깊이/너비 우선 탐색(DFS/BFS)

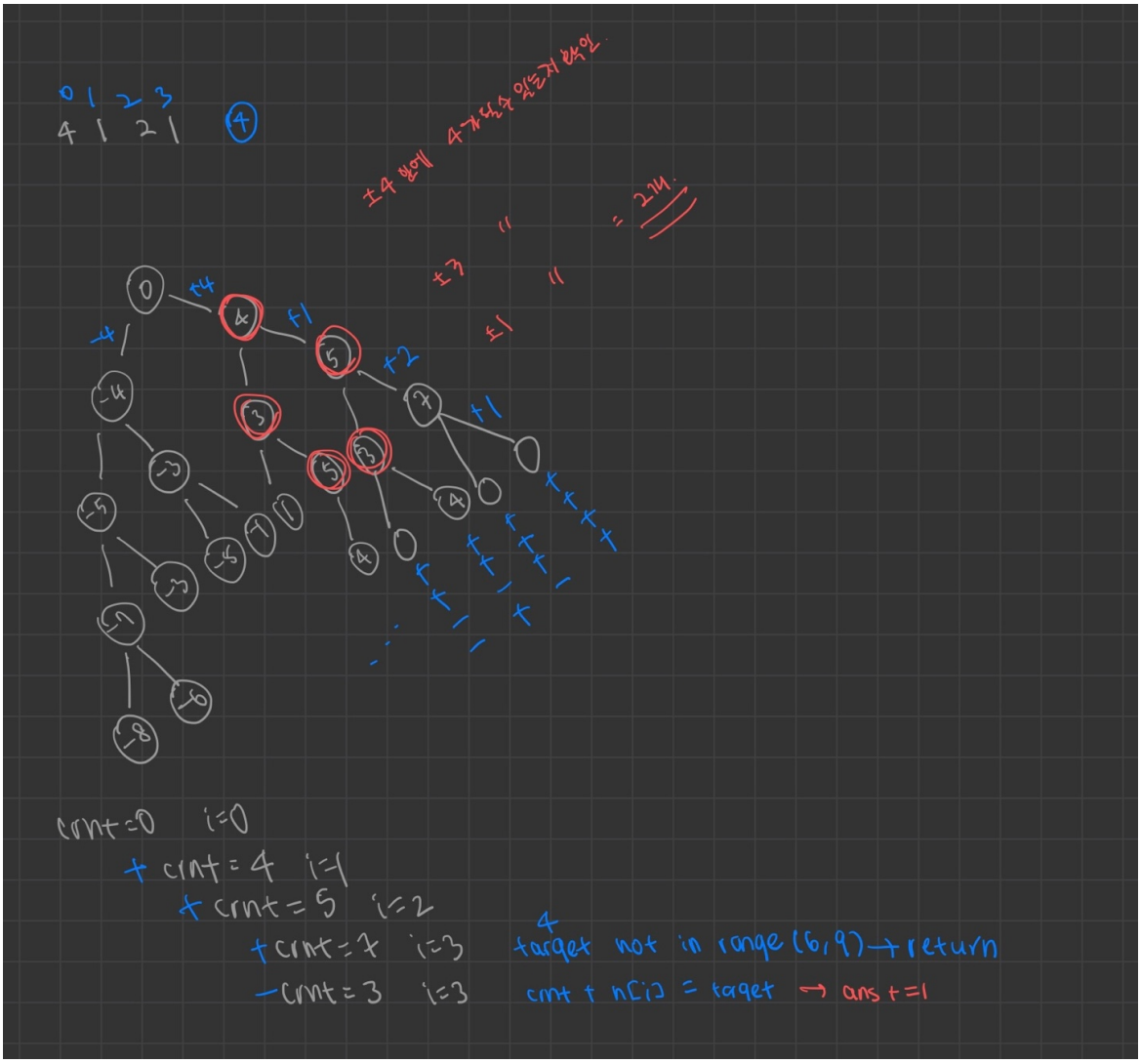
타겟 넘버

- n개의 음이 아닌 정수들을 순서를 바꾸지 않고 더하거나 빼서 타겟 넘버를 만든다 → 만드는 방법의 수를 리턴

```
# example 1
numbers = [1,1,1,1,1], target = 3 인 경우 다음 다섯가지 방법이 가능
-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3
return = 5

# example 2
numbers = [4,1,2,1], target = 4
+4+1-2+1 = 4
+4-1+2-1 = 4
return = 2
```

- 풀이 아이디어
  - dfs가 아니라 완전 탐색해야하는 거 아닌가? 생각했는데 dfs도 완전 탐색의 한 방법이다.
  - dfs 재귀 이용 : 더 이상 자식이 없으면 한 단계 위로 올라와서 다른 자식을 탐색

```
# 풀이 아이디어(DFS) - example 2 기준
+4+1+2+1 부터 -4-1-2-1 까지 탐색하는 과정

1) + + + +
  4번째 + 한 후 다시 3번째로 돌아가서 다른 자식 (4번째에 -가 오는 경우) 탐색
2) + + + -
  더 이상 3번째 + 의 자식이 없음(4번째 +, - 모두 탐색 완료)
  두번째로 돌아가서 다른 자식 탐색(3번째 -가 오는 경우)
3) + + - +
4) + + - -

...
```




```
def checkChildren(crnt, target, answer, numbers, i):
    # 하위 노드들을 계산할 필요가 있는지 확인 (계산할 필요없으면 리턴)
    ## ex) 위 그림에서 첫번째 -한 경우 (-4) 해당 분기에서 최종 값들은 4 이후 값들([1,2,1])로 계산해봤을 때 (-4-1-2-1) 과 (-4+1+2+1) 사이에 존재함
    ## 타겟 넘버 4는 해당범위에 존재하지 않으므로 첫번째에 -를 사용한 경우의 분기는 탐색할 필요 없음
    if target not in range(crnt-sum(numbers[i:]), crnt+sum(numbers[i:])+1):
        return answer

    if i == len(numbers)-1: # 마지막 값을 더하거나 뺄 차례
        if crnt + numbers[i] == target:
            answer += 1
        if crnt - numbers[i] == target:
            answer += 1
            pass
        return answer # 더 이상 더하거나 뺄 자식이 없으므로 리턴
                      # crnt에서 + 분기로 들어온 값이었다면 crnt로 돌아가서 형제인 - 분기 탐색하러감
                      # crnt에서 - 분기로 들어온 값이었다면 crnt에서의 분기들은 모두 탐색 완료 -> crnt의 부모로 돌아갈 것임

    answer = checkChildren(crnt+numbers[i],target, answer, numbers,i+1) # crnt 위치에서 + 분기 탐색
    answer = checkChildren(crnt-numbers[i], target, answer, numbers, i+1) # crnt 위치에서 + 분기 탐색 완료 후 - 분기 탐색하러 감

    return answer

def solution(numbers, target):
    answer = 0
    crnt = 0 # 현재까지 계산한 값

    answer = checkChildren(crnt, target, answer, numbers, 0)

    return answer
```

네트워크

```
n : 컴퓨터의 개수
computers : 컴퓨터 간 연결 정보가 담긴 배열
return : 네트워크의 개수

# example
n = 3
computers = [[1,1,0],[1,1,0],[0,0,1]]
# 모든 컴퓨터는 자기자신과 연결되어있음 (computer[i][i]는 항상 1)
# 0번 컴퓨터는 0번, 1번 컴퓨터에 연결되어있음
# 1번 컴퓨터는 0번, 1번 컴퓨터에 연결되어있음
# 2번 컴퓨터는 2번 컴퓨터에 연결되어있음
# 즉 0번 -- 1번 / 2번 이므로 네트워크는 2개
```

- 내 풀이
  - 각 망에다 번호를 붙임 (초기에는 모두 다른 번호의 망을 사용)

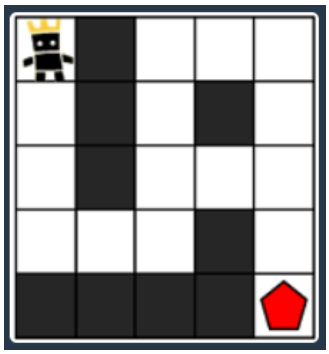
```
def solution(n, computers):
    answer = 0
    network = [i for i in range(n)] # 각 컴퓨터가 속한 망 번호 정보
    # 초기에는 모두 다른 망 번호 [0,1,2] : 0번 컴퓨터의 망 번호 0

    for idx, netInfo in enumerate(computers):
        for computer, connected in enumerate(netInfo):
            if network[idx] == network[computer]:
                continue
            if connected == 1:  # 두 컴퓨터가 연결됐다면 두 컴퓨터에 같은 망 번호를 붙여야 함
                if network[idx] > network[computer]: # 두 망 번호 중 작은 값을 선택
                    for i in range(n): # 망번호를 변경할 네트워크에 속한 모든 컴퓨터의 망번호를 변경
                        if network[i] == network[idx]:
                            network[i] = network[computer]
                else:
                    for i in range(n):
                        if network[i] == network[computer]:
                            network[i] = network[idx]

    answer = len(set(network))
    return answer
```

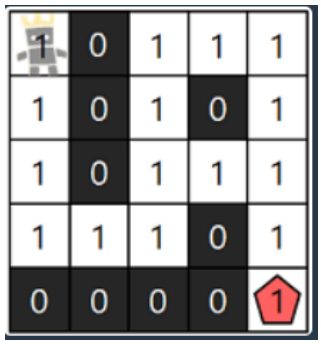
- 다른 풀이(DFS)

```
def solution(n, computers):
    answer = 0
    visited = [0 for i in range(n)]

    def dfs(computers, visited, start):
        # 현재 네트워크에 있는 모든 컴퓨터 탐색
        stack = [start]
        while stack:
            j = stack.pop() # 현재 컴퓨터(j)
            if visited[j] == 0:
                visited[j] = 1
            for i in range(0, len(computers)):
                if computers[j][i] ==1 and visited[i] == 0: # 현재 컴퓨터(j)에 연결된 컴퓨터를 찾음
                    stack.append(i)
    i=0
    while 0 in visited: # 탐색에 걸리지 않은 컴퓨터가 있는 한 반복
        if visited[i] ==0:
            dfs(computers, visited, i) # 네트워크 망 하나를 찾아냄
            answer +=1
        i+=1
    return answer
```

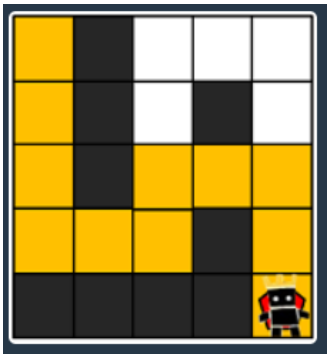
게임 맵 최단거리

- 목적지까지 가는 가장 짧은 거리는?






```
위의 지도는 다음과 같이 표현됨 (하얀부분(1)으로만 통행 가능)
[[1,0,1,1,1],[1,0,1,0,1],[1,0,1,1,1],[1,1,1,0,1],[0,0,0,0,1]]

아래의 두 가지 경우로 도달 가능
첫번째 경우는 11칸, 두번째 경우는 15칸을 거쳐 도달하므로
리턴값은 11로 반환
```



- DFS로 풀어야 할지 BFS로 풀어야 할지 잘 판단!
- DFS로 풀이 (효율성 통과 못함)
  - 현재 위치에서 갈 수 있는 분기가 생길때마다 재귀함수 호출 (목적지로 갈 수 있는 모든 경우에서의 거리를 모두 구한 후 가장 작은 값을 리턴함)
  - 복잡한 미로의 경우 경우의 수가 너무 많음 → 시간복잡도 저멀리,,

```
def findWay(i, j, answer, cases, maps, n, m):
    # maps[i][j] 현재 위치를 visited(0)로 바꿈
    maps[i][j] = 0
    if i == n-1 and j == m-1 : # 목적지에 도착
        cases.append(answer+1) # 이 경로의 거리 저장
        maps[i][j] = 1
        return

    # 현재 위치에서 갈 수 있는 길 체크
    if i == 0:
        upside = 0
    else:
        upside = maps[i-1][j]

    if i == n-1:
        downside = 0
    else:
        downside = maps[i+1][j]

    if j == 0:
```

```
        leftside = 0
    else:
        leftside = maps[i][j-1]

    if j == m-1:
        rightside = 0
    else:
        rightside = maps[i][j+1]

    # i = 0 : 윗방향은 체크 x
    # j = 0 : 왼쪽 방향 체크 x
    # i = n-1 : 아래 방향 체크 x
    # j = m-1 : 오른쪽 방향 체크 x

    # 현재 위치에서 갈 수 있는 각 분기별로 재귀
    if rightside == 1 : # move rightside
        findWay(i, j+1, answer+1, cases, maps, n, m)
    if downside == 1 : # move downside
        findWay(i+1, j, answer+1, cases, maps, n, m)
    if upside == 1 : # move upside
        findWay(i-1, j, answer+1, cases, maps, n, m)
    if leftside == 1 : # move leftside
        findWay(i, j-1, answer+1, cases, maps, n, m)

    maps[i][j] = 1 # 리턴하기 전에 현재 위치 not visited(1)로 돌려놓음
    return

def solution(maps):
    answer = 0
    cases = [] # 목적지까지 가는 모든 경우의 거리를 보관
    n = len(maps)
    m = len(maps[0])

    i = 0
    j = 0
    findWay(i, j, answer, cases, maps, n, m)

    if len(cases) == 0: # 목적지 도달 불가
        return -1

    return min(cases) # 가장 짧은 거리 반환
```

▼ 파이썬

- 파이썬 2차원 배열의 deep copy
  - findWay 함수에서 현재 위치를 visited로 바꿀 때(maps[i][j] = 0) 재귀함수가 끝난 후 다시 돌아올 때를 대비해 원본을 수정하면 안되니까 deep copy한 새 2차원 배열을 만들려고 했었음 (위의 코드에선 재귀함수 리턴하기 전에 다시 not visited로 돌려놓는 방식으로 함 maps[i][j] = 1)
  - 2차원 이상의 다차원 배열은 copy 모듈의 deepcopy 함수로만 깊은 복사가 가능

```
import copy
maps = [[1,0,1,1,1],[1,0,1,0,1],[1,0,1,1,1],[1,1,1,0,1],[0,0,0,0,1]]
tempMaps = copy.deepcopy(maps)

tempMaps[0][1] = 1
print(maps) # maps = [[1,0,1,1,1],[1,0,1,0,1],[1,0,1,1,1],[1,1,1,0,1],[0,0,0,0,1]]
print(tempMaps) # tempMaps = [[1,1,1,1,1],[1,0,1,0,1],[1,0,1,1,1],[1,1,1,0,1],[0,0,0,0,1]]
```

- BFS로 풀이
  - 당장 접근할 수 있는 곳들을 먼저 탐색하다가 목적지를 만났을 때 멈추면 그게 바로 가장 가까운 경로이므로 이 문제는 BFS로 푸는 것이 효율적