코딩 테스트 대비 핵심 알고리즘

핵심 유형 문제풀이

핵심 유형 문제풀이 | 다양한 문제를 접하며 코딩 테스트에 익숙해지기

강사 나동빈

코딩 테스트 대비
핵심 알고리즘

핵심 유형 문제풀이

드래곤 커브: https://www.acmicpc.net/problem/15685

혼자 힘으로 풀어보기

문제 제목: 드래곤 커브

문제 난이도: ★★★☆☆

문제 유형: 시뮬레이션

추천 풀이 시간: 60분

드래곤 커브: https://www.acmicpc.net/problem/15685

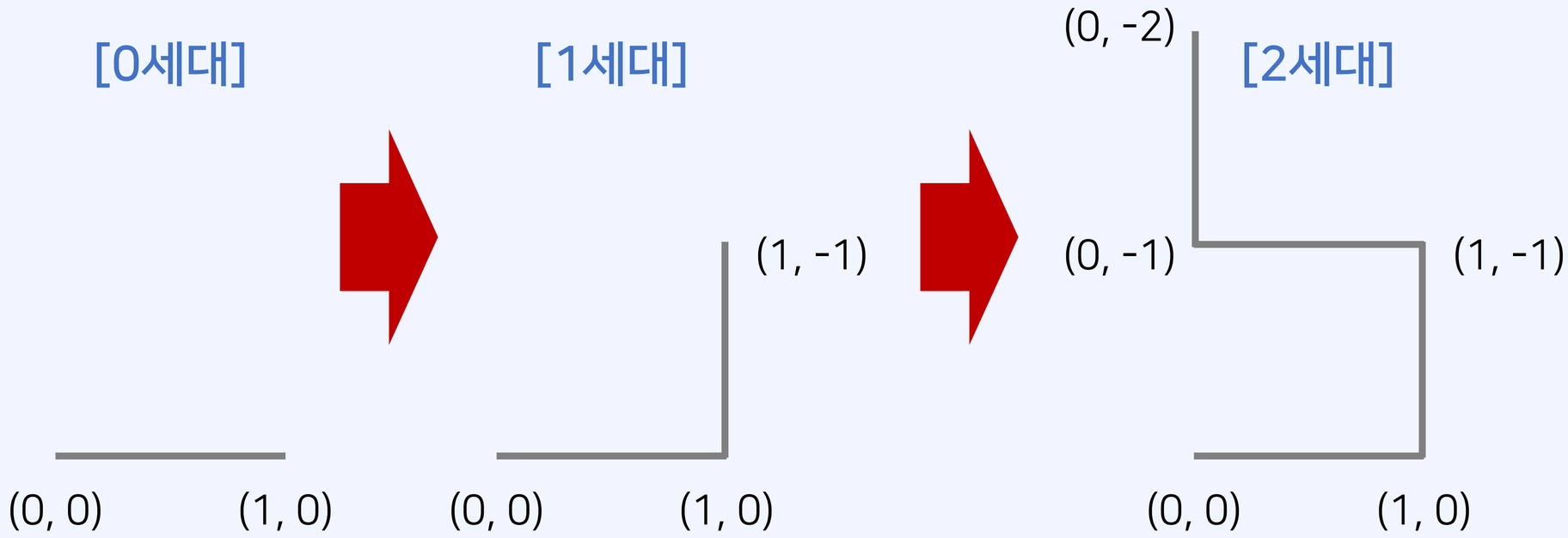
문제 해결 아이디어

- 문제의 요구사항 그대로 구현하면 되는 **시뮬레이션** 유형의 문제다.
- 다만, 문제를 풀기 위한 기본적인 아이디어를 떠올릴 수 있어야 한다.
- 본 문제는 **[규칙성]**을 발견한 뒤에, 코드로 작성하여 문제를 해결할 수 있다. 다시 말해 규칙성을 실제로 시뮬레이션(구현)하면 된다.

드래곤 커브: https://www.acmicpc.net/problem/15685

문제 해결 아이디어

- 드래곤 커브는 ① **시작점**, ② **시작 방향**, ③ **세대**로 구성된다.
- 세대를 거칠 때는, 시계 방향으로 90도 회전시킨 뒤에 이전 세대의 끝점에 붙인다.
- (0, 0)에서 시작하고, 시작 방향이 오른쪽인 드래곤 커브는 다음과 같다.



드래곤 커브: https://www.acmicpc.net/problem/15685

문제 해결 아이디어

- 본 문제에서는 반시계 방향으로 4가지 방향을 고려하며, 차례대로 다음과 같다.
- **[방향 정보]** 0: 우(→), 1: 상(↑), 2: 좌(←), 3: 하(↓)
- 이것을 표현하면 다음과 같다.

```
# [중요] 이 문제에서는 x축과 y축의 의미가 일반적인 문제와 반대
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

드래곤 커브: https://www.acmicpc.net/problem/15685

문제 해결 아이디어

- (0, 0)에서 시작하고, 시작 방향이 오른쪽(0)인 드래곤 커브의 예시를 확인해 보자.
- **0단계**: 0
- **1단계**: 0, 1
- **2단계**: 0, 1, 2, 1
- **3단계**: 0, 1, 2, 1, 2, 3, 2, 1
- 각 단계를 확인한 뒤에 규칙성을 도출할 수 있는데, **다음 단계 배열**은 다음과 같다.
- (이전 단계 배열) + (이전 단계 배열을 뒤집고, 각 원소에 1을 더한 것)

드래곤 커브: https://www.acmicpc.net/problem/15685

문제 해결 아이디어

- 예를 들어 3단계 [0, 1, 2, 1, 2, 3, 2, 1]를 살펴보자.

1) 이것을 뒤집으면 [1, 2, 3, 2, 1, 2, 1, 0]이 된다.

2) 이후에 각 원소에 1을 더하면(반시계 방향 회전) [2, 3, 0, 3, 2, 3, 2, 1]이 된다.

[결과] 4단계: [0, 1, 2, 1, 2, 3, 2, 1, 2, 3, 0, 3, 2, 3, 2, 1]

드래곤 커브: https://www.acmicpc.net/problem/15685

문제 해결 아이디어

- 세대(g)가 최대 10이므로, 각 드래곤 커브의 원소는 최대 1,024개다.
- 드래곤 커브의 수(N)은 최대 20이다.
- 따라서, $N \times N$ 배열을 만든 뒤에, **드래곤 커브가 지나간 위치를 True 값으로 설정**한다.
- 입력으로 주어지는 드래곤 커브는 격자 밖으로 벗어나지 않으므로, 단순히 시뮬레이션을 통해 문제를 해결할 수 있다.

드래곤 커브: https://www.acmicpc.net/problem/15685

소스 코드 1)


```python
import sys
# 빠른 입력 함수 사용
input = sys.stdin.readline

n = int(input())
arr = [[False] * 101 for _ in range(101)]

# [중요] 이 문제에서 x축과 y축은 일반적인 문제와 반대되는 의미를 가짐
# 0: 우(→), 1: 상(↑), 2: 좌(←), 3: 하(↓)
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

드래곤 커브: https://www.acmicpc.net/problem/15685

소스 코드 2)


```
for _ in range(n):
    x, y, d, g = map(int, input().split()) # 드래곤 커브의 시작점(x, y), 시작 방향(d), 세대(g)
    current = [d]  # 현재 이동 계획(current)
    for i in range(g):
        temp = [] # 현재 배열을 뒤집은 뒤에 1씩 더하여 temp 배열 생성
        for j in range(len(current) - 1, -1, -1):
            temp.append((current[j] + 1) % 4)
        current += temp # temp 배열을 current 뒤에 붙이기
    # 드래곤 커브가 지나가는 점들
    arr[x][y] = True
    for i in range(len(current)):
        x = x + dx[current[i]]
        y = y + dy[current[i]]
        arr[x][y] = True

# 네 꼭짓점이 모두 드래곤 커브의 일부인 정사각형의 개수 계산
result = 0
for i in range(100):
    for j in range(100):
        if (arr[i][j] and arr[i][j + 1] and arr[i + 1][j] and arr[i + 1][j + 1]):
            result += 1
print(result)
```

혼자 힘으로 풀어보기

문제 제목: 마법사 상어와 파이어스톰

문제 난이도: ★★★☆☆

문제 유형: 시뮬레이션, DFS/BFS

추천 풀이 시간: 60분

마법사 상어와 파이어스톰: https://www.acmicpc.net/problem/20058

문제 해결 아이디어

- 문제의 요구사항 그대로 구현하면 되는 **시뮬레이션** 유형의 문제다.
- **"가장 큰 덩어리(연결요소)가 차지하는 칸의 개수"**를 구하기 위해 **DFS**를 사용할 수 있다.
- 맵의 크기는 최대 64 X 64이므로(N이 최대 6), 원소의 개수는 약 3,600개다.
- 연산의 횟수(Q)는 최대 1,000이므로, 시뮬레이션을 위해 약 360만 회의 연산이 필요하다.

문제 해결 아이디어

- 본 문제에서는 파이어스톰을 **총 Q번 시전**한다.
- **파이어스톰**: 전체 맵을 $2^L \times 2^L$ 크기의 부분 격자로 나누고, 각 부분 격자를 90도 회전시킨다.

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	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

마법을 시전하기 전

9	1	11	3	13	5	15	7
10	2	12	4	14	6	16	8
25	17	27	19	29	21	31	23
26	18	28	20	30	22	32	24
41	33	43	35	45	37	47	39
42	34	44	36	46	38	48	40
57	49	59	51	61	53	63	55
58	50	60	52	62	54	64	56

$L = 1$일 때

25	17	9	1	29	21	13	5
26	18	10	2	30	22	14	6
27	19	11	3	31	23	15	7
28	20	12	4	32	24	16	8
57	49	41	33	61	53	45	37
58	50	42	34	62	54	46	38
59	51	43	35	63	55	47	39
60	52	44	36	64	56	48	40

$L = 2$일 때

문제 해결 아이디어

[행렬을 시계 방향으로 90도 회전시키는 함수]

- 본 문제를 풀기 위해서 행렬(matrix)을 시계 방향으로 90도 회전시키는 함수가 필요하다.

1	2	3	4
5	6	7	8
9	10	11	12

90도 회전

9	5	1
10	6	2
11	7	3
12	8	4

문제 해결 아이디어

[행렬을 시계 방향으로 90도 회전시키는 함수]

- 본 문제를 풀기 위해서 행렬(matrix)을 시계 방향으로 90도 회전시키는 함수가 필요하다.

```
# 하나의 행렬이 있을 때, 시계 방향으로 90도 회전시키는 함수
def rotate(matrix):
    n, m = len(matrix), len(matrix[0])
    result = [[0] * n for _ in range(m)]
    for i in range(m):
        for j in range(n):
            result[i][j] = matrix[n - 1 - j][i]
    return result

matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
print(rotate(matrix))
```

문제 해결 아이디어

[N X N 행렬을 부분 격자들로 나누는 함수]

- 전체 행렬을 여러 개의 부분 격자로 나누는 함수가 필요하다.

```
# N X N 행렬을 (2^L × 2^L) 크기의 부분 격자들로 나누는 함수
# 각 부분 격자는 (① 왼쪽 위, ② 오른쪽 아래) 위치로 표현 가능
def devide(l):
    step_size = 2**l
    positions = []
    for i in range(0, 2**n, step_size):
        for j in range(0, 2**n, step_size):
            positions.append(((i, j), (i + step_size, j + step_size)))
    return positions
```

소스 코드 1)


```python
import sys
input = sys.stdin.readline # 빠른 입력 함수 사용
sys.setrecursionlimit(int(1e5)) # DFS를 위한 재귀 제한 해제

# 하나의 행렬이 있을 때, 시계 방향으로 90도 회전시키는 함수
def rotate(matrix):
    n, m = len(matrix), len(matrix[0])
    result = [[0] * n for _ in range(m)]
    for i in range(m):
        for j in range(n):
            result[i][j] = matrix[n - 1 - j][i]
    return result

# N X N 행렬을 (2^L × 2^L) 크기의 부분 격자들로 나누는 함수
# 각 부분 격자는 (① 왼쪽 위, ② 오른쪽 아래) 위치로 표현 가능
def devide(l):
    step_size = 2**l
    positions = []
    for i in range(0, 2**n, step_size):
        for j in range(0, 2**n, step_size):
            positions.append(((i, j), (i + step_size, j + step_size)))
    return positions
```

소스 코드 2)


```python
# 북, 동, 남, 서
dx = [-1, 0, 1, 0]
dy = [0, 1, 0, -1]

# 맵의 크기 정보 N, 연산 횟수 Q 입력
n, q = map(int, input().split())
arr = []

# (2^N × 2^N) 행렬 입력
for i in range(2**n):
    row = list(map(int, input().split()))
    arr.append(row)

# 모든 연산 입력
operators = list(map(int, input().split()))
```

```python
# 인접한 곳에 얼음이 없는 곳들을 녹이는 함수
def melt():
    melted = []
    for x in range(2**n):
        for y in range(2**n):
            cnt = 0
            for i in range(4):
                nx = x + dx[i]
                ny = y + dy[i]
                # 인접한 위치가 범위를 벗어나는 경우 무시
                if nx < 0 or ny < 0 or nx >= 2**n or ny >= 2**n:
                    continue
                # 인접한 위치에 얼음이 있는 경우 카운트
                if arr[nx][ny] >= 1:
                    cnt += 1
            # 주변에 얼음이 2개 이하로 있다면, 녹이기
            if cnt < 3:
                melted.append((x, y))
    return melted
```

소스 코드 3)


```
# 하나씩 연산을 확인하며
for l in operators:
    # N X N 행렬을 (2^L × 2^L) 크기의 부분 격자들로 나누기
    positions = devide(l)
    for position in positions:
        pos1, pos2 = position
        # 부분 격자(current) 초기화
        current = []
        for i in range(pos1[0], pos2[0]):
            row = []
            for j in range(pos1[1], pos2[1]):
                row.append(arr[i][j])
            current.append(row)
        # 부분 격자(current) 회전시키기
        current = rotate(current)
        # 회전된 결과 저장하기
        step_size = 2**l
        for i in range(step_size):
            for j in range(step_size):
                arr[i + pos1[0]][j + pos1[1]] = current[i][j]
    melted = melt()  # 녹일 위치 찾기
    for (x, y) in melted:  # 각 위치에 있는 얼음 녹이기
        if arr[x][y] >= 1: arr[x][y] -= 1
```

소스 코드 4)


```python
# 모든 위치에 있는 얼음들의 합 출력
answer = 0
for row in arr:
    answer += sum(row)
print(answer)

# DFS를 위한 방문 처리 배열
visited = [[False] * (2**n) for _ in range(2**n)]

# DFS로 가장 큰 연결 요소(connected component) 찾기
def dfs(x, y):
    result = 1  # 원소의 개수 세기
    for i in range(4):  # 인접한 위치 확인
        nx = x + dx[i]
        ny = y + dy[i]
        # 인접한 위치가 범위를 벗어나는 경우 무시
        if nx < 0 or ny < 0 or nx >= 2**n or ny >= 2**n:
            continue
        # 처음 방문하고, 얼음이 있는 곳에 대해서 세기
        if not visited[nx][ny] and arr[nx][ny] >= 1:
            visited[nx][ny] = True  # 방문 처리
            result += dfs(nx, ny)
    return result
```

```python
# 가장 큰 연결 요소의 크기 계산
max_value = 0
for i in range(2**n):
    for j in range(2**n):
        # 처음 방문하고, 얼음이 있으면(연결 요소) DFS 수행
        if not visited[i][j] and arr[i][j] >= 1:
            visited[i][j] = True  # 방문 처리
            max_value = max(max_value, dfs(i, j))
print(max_value)
```