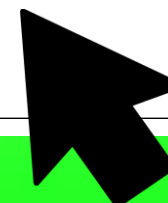


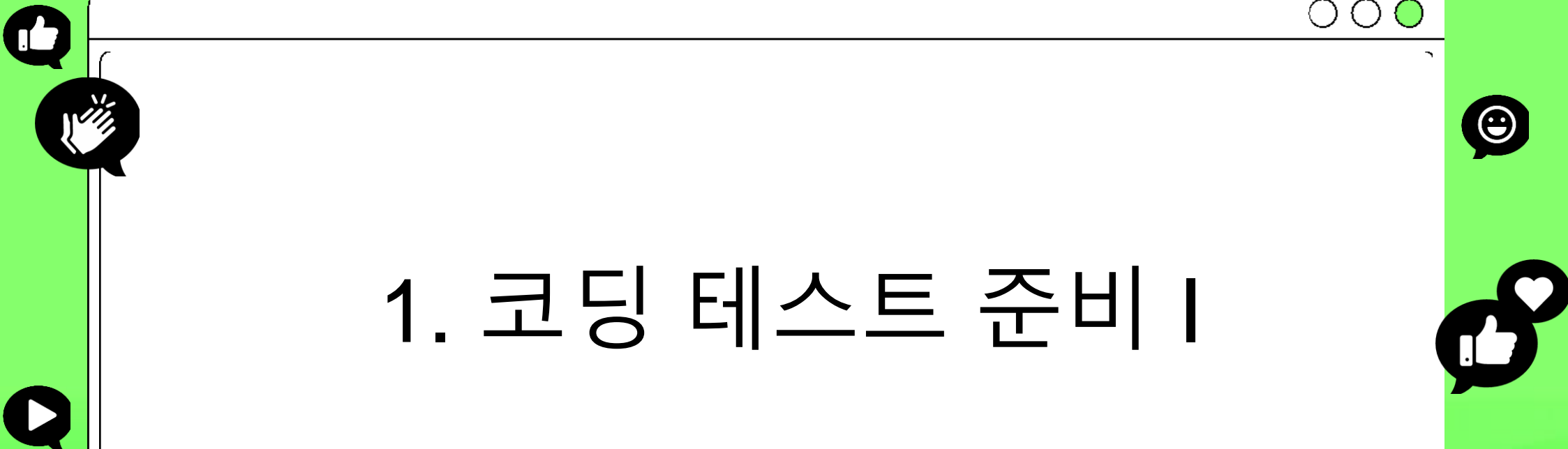
코딩 테스트 준비 II

(기본)



1. 코딩 테스트 준비 I

2. 단순 구현
(Implementation)



1. 코딩 테스트 준비 I



기본 코딩 테스트는 주로 **문제의 내용을 코드로 구현 가능**한지 테스트한다.

- 문제 풀이에 시간 제한이 없는 경우가 많기 때문에 시간 복잡도를 생각하지 않고 풀어보는 것이 좋다.
- 완전탐색 중에서도 2차원 배열의 탐색, 델타 탐색 등 선형 탐색이 주를 이룬다.
- 삼성 SW 역량테스트 IM 시험이 대표적인 예시이다.

2. 단순 구현(Implementation)

단순 구현(Implementation)은 문제에 제시된 풀이 과정을 그대로 구현하는 유형이다.

- 시뮬레이션의 경우 완전탐색 유형 중 하나로서, 모든 경우의 수를 탐색하여 풀이한다 .
- 아이디어나 알고리즘을 요구하기 보다는, 문제에 제시된 과정을 그대로 구현할 수 있는가가 핵심이다.
- 삼성 SW 역량테스트 IM, A형에서 주로 출제된다.

2. 단순 구현 (Implementation)

단순 구현 연습

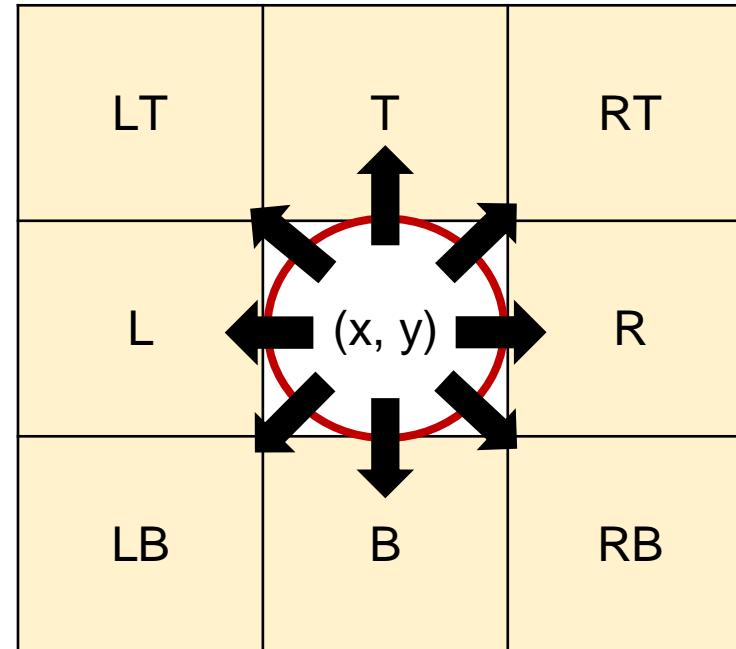
문제 번호	문제	링크
BOJ 1063	킹	https://www.acmicpc.net/problem/1063

킹 문제의 경우 **델타값을 이용한 탐색**과 **아스키코드**를 사용한다.
단순 구현 유형이므로, 문제에 제시된 동작을 그대로 구현하면 된다.
(제한 시간 40분 이상)

2. 단순 구현 (Implementation)

상하좌우 + 대각선의 8방향 델타값을 이용한다.

```
# 8방향 델타값
directions = {
    "R": (0, 1),
    "L": (0, -1),
    "B": (1, 0),
    "T": (-1, 0),
    "RT": (-1, 1),
    "LT": (-1, -1),
    "RB": (1, 1),
    "LB": (1, -1),
}
```



방향이 알파벳으로 입력되므로, 딕셔너리를 사용한다

2. 단순 구현 (Implementation)

아스키코드를 이용해 체스판 위치(A1, A2 ...)를 좌표로 변환한다.

```
k, s, n = input().split()

kx, ky = 8 - int(k[1]), ord(k[0]) - 65 # king x, y
sx, sy = 8 - int(s[1]), ord(s[0]) - 65 # stone x, y
```

`ord()`는 특정 문자를 아스키코드로 변환하는 파이썬 내장 함수이다.

65는 아스키코드에서 “A”를 나타내므로, 이를 빼서 열의 좌표값을 구한다.

2. 단순 구현 (Implementation)

DFS를 이용해 이차원 격자를 탐색하는 문제가 자주 출제된다.

[예제]

세로가 n , 가로가 m 의 길이를 갖는 $n \times m$ 미로가 있다.

미로의 통로는 0, 벽은 1로 표현된다. 벽이 있는 곳으로는 이동할 수 없다.

출발점이 (0, 0)이고 도착점이 ($n-1$, $m-1$)일 때, 출발점에서 도착점까지의 경로를 출력하시오.

[입력]

```
5 5
0 0 0 0 0
1 0 1 1 1
0 0 1 1 1
1 0 0 0 0
1 1 1 1 0
```

[출력]

```
[(0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (4, 4)]
```

2. 단순 구현 (Implementation)

미로는 통로는 0, 벽은 1인 이차원 격자

0	0	0	0	0
1	0	1	1	1
0	0	1	1	1
1	0	0	0	0
1	1	1	1	0

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

출발				
벽		벽	벽	벽
		벽	벽	벽
벽				
벽	벽	벽	벽	도착

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

출발				
벽		벽	벽	벽
		벽	벽	벽
벽				
벽	벽	벽	벽	도착

막혔다!

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

출발				
벽		벽	벽	벽
		벽	벽	벽
벽				
벽	벽	벽	벽	도착

마지막 갈림길로 돌아감

6. 이차원 격자에서의 DFS

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

막혔
다!

출발				
벽		벽	벽	벽
		벽	벽	벽
벽				
벽	벽	벽	벽	도착

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

출발				
벽		벽	벽	벽
		벽	벽	벽
벽				
벽	벽	벽	벽	도착

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

출발				
벽		벽	벽	벽
		벽	벽	벽
벽				
벽	벽	벽	벽	도착

탈출 성공!
공!

6. 이차원 격자에서의 DFS

DFS 방식으로 출발점부터 도착점까지의 경로를 찾아보자!

(0,0)	(0,1)			
벽	(1,1)	벽	벽	벽
	(2,1)	벽	벽	벽
벽	(3,1)	(3,2)	(3,3)	(3,4)
				(4,4)

```
# 입력
5 5
0 0 0 0 0
1 0 1 1 1
0 0 1 1 1
1 0 0 0 0
1 1 1 1 0
```



```
# 출력 (출발점 -> 도착점 경로)
```

```
>>> [(0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (4, 4)]
```

그런데 DFS는 그래프 탐색 알고리즘 아닌가요? 이건 그래프가 아니라 격자인데...

0	0	0	0	0
1	0	1	1	1
0	0	1	1	1
1	0	0	0	0
1	1	1	1	0

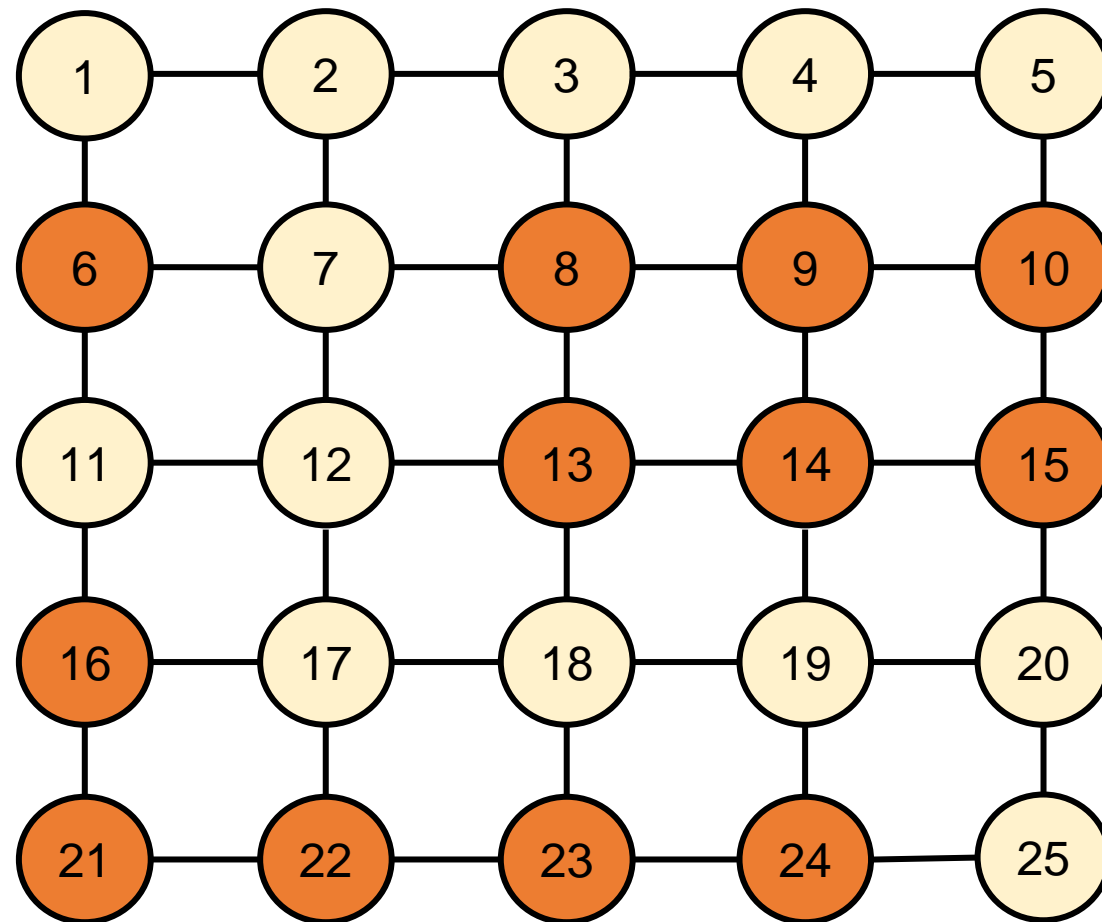
격자의 각 칸에 번호를 붙여볼까요?

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

이제 각 칸이 하나의 정점이고, 번호는 정점의 번호라고 생각해봅시다!

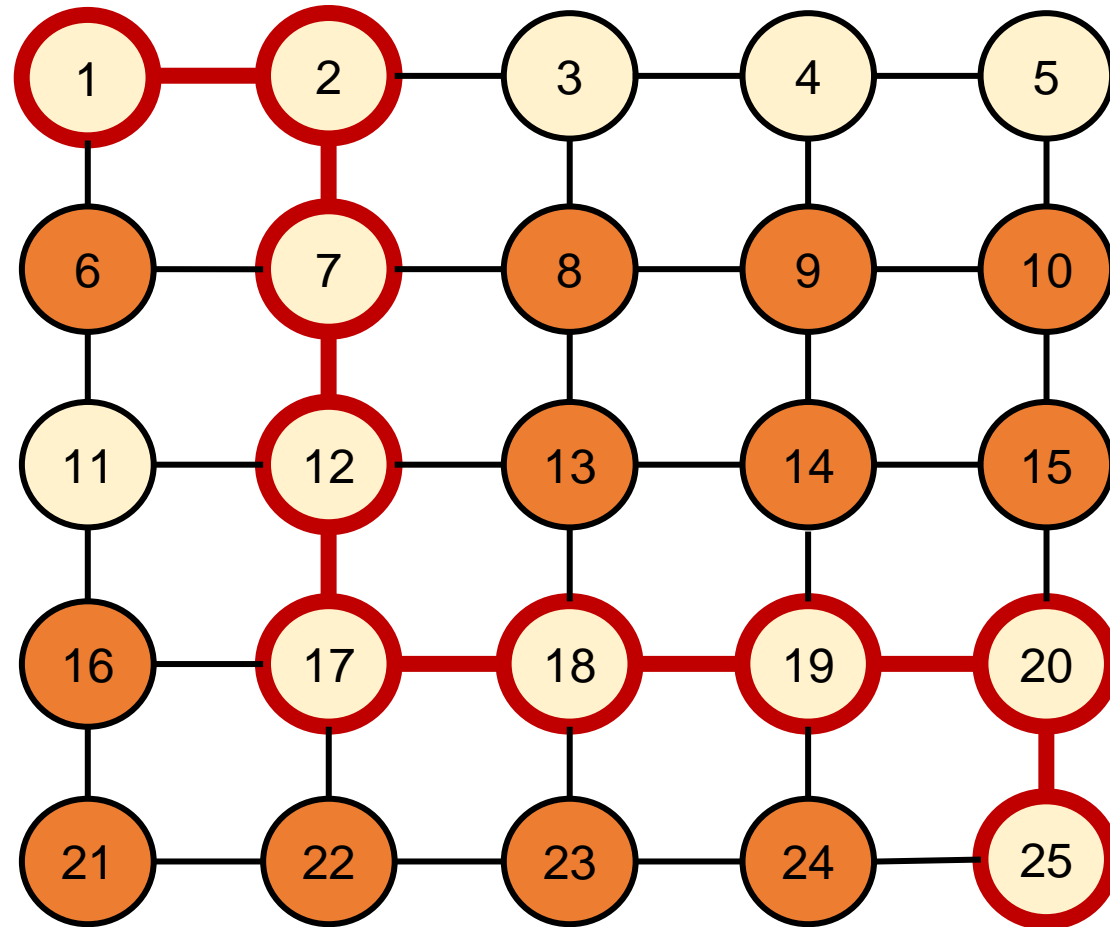
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

이차원 격자는 결국 상하좌우로 연결된 그래프와 같다!



6. 이차원 격자에서의 DFS

결국 1번 정점에서 DFS를 시작하여 25번 정점에 도착하는 경로를 찾는 문제다!



이를 위해 방문 체크 리스트도 이차원의 형태로 선언한다.

```
visited = [[False] * m for _ in range(n)]
```

False	False	False	False	False
False	False	False	False	False
False	False	False	False	False
False	False	False	False	False
False	False	False	False	False

2. 단순 구현 (Implementation)

인접 정점은 델타값을 이용한 상하좌우 이동으로 갈 수 있다.

