구현 2

23.12.25 (월)

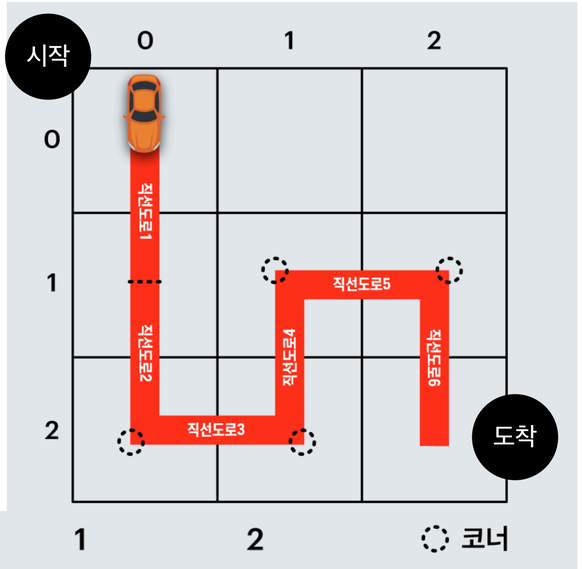
완전 탐색으로 풀기
너비 우선 탐색(BFS)
문제 62 - level3

문제 62번 이해하기

건설회사의 설계사인 `죠르디` 는 고객사로부터 자동차 경주로 건설에 필요한 견적을 의뢰받았습니다.

제공된 경주로 설계 도면에 따르면 경주로 부지는 `N x N` 크기의 정사각형 격자 형태이며 각 격자는 `1 x 1` 크기입니다.

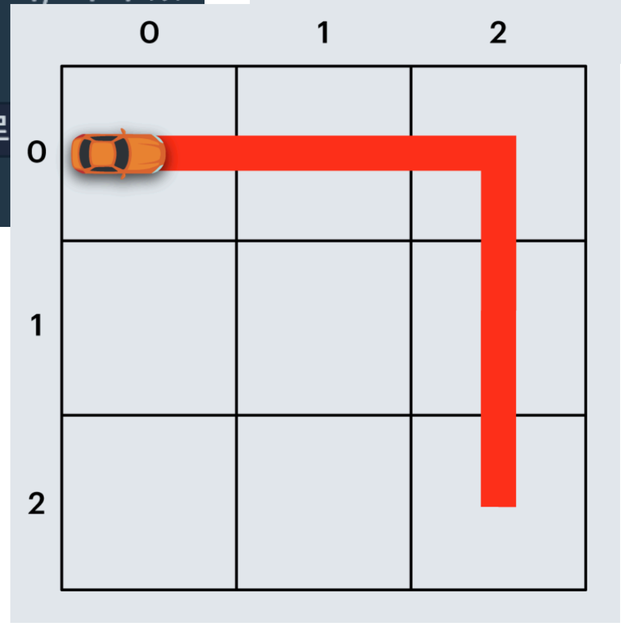
설계 도면에는 각 격자의 칸은 `0` 또는 `1` 로 채워져 있으며, `0` 은 칸이 비어 있음을 `1` 은 해당 칸이 벽으로 채워져 있음을 나타냅니다.



직선 도로 - 100원
코너 - 500원

경주로는 상, 하, 좌, 우로 인접한 두 빈 칸을 연결하여 건설할 수 있으며, 벽이 있는 칸에는 경주로를 건설할 수 없습니다.

이때, 인접한 두 빈 칸을 상하 또는 좌우로 연결한 경주로를 `직선 도로` 니다.



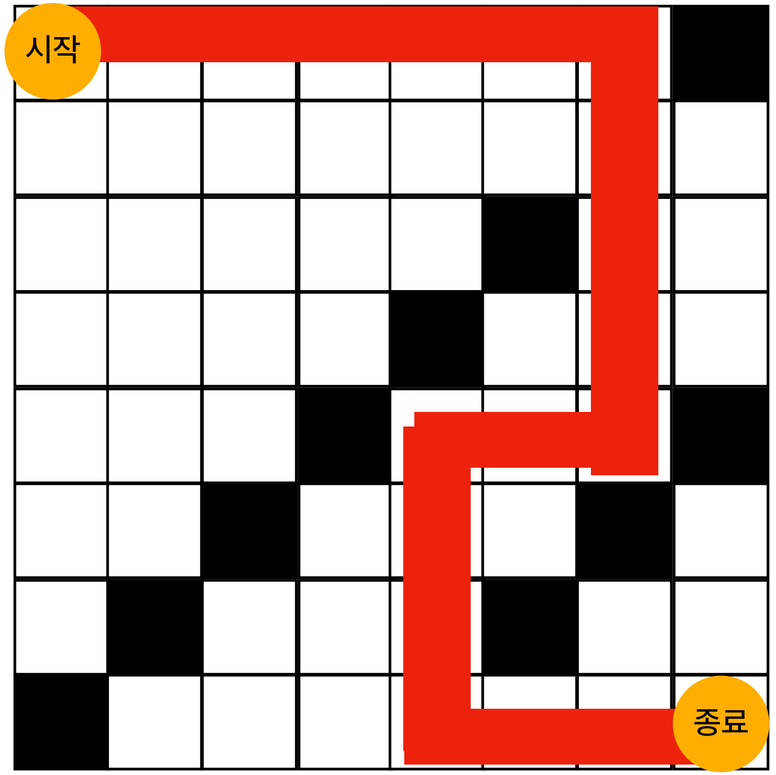
board	result
[[0,0,0],[0,0,0],[0,0,0]]	900
[[0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0],[0,0,0,0,0,1,0,0],[0,0,0,0,1,0,0,0],[0,0,0,1,0,0,0,1],[0,0,1,0,0,0,1,0],[0,1,0,0,0,1,0,0],[1,0,0,0,0,0,0,0]]	3800
[[0,0,1,0],[0,0,0,0],[0,1,0,1],[1,0,0,0]]	2100
[[0,0,0,0,0,0],[0,1,1,1,1,0],[0,0,1,0,0,0],[1,0,0,1,0,1],[0,1,0,0,0,1],[0,0,0,0,0,0]]	3200

최소 비용으로 경주로 건설하는 값 리턴

문제 62번 예제 케이스 이해하기

board	result
[[0,0,0],[0,0,0],[0,0,0]]	900
[[0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0],[0,0,0,0,0,1,0,0],[0,0,0,0,1,0,0,0],[0,0,0,1,0,0,0,1],[0,0,1,0,0,0,1,0],[0,1,0,0,0,1,0,0],[1,0,0,0,0,0,0,0]]	3800
[[0,0,1,0],[0,0,0,0],[0,1,0,1],[1,0,0,0]]	2100
[[0,0,0,0,0,0],[0,1,1,1,1,0],[0,0,1,0,0,0],[1,0,0,1,0,1],[0,1,0,0,0,1],[0,0,0,0,0,0]]	3200

case

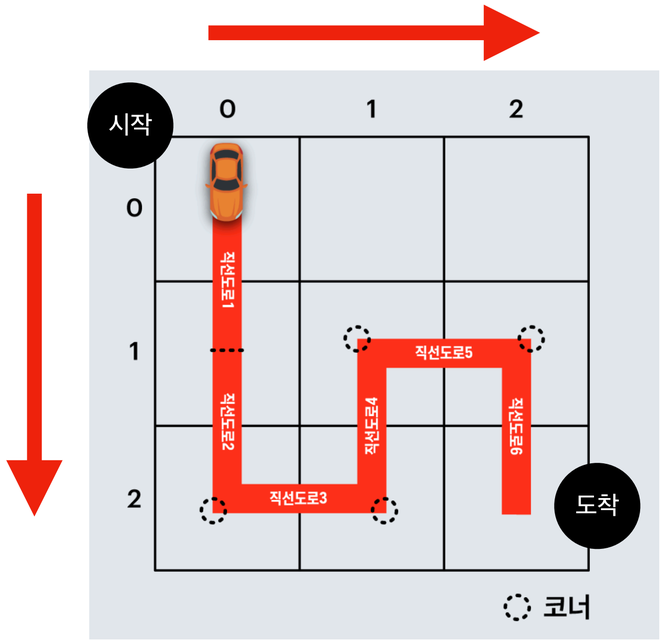


문제 62번 풀이 전략

BFS는 BFS대로 수행하고
그 과정에서 발행하는 비용을 계산하여 최소 비용을 알아내야 한다.

BFS로 알아낸 최소거리로 가면 곧 최소비용으로 경주로를 건설할 수 있을까?

최소 거리 => 최소비용?



전략

즉 시작위치에서 출발가능한

모든 경로에서 BFS 탐색을 수행하고

비용을 각각 계산하여

그 중에서 제일 작은 값을 리턴한다.

코드 작성 순서

1.탐색을 위한 준비 코드
2.BFS 함수 구현
3.최소 비용을 판단

문제 62번 풀이 - 탐색을 위한 준비

```
from collections import duque
def solution(board):
    size = len(board)
    paths = [(-1,0), (0,-1), (1,0), (0,1)]
```

문제에서는 상,하,좌,우로만 경주로를 지을 수 있다.

문제 62번 풀이 - BFS 함수 작성

```
def bfs(x,y,cost,path):
    graph = [[0] * size for _ in range(size)]
    for a in range(size):
        for b in range(size):
            if board[a][b] == 1: graph[a][b] = -1

    q = deque()
    q.append((x,y,cost,path))

    while q:
        x,y,cost,path = q.popleft()
        for i in range(len(paths)):
            nx = x + paths[i][0]
            ny = y + paths[i][1]

            if nx < 0 or nx >=size or ny <0 or ny >= size or graph[nx][ny] == -1 : continue

            if path == i : newest = cost + 100
            else : newest = cost + 600

            if graph[nx][ny] ==0 or (graph[nx][ny] !=0 and graph[nx][ny] > newcost):
                q.append((nx,ny,newcost,i))
                graph[nx][ny] = newcost

return graph[size-1][size-1]
```

주어진 board의 길이만큼 N X N 사이즈의 graph생성

주어진 Board에서 특정 위치의 값이 -1이라면 똑같이 graph의 해당 위치에도 -1 넣어주기

BFS 수행을 위한 큐 생성

제일 처음 값을 큐에 넣어서 시작

큐에 값이 있는 동안 BFS 수행

paths = [(-1,0), (0,-1), (1,0), (0,1)], x와 y의 특정 값에서 상,하,좌,우로 움직일 수 있다.

조건을 만족하지 않으면 그냥 continue

직선 코스면 100원 추가

그렇지 않으면 코너로 간주하고 600원 추가

기존의 newcost값이 더 작다면 그 값으로 [nx][ny]의 값을 newcost로 바꿔주기

주어진 정보 board

	1	

bfs 탐색을 위한 graph

	-1	

탐색을 위한 큐

(x,y,cost,path)

paths의 상하좌우 탐색을 통해 그 다음 nx,ny 위치값 찾기

	0	
1	현재 위치	3
	2	

- 0: 상 (위로 이동)
- 1: 좌 (왼쪽으로 이동)
- 2: 하 (아래로 이동)
- 3: 우 (오른쪽으로 이동)

queue에 있는 값들을 하나씩 처리하면서 newcost값 더해주기

0	100	200
		800
		900

문제 62번 풀이 - 최소 비용 계산

```
from collections import duque
def solution(board):
    <탐색 준비 코드>
    <BFS함수 부분>
    return min(bfs(0,0,0,2), bfs(0,0,0,3)
```

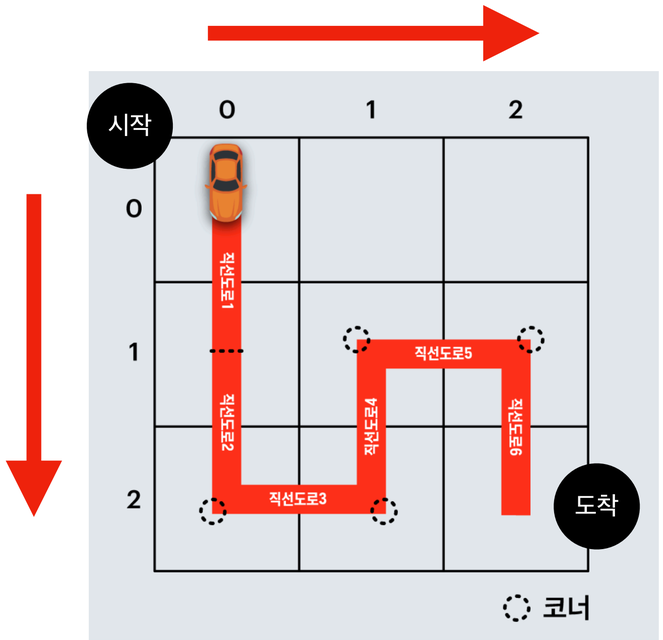
문제에서는 상,하,좌,우로만 경주로를 지을 수 있다.

	0	
1	현재 위치	3
	2	

- 0: 상 (위로 이동)
- 1: 좌 (왼쪽으로 이동)
- 2: 하 (아래로 이동)
- 3: 우 (오른쪽으로 이동)

시작	3	
2		

- 0: 상 (위로 이동)
- 1: 좌 (왼쪽으로 이동)
- 2: 하 (아래로 이동)
- 3: 우 (오른쪽으로 이동)



탐욕 알고리즘이란?

탐욕 알고리즘의 기본

- 지금까지는 최솟값 / 최댓값을 구하려면 항상 ‘전체’를 바라봐야 했다.
- 전체 결과를 따져야 하고 전체 결과를 따지려면 완전 탐색이 필수적으로 요구된다.
- 즉 모든 것을 확인하기 전까지는 최선의 선택지인지 판단이 불가능하다.
- 하지만 입력 크기가 너무 커서 완전 탐색이 불가능하거나 현실적으로 모든 경우의 수를 다 따져볼 수 없는 경우 주어진 선택지에서 가장 좋은 선택지를 계속 고르는 것이 차선책이 될 수 있다.
- 이 방법을 탐욕 알고리즘 (greedy algorithm)이라고 한다.

그리디 알고리즘

현재 상황에서의 최선

그리디 : 각 단계에서 미래를 생각하지 않고 현재 선택할 수 있는 것들 중에서 가장 최선의 것을 선택

그렇다면 '최선의 선택'에 대한 조건이 필요하다.

조건 1 **탐욕적 선택 속성**

한번 선택한 결과는 다음 선택에 어떠한 영향도 미치지 않아야 한다.
한번 선택하면 이전 결과를 다시 볼 필요가 없다.

조건 2 **최적 부분 구조**

작은 단위로 쪼개 만든 문제에서 최적해가 나왔을 때 큰 문제로 올라가도
똑같이 최적해를 만들어 낼 수 있어야 한다.

기본적으로 작은 문제의 해답을 이용하여 큰 문제의 정답을 구하는
구조라면 이를 만족한다.

그리디

그리디는 작은 문제든 큰 문제든 항상 일관적으로
'최선의 경우'만 선택한다.

동적계획법

동적계획법은 작은 문제가 주어졌을 때 특정
계산법을 이용해 최적의 정답을
끌어내는 원리를 큰 문제에서도 활용하는 것.

그리디 알고리즘

그리디를 사용할 수 있는 경우

분할 가능한 배낭 문제

배낭에 물건을 넣을 때 가장 가치 있게 넣는 방법을 물어볼 때 사용할 수 있다.

프림

세가지 그래프 알고리즘에서 모두 '최소 비용'으로 간선을 잇는 방법을 제시하며
이 때 그리디 방식을 사용할 수 있다.

크루스칼

다익스트라

회의실 배정 문제

각 회의의 시작 시간과 끝 시간이 정해져 있을 때 어떻게 회의 시간을 배정해야 가장 많은 회의를 할 수 있을까?

허프만 코드

주어진 문자열을 압출할 때 가장 빈도수가 많은 글자를 중심으로 문자열을 줄여나가는 알고리즘

그리디 알고리즘

그리디를 사용할 수 없는 경우

이진 트리의 최적합 경로

두 갈래길에서 처음 길은 비용이 매우 싸지만 다음 길이 비용이 비쌀 수 있으므로 그리디로 풀 수 없다.

0-1 배낭 문제

단순 가성비가 아니라 물건을 넣었을 때의 이득, 넣지 않았을 때의 이득을 따져야 한다면 그리디로 풀 수 없다.

외판원 문제

n개의 도시를 최소 비용으로 방문하려고 하는데 어떻게 돌아야 하는지 물어볼 때는
한 순간의 선택이 최선의 선택을 보장할 수 없다.

탐욕 알고리즘
문제 66 - level 3

문제 66번 이해하기

고속도로를 이동하는 모든 차량이 고속도로를 이용하면서 단속용 카메라를 한 번은 만나도록 카메라를 설치하려고 합니다.

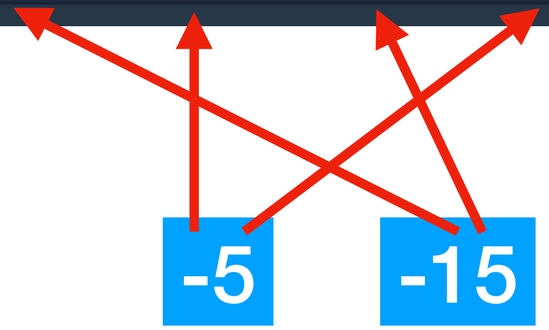
고속도로를 이동하는 차량의 경로 routes가 매개변수로 주어질 때, 모든 차량이 한 번은 단속용 카메라를 만나도록 하려면 최소 몇 대의 카메라를 설치해야 하는지를 return 하도록 solution 함수를 완성하세요.

제한사항

- 차량의 대수는 1대 이상 10,000대 이하입니다.
- routes에는 차량의 이동 경로가 포함되어 있으며 routes[i][0]에는 i번째 차량이 고속도로에 진입한 지점, routes[i][1]에는 i번째 차량이 고속도로에서 나간 지점이 적혀 있습니다.
- 차량의 진입/진출 지점에 카메라가 설치되어 있어도 카메라를 만난것으로 간주합니다.
- 차량의 진입 지점, 진출 지점은 -30,000 이상 30,000 이하입니다.

입출력 예

routes	return
[[-20,-15], [-14,-5], [-18,-13], [-5,-3]]	2



문제 66번 풀이 전략

모든 차량이 단속용 카메라를 최소 한번은 만나야 한다.
-> 첫 카메라의 위치가 중요하다.

첫번째 카메라는 처음으로 나가는 차량의 진출 지점에 달고
두번째 이후 카메라는 첫 번째 카메라에 해당하지 않는 차량들의 진출 지점 중 마지막으로 위치한
카메라의 지점과 가장 가까운 지점에 달면 된다.

진출 지점을 기준으로 데이터를 정렬하자.

진출 지점을 기준으로 정렬된 데이터에서 첫 번째 카메라 위치가 선정되면
그 다음부터는 첫번째 카메라에 해당하지 않는 차량들만 계속해서 확인하면 된다.

문제 66번 풀이 - 주어진 데이터를 진출 시점으로 정렬하기

```
def solution(routes):
    routes = sorted(routes,key=lambda x : x[1])
    last_camera = -30001
    answer = 0
```

x[0] - 진입시점
x[1] - 진출시점이므로 진출시점을 기준으로 routes 데이터 정렬하기

진입과 진출의 값 범위가
-30,000 ~ 30,000이므로 마지막 카메라의 위치는 -30001로 잡기

입출력 예

routes	return
[[-20,-15], [-14,-5], [-18,-13], [-5,-3]]	2

```
[[-20, -15], [-18, -13], [-14, -5], [-5, -3]]
```

sorted 후

문제 66번 풀이 - 새 차량이 진입하는 지점에서 카메라가 존재하는지 판단하기

```
for route in routes:
    if last_camera < route[0]:
        answer += 1
        last_camera = route[1]
```

제일 작게 잡아 놓은 last_camera 값 보다 route의 진입 값이 더 크다면

last_camera의 위치를 바꿔주기

last_camera < route[0] 은
마지막 카메라의 위치보다 진입 시점의 값이 크다는 것을 의미하고
이 말은 해당 route는 카메라를 통과하니까 카메라에 잡힌다는 것을 의미한다.