[0])):
    cost = solve(grid, 0, col)
    min_cost = min(min_cost, cost)
```

메모이제이션

특정 row, column 에서 direction 방향으로 간 적이 있다면, 그 결과값을 메모하여 불필요 연산을 줄이자



```python
def solve(grid, row, col, last_direction=None, memo=None):
    if row == len(grid) - 1:
        # 마지막 행에 도달했을 경우 현재 셀의 비용을 반환
        return grid[row][col]


    directions = []
    # 아래로 이동 (0)
    if last_direction != 0:
        directions.append((row + 1, col, 0))
    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and last_direction != -1:
        directions.append((row + 1, col - 1, -1))
    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and last_direction != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = float('inf')
    for next_row, col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction, memo)
        min_cost = min(min_cost, cost)

    return min_cos
```

```python
def solve(grid, row, col, last_direction=None, memo=None):
    if row == len(grid) - 1:
        # 마지막 행에 도달했을 경우 현재 셀의 비용을 반환
        return grid[row][col]

    # 메모이제이션 체크
    if (row, col, last_direction) in memo:
        return memo[(row, col, last_direction)]

    directions = []
    # 아래로 이동 (0)
    if last_direction != 0:
        directions.append((row + 1, col, 0))
    # 왼쪽 대각선 아래로 이동 (-1)
    if col > 0 and last_direction != -1:
        directions.append((row + 1, col - 1, -1))
    # 오른쪽 대각선 아래로 이동 (1)
    if col < len(grid[0]) - 1 and last_direction != 1:
        directions.append((row + 1, col + 1, 1))

    min_cost = float('inf')
    for next_row, col, direction in directions:
        cost = grid[row][col] + solve(grid, next_row, next_col, direction, memo)
        min_cost = min(min_cost, cost)

    # 결과를 메모이제이션에 저장
    memo[(row, col, last_direction)] = min_cost

    return min_cos
```

마지막

백준 온라인 저지에서 1800문제 넘게 풀면서 돌이켜보면 그렇게 공부를 효율적으로 하지 못했습니다.

훨씬 짧게 공부하신 분들이 더 빠르게 올라가는 것도 정말 많이 봐왔습니다.

물론, 각자의 속도가 있지만 최대한의 시간낭비를 줄이면서 실력을 키울 수 있는 “제 생각”을 전달드리자면

1. 이미 실력이 있는데도 고민하는 시간이 싫고, `맞았습니다!!` 를 보기 위해 의미 없는 쉬운 문제를 푸는 것 X
2. 문제를 해결하고 나서 끝내기보다 다른 사람의 풀이 꼭 확인해보기 – 생각보다 많은 것을 얻어갈 수 있습니다
3. 문제를 읽자마자 입력부터 받는 코드를 작성하기 위해 키보드를 만지기 보다, 생각이 정리될 때까지 펜을 만지자!
4. 문제 지문 자체에 힌트가 정말 많습니다: 입력의 범위, 예외 조건 등 놓치기 쉬운 정보들을 다 챙겨야합니다.
5. 꾸준히 하는 사람이 제일 무섭습니다

문제 선정

2xN 타일링 : https://www.acmicpc.net/problem/11726

1, 2, 3 더하기 : https://www.acmicpc.net/problem/9095

MORE..?

계단 오르기 : https://www.acmicpc.net/problem/2579

2xN 타일링 2 https://www.acmicpc.net/problem/11727

정수 삼각형 : https://www.acmicpc.net/problem/1932

저는 머리가 좋지 못해 DP를 너무 어려워해서 DP문제만 난이도순으로 정렬하여 100개를 풀어 감을 잡았습니다 :)