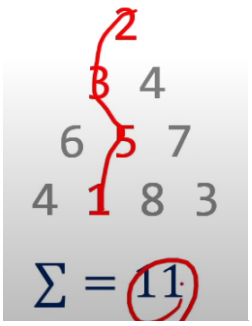
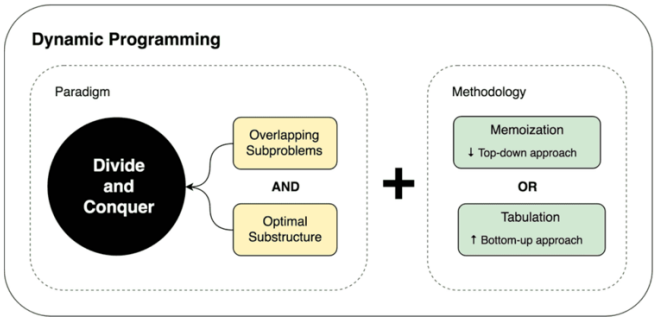
챕터 5. Dynamic Programming (동적 계획법) - (4,5) 삼각형의 최단 경로합 문제

LeetCode 120.Triangle

- 삼각형 배열을 줄테니, 최상단에서 최하단까지 최단 경로합을 구해보자.
  - 삼각형에서 이동은 바로 아래, 바로 아래의 오른쪽으로만 이동 가능

```
vector<vector<int>> triangle(n);
```

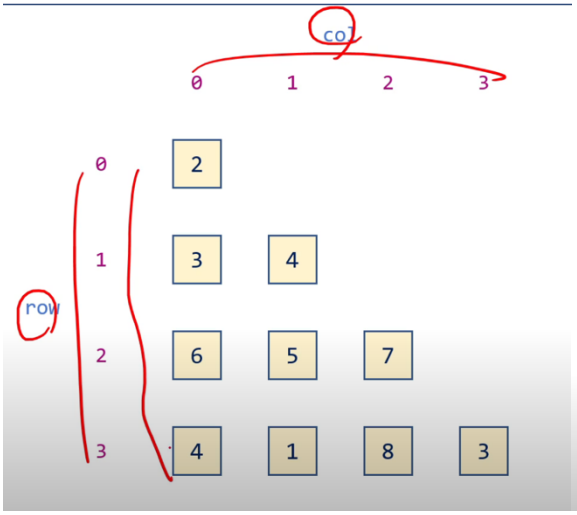


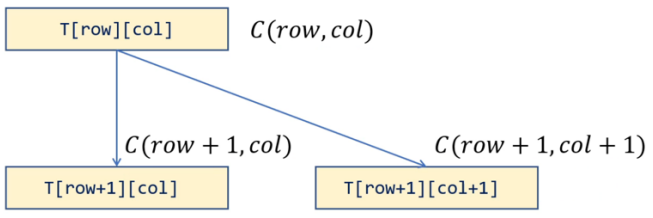


단계별 문제해결:

- Step 1: Divide-and-Conquer
  - 재귀함수를 사용하여 문제 해결
- Step 2: Dyn. Prog. 1 Memoization
  - 재귀함수에서 중복 부분문제를 해결
- Step 3: Dyn. Prog. 2 Tabulation
  - 재귀 관계를 Bottom-Up 방식으로 해결

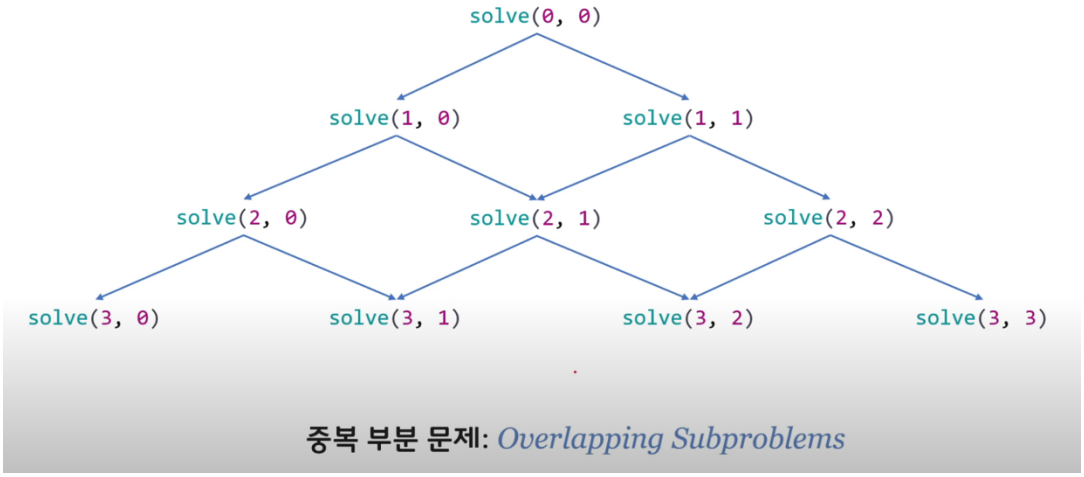
T : n X n matrix





$$C(row, col) = T[row][col] + \min(C(row+1, col), C(row+1, col+1))$$

$$C(n, col) = 0$$



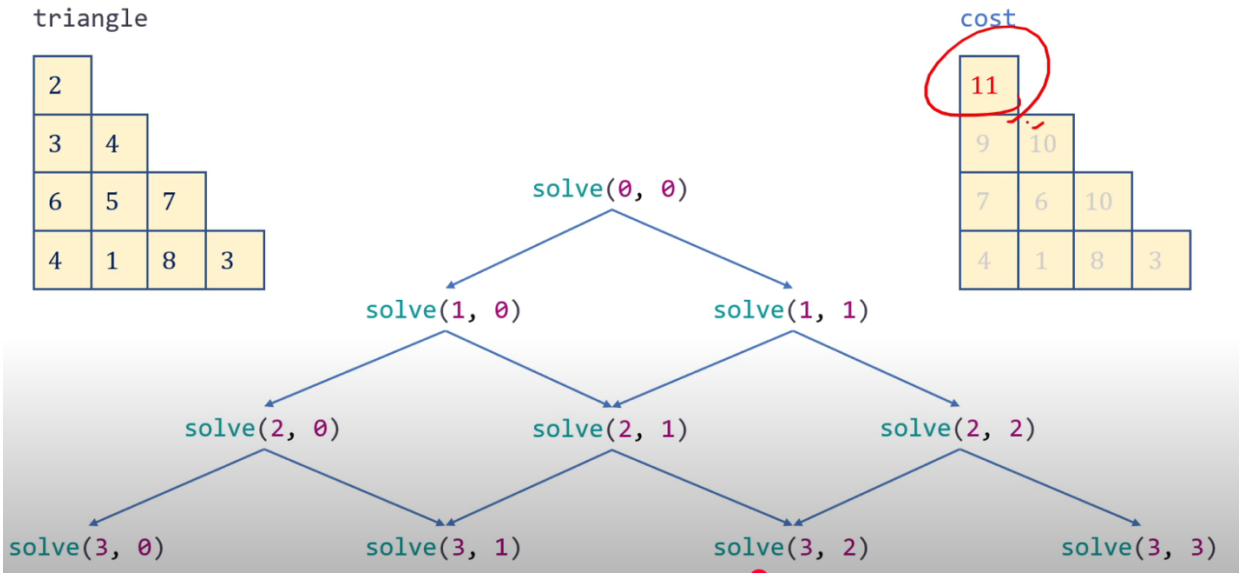
중복 부분 문제: *Overlapping Subproblems*

중복 부분 문제를 어떻게 해결할 것인가

- Step 2: **동적계획법** 1. *Memoization*

```
def solve(row, col, triangle, cost):
    if row == len(triangle):
        return 0
    elif cost[row][col] == MIN:
        cost[row][col] = triangle[row][col] + \
                         min(solve(row + 1, col, triangle, cost), \
                             solve(row + 1, col + 1, triangle, cost))
    return cost[row][col]

class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        cost = [[MIN for j in range(len(triangle[i]))] for i in range(len(triangle))]
        return solve(0, 0, triangle, cost)
```



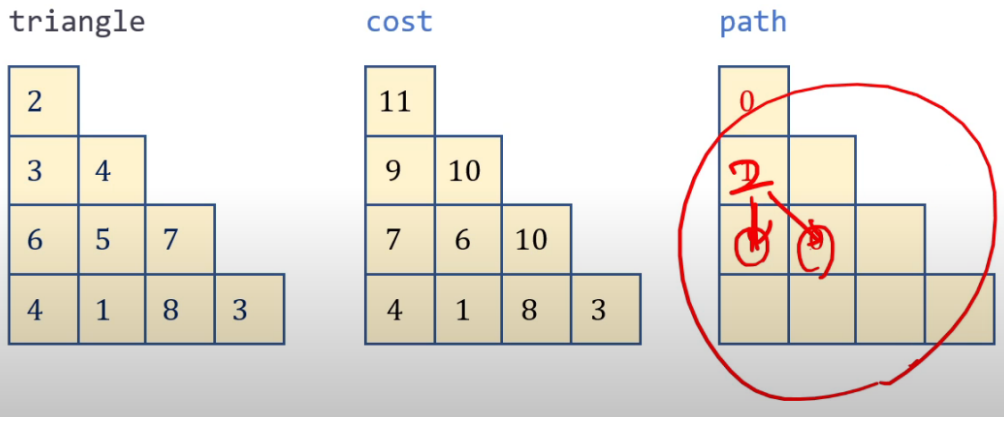
- Step 2: 동적계획법 2. *Tabulation*

Bottom-up

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        for row in range(len(triangle) - 2, -1, -1):
            for col in range(len(triangle[row])):
                triangle[row][col] += min(triangle[row + 1][col], \
                                          triangle[row + 1][col + 1])
        return triangle[0][0]
```

- 더 생각해 볼 문제:
  - 공간 복잡도: *Space* Complexity
    - *cost* 배열을 전부 사용할 필요가 있을까?
    - $O(n)$의 공간 복잡도로 구현하려면?
  - 만약, 최단 경로합을 구성하는 경로의 개수를 출력하라고 하면?

- 더 생각해 볼 문제:
  - 만약, 최단 경로합을 구성하는 경로를 출력하라고 하면?
    - 별도의 삼각형 배열을 사용해서 선택한 방향을 기록



```cpp
// Triangle Path (1): Find Optimal Value

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int path3(int y, int x, const vector<vector<int>>& triangle, vector<vector<int>>& cache) {
    int n = triangle.size();

    // 맨 아래에 도달했을 경우
    if (y == n - 1)
        return triangle[y][x];

    // 이미 계산한 경우
    if (cache[y][x] != -1)
        return cache[y][x];

    // 아래 또는 아래 오른쪽 중 큰 값 선택
    cache[y][x] = triangle[y][x] + max(
        path3(y + 1, x, triangle, cache),
        path3(y + 1, x + 1, triangle, cache)
    );

    return cache[y][x];
}

int main() {
    int c;
    cin >> c;  // 테스트 케이스 수

    while (c--) {
        int n;
        cin >> n;

        vector<vector<int>> triangle(n);
        vector<vector<int>> cache(n, vector<int>(n, -1));

        for (int i = 0; i < n; i++) {
            triangle[i].resize(i + 1);
            for (int j = 0; j <= i; j++) {
                cin >> triangle[i][j];
            }
        }

        cout << path3(0, 0, triangle, cache) << endl;
    }

    return 0;
}

```

```cpp
// Triangle Path (1): Find Optimal Value

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int path3(int y, int x, const vector<vector<int>>& triangle, vector<vector<int>>& cache) {
    int n = triangle.size();

    // 맨 아래에 도달했을 경우
    if (y == n - 1)
        return triangle[y][x];

    // 이미 계산한 경우
    if (cache[y][x] != -1)
        return cache[y][x];

    // 아래 또는 아래 오른쪽 중 큰 값 선택
    cache[y][x] = triangle[y][x] + max(
        path3(y + 1, x, triangle, cache),
        path3(y + 1, x + 1, triangle, cache)
    );

    return cache[y][x];
}

int main() {
    int c;
    cin >> c;  // 테스트 케이스 수

    while (c--) {
        int n;
        cin >> n;

        vector<vector<int>> triangle(n);
        vector<vector<int>> cache(n, vector<int>(n, -1));

        for (int i = 0; i < n; i++) {
            triangle[i].resize(i + 1);
            for (int j = 0; j <= i; j++) {
                cin >> triangle[i][j];
            }
        }

        cout << path3(0, 0, triangle, cache) << endl;
    }

    return 0;
}

```