Week9 - DP(2)

#1890. 점프

문제

N×N 게임판에 수가 적혀져 있다. 이 게임의 목표는 가장 왼쪽 위 칸에서 가장 오른쪽 아래 칸으로 규칙에 맞게 점프를 해서 가는 것이다.

각 칸에 적혀있는 수는 현재 칸에서 갈 수 있는 거리를 의미한다. 반드시 오른쪽이나 아래쪽으로만 이동해야 한다. 0은 더 이상 진행을 막는 종착점이며, 항상 현재 칸에 적혀있는 수만큼 오른쪽이나 아래로 가야 한다. 한 번 점프를 할 때, 방향을 바꾸면 안 된다. 즉, 한 칸에서 오른쪽으로 점프를 하거나, 아래로 점프를 하는 두 경우만 존재한다.

가장 왼쪽 위 칸에서 가장 오른쪽 아래 칸으로 규칙에 맞게 이동할 수 있는 경로의 개수를 구하는 프로그램을 작성하시오.

⇒ N×N 게임판에서 맨 위, 가장 왼쪽부터 맨 아래, 가장 오른쪽까지 이동할 수 있는 경로 개수 계산

입력

첫째 줄에 게임 판의 크기 N (**4 ≤ N ≤ 100**)이 주어진다. 그 다음 N개 줄에는 각 칸에 적혀져 있는 수가 N개씩 주어진다. 칸에 적혀있는 수는 0보다 크거나 같고, 9보다 작거나 같은 정수이며, 가장 오른쪽 아래 칸에는 항상 0이 주어진다.

출력

가장 왼쪽 위 칸에서 가장 오른쪽 아래 칸으로 문제의 규칙에 맞게 갈 수 있는 경로의 개수를 출력한다. 경로의 개수는 $2^{63} - 1$보다 작거나 같다.

예제 입력

```
4
2 3 3 1
1 2 1 3
1 2 3 1
3 1 1 0
```

```
3
```

힌트

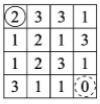


그림 1

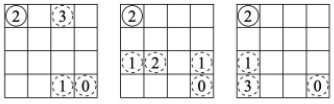


그림 2

코드

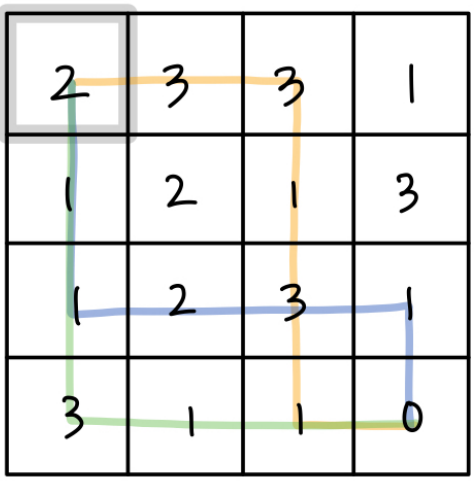
```
n = int(input())
board = [list(map(int, input().split())) for _ in range(n)]

results = [[0]*n for _ in range(n)]
results[0][0] = 1

for x in range(n):
    for y in range(n):
        if (x, y) == (n-1, n-1): # 도착 지점
            continue
        move_cnt = board[x][y] # 이동 가능한 칸

        # 이동 경우의 수: 아래, 오른쪽
        down = x + move_cnt
        right = y + move_cnt

        if down < n: # 게임판 내 좌표
            results[down][y] += results[x][y]
        if right < n:
            results[x][right] += results[x][y]
print(results[n-1][n-1])
```

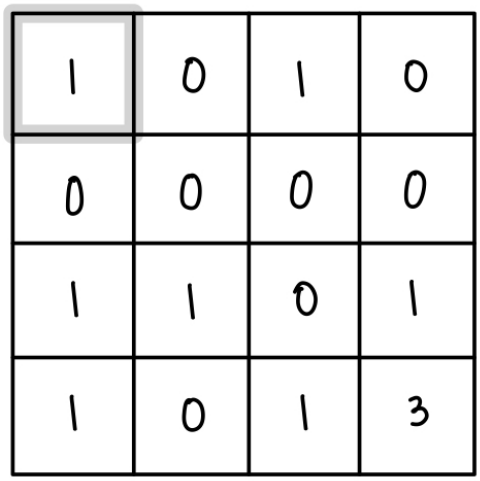
접근 방식

- 거쳐야 할 지점 A, B이면서 C칸까지 가야할 때, 경우의 수

= A칸까지의 경우의 수 + B칸까지의 경우의 수

- (0,0)부터 (n-1, n-1) 모든 좌표 확인





  - 현재 → 오른쪽, 아래
  - (0, 0) → (0, 2) = 1, (2, 0) = 1
  - (0, 2) → (0, 5) = X, (3, 2) = 1
  - (1, y) = 0
  - (2, 0) → (2, 1) = 1, (3, 0) = 1
  - (2, 1) → (2, 3) = 1, (3, 1) = X
  - (2, 3) → (2, 4) = X, **(3, 3)** = 1
  - (3, 0) → **(3, 3)** = 1+1, (6, 0) = X
  - (3, 2) → **(3, 3)** = 2+1, (4, 2) = X

    ⇒ (3, 3) = 3: (3, 3)에 도착하는 경로마다 dp[n-1][n-1] += 1

- 점화식
  - 이동할 경로가 오른쪽인 경우,

    dp[x][y+board[x][y]] = dp[x][y+board[x][y]] + dp[x][y]

    → y+board[x][y] = y+(이동 가능한 칸 수) = 오른쪽
  - 이동할 경로가 아래인 경우,

```
dp[x+board[x][y]][y] = dp[x+board[x][y]][y] + dp[x][y]
```

→ x+board[x][y] = x+(이동 가능한 칸 수) = 아래

#2156. 포도주 시식

문제

효주는 포도주 시식회에 갔다. 그 곳에 갔더니, 테이블 위에 다양한 포도주가 들어 있는 포도주 잔이 일렬로 놓여 있었다. 효주는 포도주 시식을 하려고 하는데, 여기에는 다음과 같은 두 가지 규칙이 있다.

1. 포도주 잔을 선택하면 그 잔에 들어있는 포도주는 모두 마셔야 하고, 마신 후에는 원래 위치에 다시 놓아야 한다.
2. 연속으로 놓여 있는 3잔을 모두 마실 수는 없다.

효주는 될 수 있는 대로 많은 양의 포도주를 맛보기 위해서 어떤 포도주 잔을 선택해야 할지 고민하고 있다. 1부터 n까지의 번호가 붙어 있는 n개의 포도주 잔이 순서대로 테이블 위에 놓여 있고, 각 포도주 잔에 들어있는 포도주의 양이 주어졌을 때, 효주를 도와 가장 많은 양의 포도주를 마실 수 있도록 하는 프로그램을 작성하시오.

예를 들어 6개의 포도주 잔이 있고, 각각의 잔에 순서대로 6, 10, 13, 9, 8, 1 만큼의 포도주가 들어 있을 때, 첫 번째, 두 번째, 네 번째, 다섯 번째 포도주 잔을 선택하면 총 포도주 양이 33으로 최대로 마실 수 있다.

⇒ n개의 포도주 잔에서 연속으로 놓인 3잔을 마실 수 없을 때, 최대 마실 수 있는 양

입력

첫째 줄에 포도주 잔의 개수 n이 주어진다. (1 ≤ n ≤ 10,000) 둘째 줄부터 n+1번째 줄까지 포도주 잔에 들어있는 포도주의 양이 순서대로 주어진다. 포도주의 양은 1,000 이하의 음이 아닌 정수이다.

출력

첫째 줄에 최대로 마실 수 있는 포도주의 양을 출력한다.

예제 입력

```
6
6
10
13
9
8
1
```

예제 출력

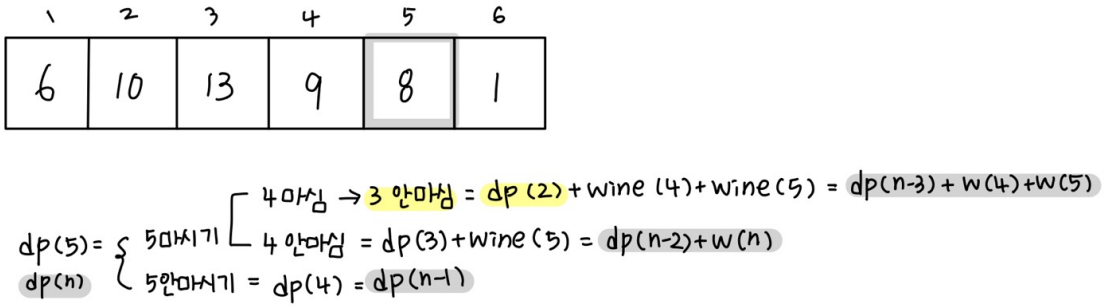
```
33
```

코드

```
n = int(input())
wines = [int(input()) for _ in range(n)]

dp = [0]*n
dp[0] = wines[0]

# 포도주 잔의 개수가 2개 이상인 경우
if n > 1:
    dp[1] = wines[0]+wines[1]

# 포도주 잔의 개수가 3개 이상인 경우
if n > 2:
    dp[2] = max(dp[1], wines[1]+wines[2], dp[0]+wines[2])

    for i in range(3, n):
        dp[i] = max(dp[i-1], dp[i-2]+wines[i], dp[i-3]+wines[i]+wines[i-1])

print(dp[n-1])
```

접근 방식

- 포도주 잔의 개수가 1개 혹은 2개일 경우, index Error가 발생하지 않게 조건처리
- 예제



⇒ 점화식: $a_n = max(a_{n-1}, a_{n-2} + w_n, a_{n-3} + w_4 + w_5)$

$a_1 = w_1,\ a_2 = w_1 + w_2,\ a_3 = max(w_1 + w_3, w_2 + w_3, w_1)$

14728. 벼락치기

- 배낭 문제(Knapsack Problem)
  - Fraction Knapsack Problem: 무게나 가치를 소수점 단위로 나눌 수 있는 문제
  - 0-1 Knapsack Problem: 무게와 가치를 0과 1로 나눌 수 없는 문제

① 브루트 포스

배낭	1	2	...	n
	False	False	...	False
	True	True	...	True

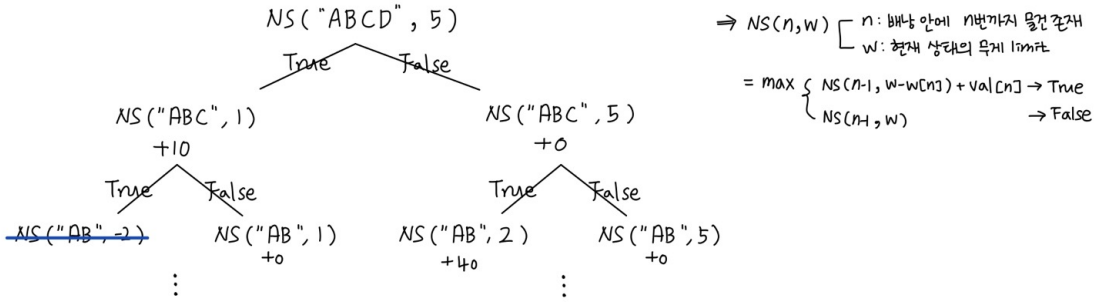
- n개의 배낭에 대해 가능한 경우의 수: 2*2*...*2 = 2^n → $O(2^n)$

② DP

	A	B	C	D
value	30	20	40	10
weight	1	2	3	4

→ 배낭 최대 무게 = 5kg

- 점화식



⇒ NS(n, w)

= max(NS(n-1, w), NS(n-1, w-weight[n])+val[n]), **w-weight[n] ≥ 0**

= NS(n-1, w), **w-weight[n] < 0** # 무게 한도를 넘은 경우, 해당 물건 제외

: 물건을 n개까지 담을 수 있고 무게 한도가 w일 때, 얻을 수 있는 최대 이익

문제

ChAOS(Chung-ang Algorithm Organization and Study) 회장이 되어 일이 많아진 준석이는 시험기간에도 일 때문에 공부를 하지 못하다가 시험 전 날이 되어버리고 말았다. 다행히도 친절하신 교수님께서 아래와 같은 힌트를 시험 전에 공지해 주셨다. 내용은 아래와 같다.

1. 여러 단원을 융합한 문제는 출제하지 않는다.
2. 한 단원에 한 문제를 출제한다. 단, 그 단원에 모든 내용을 알고 있어야 풀 수 있는 문제를 낼 것이다.

이런 두가지 힌트와 함께 각 단원 별 배점을 적어 놓으셨다. 어떤 단원의 문제를 맞추기 위해서는 그 단원의 예상 공부 시간만큼, 혹은 그보다 더 많이 공부하면 맞출 수 있다고 가정하자. 이때, ChAOS 회장 일로 인해 힘든 준석이를 위하여 남은 시간 동안 공부해서 얻을 수 있는 최대 점수를 구하는 프로그램을 만들어 주도록 하자.

⇒ 시험 전까지 남은 시간이 주어지고 각 단원에서 필요로 하는 예상 공부 시간과 해당 단원의 문제를 맞혔을 때 배점이 주어졌다면 남은 시간동안 공부해서 얻을 수

있는 최대 점수 계산

입력

첫째 줄에는 이번 시험의 단원 개수 N(1 ≤ N ≤ 100)과 시험까지 공부 할 수 있는 총 시간 T(1 ≤ T ≤ 10000)가 공백을 사이에 두고 주어진다.

둘째 줄부터 N 줄에 걸쳐서 각 단원 별 예상 공부 시간 K(1 ≤ K ≤ 1000)와 그 단원 문제의 배점 S(1 ≤ S ≤ 1000)가 공백을 사이에 두고 주어진다.

출력

첫째 줄에 준석이가 얻을 수 있는 최대 점수를 출력한다.

예제 입력

```
3 310
50 40
100 70
200 150
```

예제 출력

```
220
```

코드

```
n, t = map(int, input().split())
chapters = [list(map(int, input().split())) for _ in range(n)]

v = [0]
w = [0]
for chap in chapters:
    v.append(chap[1])
    w.append(chap[0])

ns = [[0] * (t+1) for _ in range(n+1)]
for i in range(1, n+1):
    for j in range(1, t+1):
        if j-w[i] >= 0: # True - 현재 단원을 공부할 시간적 여유가 있을 때
            ns[i][j] = max(ns[i-1][j], ns[i-1][j-w[i]]+v[i])
                        # 현재 단원 공부X, 현재 단원 공부O할 때 얻는 배점
```

```
        else: # False - 현재 단원을 공부할 시간적 여유가 없을 때, 이전 시간
            ns[i][j] = ns[i-1][j]
print(ns[n][t])
```

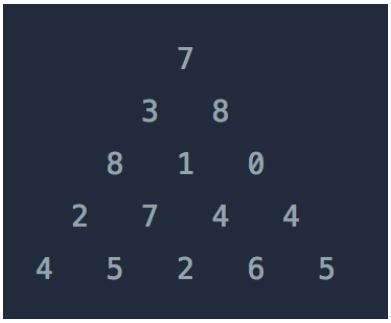
접근 방식

- 0-1 knapsack problem

	1단원	2단원	3단원
value(배점)	40	70	150
weight(시간)	50	100	200

#정수 삼각형 - 프로그래머스

문제



위와 같은 삼각형의 꼭대기에서 바닥까지 이어지는 경로 중, 거쳐간 숫자의 합이 가장 큰 경우를 찾아보려고 합니다. 아래 칸으로 이동할 때는 대각선 방향으로 한 칸 오른쪽 또는 왼쪽으로만 이동 가능합니다. 예를 들어 3에서는 그 아래칸의 8 또는 1로만 이동이 가능합니다.

삼각형의 정보가 담긴 배열 triangle이 매개변수로 주어질 때, 거쳐간 숫자의 최댓값을 return 하도록 solution 함수를 완성하세요.

제한사항

- 삼각형의 높이는 1 이상 500 이하입니다.
- 삼각형을 이루고 있는 숫자는 0 이상 9,999 이하의 정수입니다.

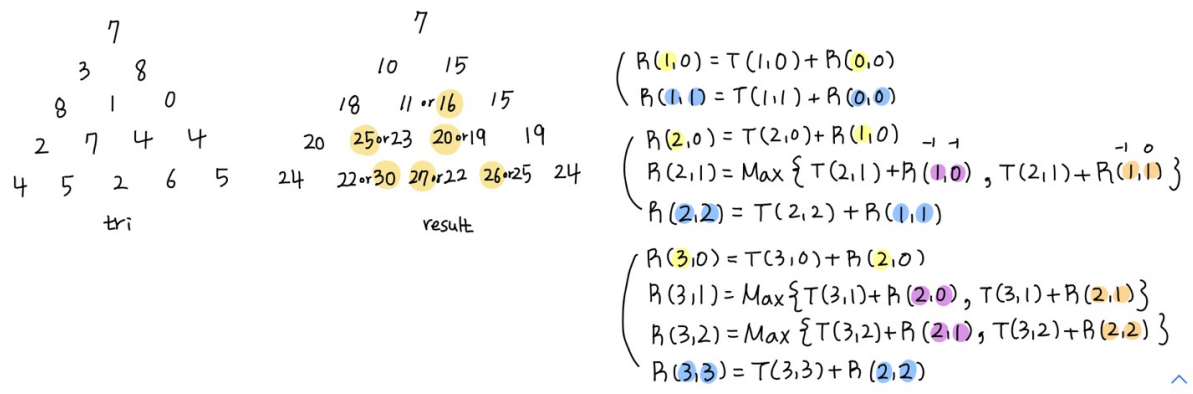
입출력 예

triangle	result
[[7], [3, 8], [8, 1, 0], [2, 7, 4, 4], [4, 5, 2, 6, 5]]	30

코드

```
def solution(triangle):
    answer = 0
    results = [[0]*len(row) for row in triangle]
    results[0] = triangle[0]

    for y in range(1, len(triangle)):
        end_x = len(triangle[y])
        for x in range(end_x):
            if x == 0: # 삼각형 가장자리(왼쪽)
                results[y][x] = results[y-1][x] + triangle[y][x]
            elif x == (end_x-1): # 삼각형 가장자리(오른쪽)
                results[y][x] = results[y-1][x-1] + triangle[y][x]
            else: # 삼각형 내부
                results[y][x] = max(results[y-1][x-1] + triangle[y][x],
                                    results[y-1][x] + triangle[y][x])
            if answer < results[y][x]: # 최대값
                answer = results[y][x]

    return answer
```

접근 방식



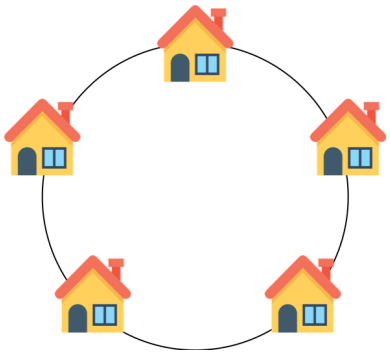
- T: 좌표 값, R: DP 배열
- 삼각형 가장자리
  - 왼쪽: R[i][0] = R[i-1][1] + T[i][1], 1 ≤ i
  - 오른쪽: R[i][j] = R[i-1][j-1] + T[i][j], i=j
- 삼각형 내부

  : R[i][j] = max(R[i-1][j]+T[i][j],R[i-1][j-1]+T[i][j])

#도둑질 - 프로그래머스

문제

도둑이 어느 마을을 털 계획을 하고 있습니다. 이 마을의 모든 집들은 아래 그림과 같이 동그랗게 배치되어 있습니다.



각 집들은 서로 인접한 집들과 방범장치가 연결되어 있기 때문에 인접한 두 집을 털면 경보가 울립니다.

각 집에 있는 돈이 담긴 배열 money가 주어질 때, 도둑이 훔칠 수 있는 돈의 최댓값을 return 하도록 solution 함수를 작성하세요.

⇒ 집이 동그랗게 배치되어 있을 때, 서로 인접한 집을 피해 돈을 훔친다면 얻을 수 있는 최대 돈 계산

제한사항

- 이 마을에 있는 집은 3개 이상 1,000,000개 이하입니다.
- money 배열의 각 원소는 0 이상 1,000 이하인 정수입니다.

입출력 예

money	return
[1, 2, 3, 1]	4

코드

```
def solution(money):
    n = len(money)
    case1 = [0]*n
    case2 = [0]*n

    # 1번집 도둑질
    case1[0] = money[0]
    case1[1] = max(case1[0], money[1])
    for i in range(2, n-1): # 마지막집은 도둑질 불가능 - 1번집과 인접
        case1[i] = max(case1[i-1], case1[i-2]+money[i])

    # n번집 도둑질
    case2[1] = max(case2[0], money[1])
    for i in range(2, n):
        case2[i] = max(case2[i-1], case2[i-2]+money[i])

    answer = max(case1[n-2], case2[n-1])

    return answer
```

접근 방식

- 1번 집과 마지막 n번 집은 인접 ∴ 둘 중 하나만 도둑질
  - case1: 1번 집 도둑질
    - case1[1] = money[1]
    - case1[2] = max(case1[1], money[2])
    - i번째 집에 대한 경우의 수

      i) (i-1)번째 집을 턴 경우(i번째 도둑질 불가)

      ii) (i-2)번째 집을 털고 i번째 집 도둑질

      ⇒ case1[i] = max(case1[i-1], case1[i-2]+money[i])
  - case2: n번 집 도둑질
    - case2[1] = 0
    - case2[2] = max(case2[1], money[2])
    - i번째 집에 대한 경우의 수

      i) (i-1)번째 집을 턴 경우(i번째 도둑질 불가)

      ii) (i-2)번째 집을 털고 i번째 집 도둑질

      ⇒ case2[i] = max(case2[i-1], case2[i-2]+money[i])
- 최종 = max(case1[n-1], case2[n])