Algorithm 3월 4주차

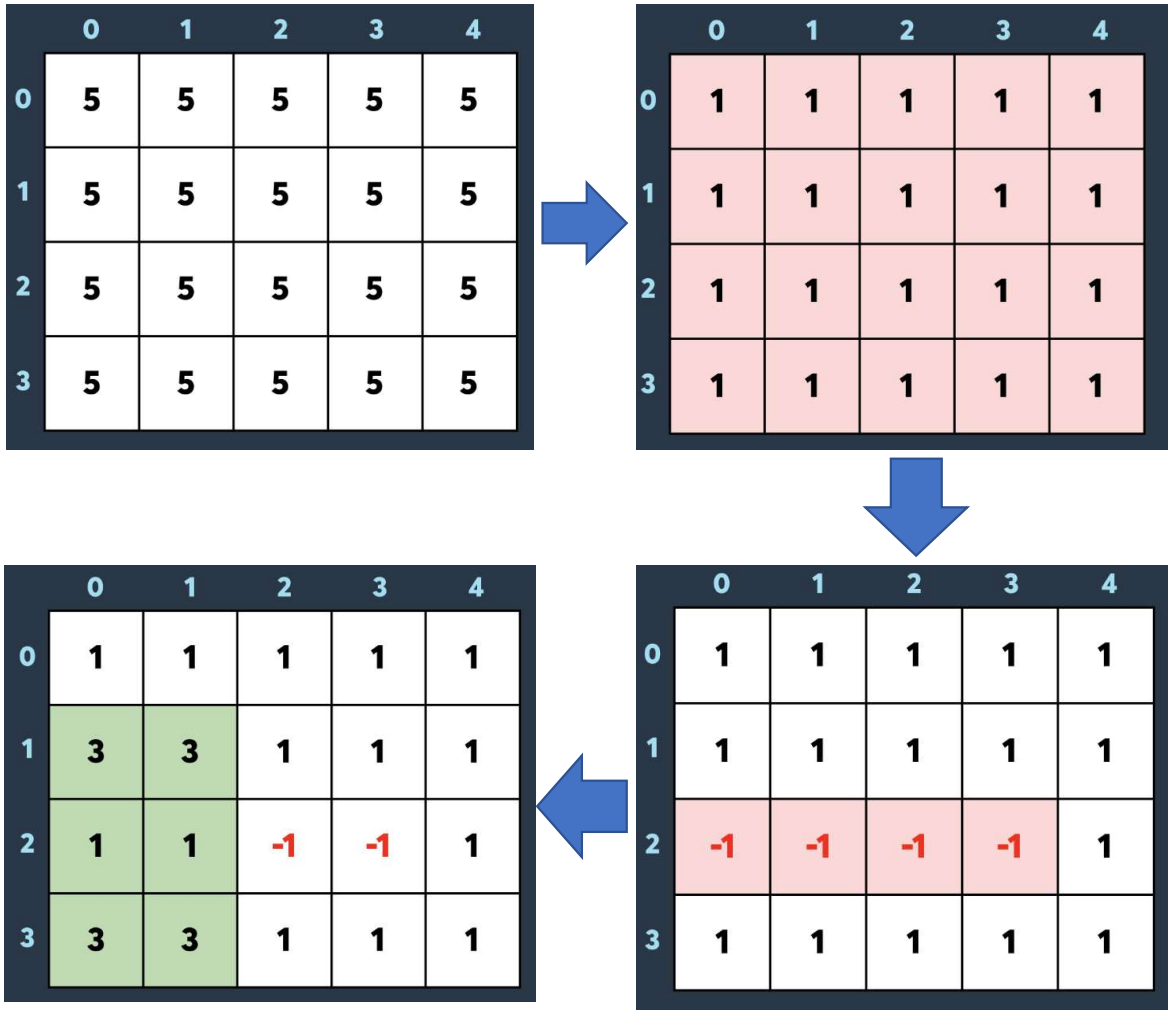
Prefix-Sum

Table of content

파괴되지 않은 건물

[프로그래머스] 파괴되지 않은 건물

- M x M 크기의 게임 맵
- 각 위치마다 건물 내구도
- 적은 범위 공격, 아군은 범위 회복 스킬
- 입력에는 피아정보, 행 시작점, 열 시작점, 행 끝점, 열 끝점, degree(피해 또는 회복 정도)가 주어짐
- 0이하는 파괴된 건물로 간주됨. 파괴되지 않은 건물 개수를 반환해야 하는 문제



시도 – Brute Force

- 모든 행과 열을 돌아가면서 degree만큼 공격 또는 회복
- 정확성은 통과했으나, 효율성에서 실패함
- Board 크기에서 시간 복잡도 100만 * skill 행의 길이 25만 = 2500억의 시간복잡도
- 시간복잡도 O(skill * M * N) = O(n^3)

```python
def solution(board, skill):
    answer = 0
    for kind, row_s, col_s, row_e, col_e, degree in skill:
        for row in range(row_s, row_e + 1):
            for col in range(col_s, col_e + 1):
                if kind == 1:
                    board[row][col] -= degree
                else:
                    board[row][col] += degree

    for row in range(len(board)):
        for col in range(len(board[0])):
            if board[row][col] > 0:
                answer += 1

    return answer
```

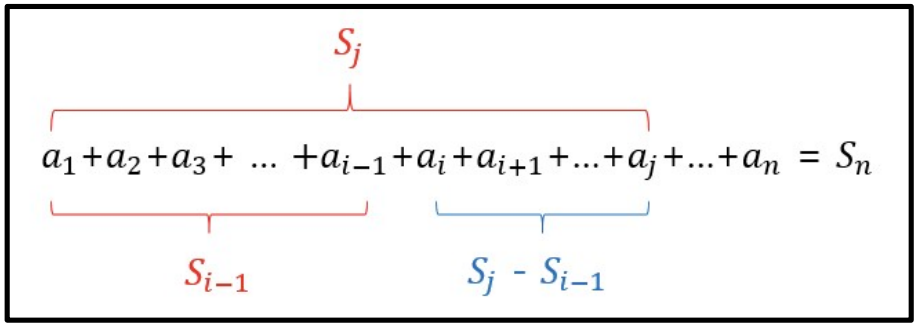
- 1 ≤ `board` 의 행의 길이 (= `N`) ≤ 1,000
- 1 ≤ `board` 의 열의 길이 (= `M`) ≤ 1,000
- 1 ≤ `board` 의 원소 (각 건물의 내구도) ≤ 1,000
- 1 ≤ `skill` 의 행의 길이 ≤ 250,000

테스트 1 〉 통과 (0.01ms, 9.99MB)
테스트 2 〉 통과 (0.05ms, 10.3MB)
테스트 3 〉 통과 (0.16ms, 10.1MB)
테스트 4 〉 통과 (0.56ms, 10.1MB)
테스트 5 〉 통과 (1.56ms, 10.2MB)
테스트 6 〉 통과 (1.95ms, 10.3MB)
테스트 7 〉 통과 (3.03ms, 10.3MB)
테스트 8 〉 통과 (4.55ms, 10.2MB)
테스트 9 〉 통과 (5.98ms, 10.3MB)
테스트 10 〉 통과 (12.39ms, 10.4MB)

테스트 1 〉 실패 (시간 초과)
테스트 2 〉 실패 (시간 초과)
테스트 3 〉 실패 (시간 초과)
테스트 4 〉 실패 (시간 초과)
테스트 5 〉 실패 (시간 초과)
테스트 6 〉 실패 (시간 초과)
테스트 7 〉 실패 (시간 초과)

풀이법 – 누적합 (Prefix-sum)

- 시간복잡도를 줄이기 위해서 누적합을 사용해야 한다.
- 누적합이란 특정 배열을 만들어서 첫번째 원소부터 임의의 원소 i까지의 합 S[i]를 모든 원소에 대해 구하는 것이다. 그러니까, S[1], S[2], … , S[n]을 구하는 것임



$$S[i] = A[1] + \cdots + A[i],\ S[0] = 0$$

$$S[r] = A[1] + \ldots + A[r]$$

$$S[l-1] = A[1] + \ldots + A[l-1]$$

$$S[r] - S[l-1] = A[l] + \cdots + A[r]$$

- 이 누적합을 이용해서 특정 구간의 합을 구하려면 뺄셈을 한 번만 하면 된다. 특정 구간 S[r] – S[l – 1]을 구하는 데 걸리는 시간복잡도는 O(1)

Prefix sum vs Brute force

- 누적합의 시간복잡도는 기존 Brute force에 비해 한 차원 낮다. Brute force를 사용했을 때 시간복잡도가 O(n^2)라면 누적합을 사용하면 시간복잡도가 O(n)이 된다.

크기가 N인 정수 배열 A가 있고, 여기서 다음과 같은 연산을 최대 M번 수행해야 하는 문제가 있습니다.

- 구간 l, r $(l \leq r)$이 주어졌을 때, $A[l] + A[l+1] + \cdots + A[r-1] + A[r]$을 구해서 출력하기

$A[l] + A[l+1] + \cdots + A[r-1] + A[r]$을 구하기 위해 소스 1과 같이 모두 더하는 방법이 있습니다.

Brute force

```
for j in range(m):
    temp = 0
    for i in range(l, r+1):
        temp += a[i]
    answer.append(temp)
```

이것을 M번 반복하면 시간복잡도는
O(M * N) = O(n ^ 2)

Prefix sum

```
for i in range(1, n + 1):
    s[i] = s[i - 1] + a[i]

for l, r in arr_m:
    answer.append(S[r] - S[l - 1])
```

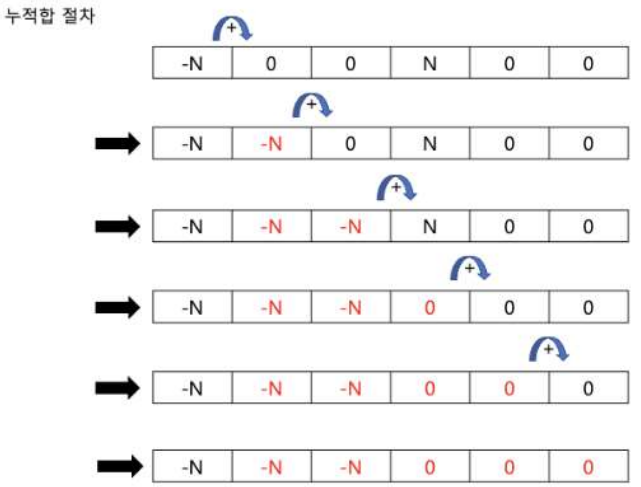
구간 합을 구하는 데 O(N), 별도의 루프에서
M번 연산. 시간복잡도는 O(N + M)

문제에 누적합 적용하기 - 1차원 배열

- 1차원 배열에 누적합 기록하기 O(n)

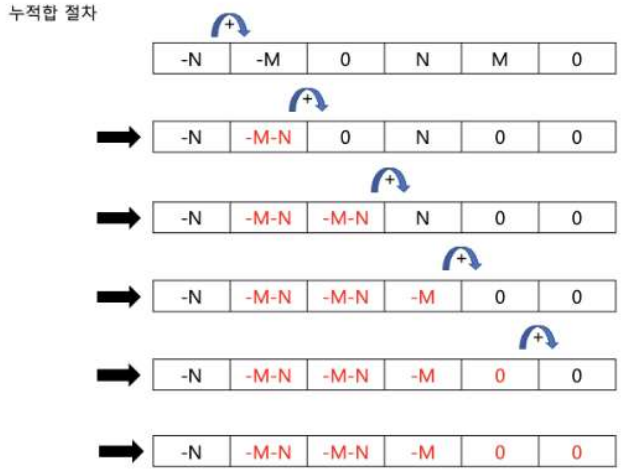
셋팅:

```
1 [-N, 0, 0, N, 0, 0]
```



- 1차원 배열 누적합 활용법:
  여러 연산을 겹칠 수 있다.

```
1 [-N, -M-N, -M-N, -M, 0, 0]
```



문제에 누적합 적용하기 - 2차원 배열

- 2차원 배열에 누적합 기록하기

셋팅:

```
[[ -N,  0,  0,  N,  0,  0],
 [  0,  0,  0,  0,  0,  0],
 [  0,  0,  0,  0,  0,  0],
 [  N,  0,  0, -N,  0,  0],
 [  0,  0,  0,  0,  0,  0],
 [  0,  0,  0,  0,  0,  0]]
```

2. 행 기준 누적합

진행방향 →

-N	-N	-N	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
N	N	N	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

3. 열 기준 누적합

진행방향 ↓

-N	-N	-N	0	0	0
-N	-N	-N	0	0	0
-N	-N	-N	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

파괴되지 않은 건물 – 누적합 풀이 코드

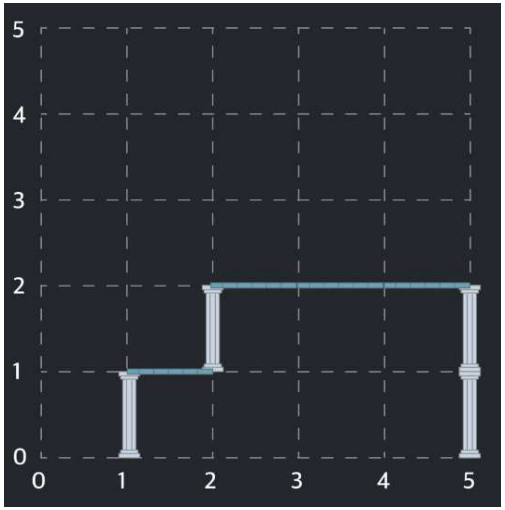
- 누적합 기록에 O(skill) 소요
- 행 기준 누적합에 O(MN) 소요
- 열 기준 누적합에 O(MN) 소요
- 기존 배열 합하는 데 O(MN) 소요
- 시간복잡도=O(skill) + O(MN) + O(MN) + O(MN) = O(n^2)

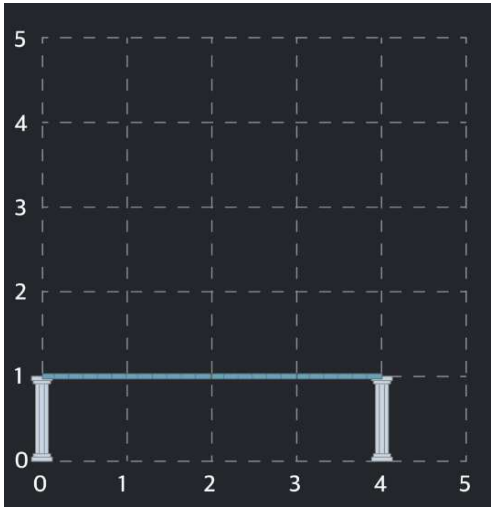
```python
def solution(board, skill):
    answer = 0
    prefix = [[0 for j in range(len(board[0]) + 1)] for i in range(len(board) + 1)]

    for kind, row_s, col_s, row_e, col_e, degree in skill:
        prefix[row_s][col_s] += degree if kind == 2 else -degree
        prefix[row_s][col_e + 1] += -degree if kind == 2 else degree
        prefix[row_e + 1][col_s] += -degree if kind == 2 else degree
        prefix[row_e + 1][col_e + 1] += degree if kind == 2 else -degree

    for row in range(len(board)):
        for col in range(len(board[0])):
            prefix[row][col + 1] += prefix[row][col]

    for col in range(len(board[0])):
        for row in range(len(board)):
            prefix[row + 1][col] += prefix[row][col]

    for row in range(len(board)):
        for col in range(len(board[row])):
            board[row][col] += prefix[row][col]
            if board[row][col] > 0:
                answer += 1

    return answer
```

기둥과 보 설치

[프로그래머스] 기둥과 보 설치

- 다음 조건을 만족하면서 기둥과 보를 세워야 함
- 기둥은 바닥 위에 있거나 보의 한쪽 끝 부분 위에 있거나, 또는 다른 기둥 위에 있어야 함.
- 보는 한쪽 끝 부분이 기둥 위에 있거나, 또는 양쪽 끝 부분이 다른 보와 동시에 연결되어 있어야 함.
- 입력 값은 [x좌표, y좌표, 기둥-보 종류, 설치삭제 여부]를 순서대로 나열한 2차원 배열이 주어짐
- 삭제도 설치 조건을 만족할 수 있는 경우에만 삭제를 실행할 수 있음.
- 기둥과 보가 설치된 형태를 이차원 배열로 반환해 줘야 한다.





시도 - 구현

- 이건 딱 봐도 구현 문제였음
- 하지만 설치 및 삭제가 가능한 모든 경우의 수를 조건문으로 구현하려다 보니 체크해야 하는 경우가 너무 많음

```python
def solution(n, build_frame):
    answer = []
    PILLAR, BEAM = 0, 1
    DELETE, INSERT = 0, 1
    build = [[[0, 0] for _ in range(n + 1)] for _ in range(n)]
```

```python
for x, y, kind, action in build_frame:
    if action == INSERT:
        if kind == PILLAR:
            if y == 0:
                build[y][x][PILLAR] = 1
            elif build[y - 1][x][PILLAR] == 1:
                build[y][x][PILLAR] = 1
            elif build[y][x][BEAM] == 1:
                build[y][x][PILLAR] = 1
            elif x > 0:
                if build[y][x - 1][BEAM] == 1:
                    build[y][x][PILLAR] = 1
        if kind == BEAM:
            if y == 0:
                continue
            elif build[y - 1][x][PILLAR] == 1:
                build[y][x][BEAM] = 1
            elif x < n:
                if build[y - 1][x + 1][PILLAR] == 1:
                    build[y][x][BEAM] = 1
                elif x > 0:
                    if build[y][x - 1][BEAM] == 1 and build[y][x + 1][BEAM] == 1:
                        build[y][x][BEAM] = 1
```

```python
if action == DELETE:
    if kind == PILLAR:
        if y == n:
            continue
        if x < n:
            if build[y + 1][x][BEAM] == 1 and build[y][x + 1][PILLAR] == 1:
                build[y][x][PILLAR] = 0
                continue
        if x > 0:
            if build[y + 1][x - 1][BEAM] == 1 and build[y][x - 1][PILLAR] == 1:
                build[y][x][PILLAR] = 0
                continue
        if 0 < x < n:
            if build[y + 1][x][BEAM] == 1 and build[y + 1][x + 1][BEAM] == 1 \
            and build[y + 1][x - 1][BEAM] == 1:
                build[y][x][PILLAR] = 0
                continue
    if kind == BEAM:
        if y == 0:
            build[y][x][BEAM] = 0
        elif build[y - 1][x][PILLAR] == 1:
```

대략 코드가 80줄 넘어갈 듯

```python
for x in range(len(build[0])):
    for y in range(len(build)):
        if build[y][x][PILLAR] == 1:
            answer.append([x, y, PILLAR])
        elif build[y][x][BEAM] == 1:
            answer.append([x, y, BEAM])

return answer
```

(참고) 시간 복잡도 팁

- 문제에 주어진 조건을 보고 허용되는 시간 복잡도를 계산할 수 있음
- 코딩테스트에서 시간제한은 1~5초 정도 (따로 시간이 명시되지 않았으면 약 5초)
- 시간 제한이 1초인 문제의 경우 다음과 같이 시간 복잡도 허용량 계산할 수 있다.

- N의 범위가 500인 경우 : 시간 복잡도가 $O(N^3)$인 알고리즘 설계 가능
- N의 범위가 2,000인 경우 : 시간 복잡도가 $O(N^2)$인 알고리즘 설계 가능
- N의 범위가 100,000인 경우 : 시간 복잡도가 $O(NlogN)$인 알고리즘 설계 가능
- N의 범위가 10,000,000인 경우 : 시간 복잡도가 $O(N)$인 알고리즘 설계 가능

풀이법 – 구현, in을 사용하여 검사

- 삼차원 배열을 선언하여 일일이 확인하는 작업은 매우 비효율적
- 그 대신, 문법 “in”을 사용하면 answer 에 있는 성분들을 검사하여 현재 지지하는 기둥과 보가 있는지, 그래서 쌓아 올리거나 빼낼 수 있는지 체크할 수 있다.
- 일단 Insert, Delete를 실행하고나서 나중에 검사

```
def solution(n, build_frame):
    answer = []
    for x, y, stuff, action in build_frame:
        if action == 1:
            answer.append([x, y, stuff])
            if not check(answer):
                answer.remove([x, y, stuff])
        else:
            answer.remove([x, y, stuff])
            if not check(answer):
                answer.append([x, y, stuff])
    return sorted(answer)
```

풀이법 – 구현, 함수 내 반복문, 완전탐색

- check 함수에서 for문을 사용하여 지지할 수 있는 기둥과 보의 모든 경우의 수를 체크함
- 주어진 조건을 만족시키면 그냥 통과하고, 만족 못 시키면 insert, delete를 취소시킨다.
- O(n^3)이 가능한 문제기 때문에 함수 내 반복문으로 완전 탐색 가능.

```
def check(answer):
    PILLAR, BEAM = 0, 1
    for x, y, stuff in answer:
        if stuff == PILLAR:
            if y == 0 or [x, y - 1, PILLAR] in answer \
            or [x, y, BEAM] in answer \
            or [x - 1, y, BEAM] in answer:
                continue
            return False
        else:
            if [x, y - 1, PILLAR] in answer \
            or [x + 1, y - 1, PILLAR] in answer \
            or ([x - 1, y, BEAM] in answer
                and [x + 1, y, BEAM] in answer):
                continue
            return False
    return True
```