Implementation, 구현

시뮬레이션과 완전탐색

완전탐색의 의미는 모든 배열을 순회하는 의미의 완전탐색이 아니라, 2차원 배열 내 좌표를 모두 돌아다니는 것을 의미한다.

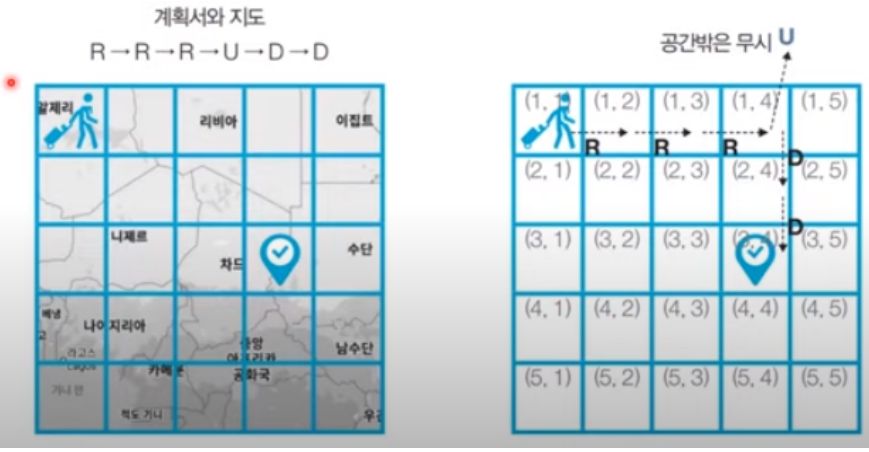
Q1. 상하좌우

〈문제〉 상하좌우: 문제 설명

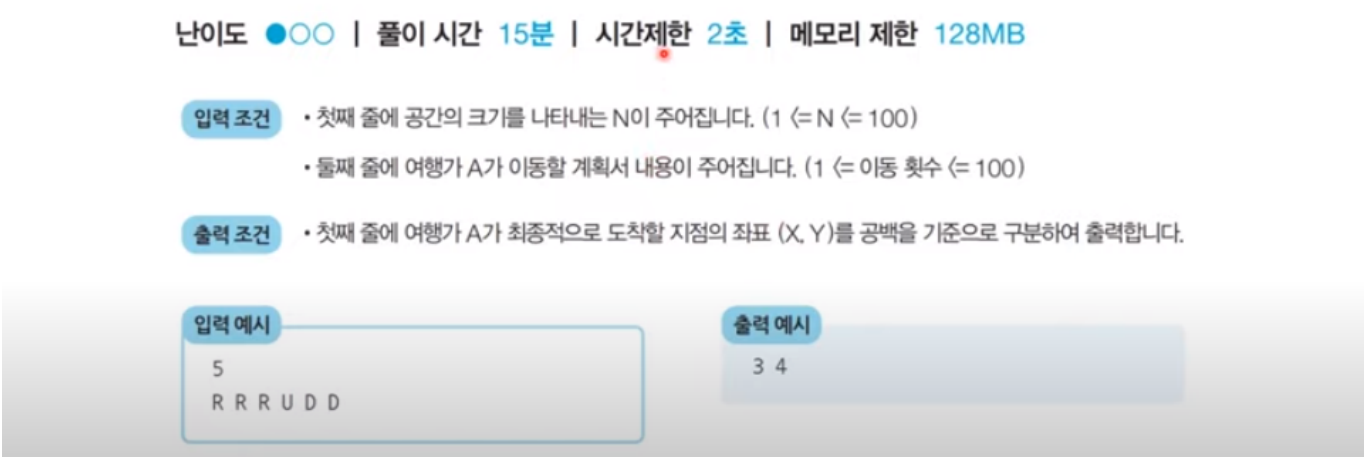
- 여행가 A는 **N × N** 크기의 정사각형 공간 위에 서 있습니다. 이 공간은 1 × 1 크기의 정사각형으로 나누어져 있습니다. 가장 왼쪽 위 좌표는 (1, 1)이며, 가장 오른쪽 아래 좌표는 (N, N)에 해당합니다. 여행가 A는 **상, 하, 좌, 우 방향으로 이동**할 수 있으며, 시작 좌표는 항상 **(1, 1)**입니다. 우리 앞에는 여행가 A가 이동할 계획이 적힌 계획서가 놓여 있습니다.
- 계획서에는 하나의 줄에 띄어쓰기를 기준으로 하여 L, R, U, D 중 하나의 문자가 반복적으로 적혀 있습니다. 각 문자의 의미는 다음과 같습니다.
  - L: 왼쪽으로 한 칸 이동
  - R: 오른쪽으로 한 칸 이동
  - U: 위로 한 칸 이동
  - D: 아래로 한 칸 이동

〈문제〉 상하좌우: 문제 설명

- 이때 여행가 A가 N × N 크기의 정사각형 공간을 벗어나는 움직임은 무시됩니다. 예를 들어 (1, 1)의 위치에서 L 혹은 U를 만나면 무시됩니다. 다음은 N = 5인 지도와 계획서입니다.



〈문제〉 상하좌우: 문제 조건




```python
def imp1(n, input):

    direction = ['L', 'R', 'U', 'D'] # 방향을 지시하는 문자열과
    dx = [0, 0, -1, 1] # 각 방향마다 +- 할 좌표의 움직임을 배열에 담는다.
    dy = [-1, 1, 0, 0]
    x, y = 1, 1 # 시작 좌표도 정한다.
    walk = input.split()

    for w in walk: # 내 이동방향을 돌면서
        for d in range(len(direction)): # 방향 목록중에
            if w == direction[d]: # 해당하는 방향이 나오면
                new_dx = x + dx[d] # 미리 설정한 좌표만큼 더해준다.
                new_dy = y + dy[d]
        # 그러나 밖을 벗어나거나, 더 이상 물러 날 수 없을 때는 넘어간다.
        if new_dx > n or new_dx < 1 or new_dy > n or new_dy < 1:
            continue
        # 새로운 좌표를 기존의 좌표에 대입한다.
        x, y = new_dx, new_dy

    print(x, y)
```

시간 복잡도

- 입력받은 N만큼, 그리고 돌아야 하는 방향만큼 2중 중첩의 반복을 순회해야 하므로 O(N2) 만큼의 시간 복잡도를 가진다.

> Q2. 시각

〈문제〉 시각: 문제 설명

- 정수 N이 입력되면 00시 00분 00초부터 N시 59분 59초까지의 **모든 시각 중에서 3이 하나라도 포함되는 모든 경우의 수를 구하는 프로그램을 작성**하세요. 예를 들어 1을 입력했을 때 다음은 3이 하나라도 포함되어 있으므로 세어야 하는 시각입니다.
  - 00시 00분 03초
  - 00시 13분 30초
- 반면에 다음은 3이 하나도 포함되어 있지 않으므로 세면 안 되는 시각입니다.
  - 00시 02분 55초
  - 01시 27분 45초

〈문제〉 시각: 문제 조건

난이도 ●○○ | 풀이 시간 15분 | 시간제한 2초 | 메모리 제한 128MB

입력 조건 • 첫째 줄에 정수 N이 입력됩니다. (0 <= N <= 23)

출력 조건 • 00시 00분 00초부터 N시 59분 59초까지의 모든 시각 중에서 3이 하나라도 포함되는 모든 경우의 수를 출력합니다.

입력 예시	출력 예시
5	11475

```
def time(n):
    result = 0 # 3이 나타난 횟수
    for h in range(n + 1): # 시간은 0시부터 n시까지
        for m in range(60): # 분, 초는 0부터 59까지
            for s in range(60): # 3이 포함되어 있으면
                if '3' in str(h) or '3' in str(m) or '3' in str(s):
                    result += 1 # result += 1
    print(result)
```

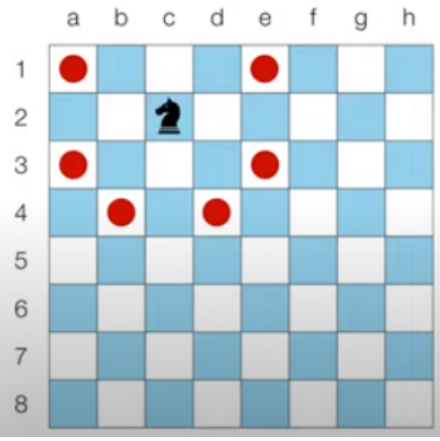
시간복잡도

- 입력값 N이 모든 시, 분, 초에 고정되어 있었다면 O(N3) 만큼의 시간 복잡도를 가진다.

> Q3. 왕실의 나이트

〈문제〉 왕실의 나이트: 문제 설명

- 이처럼 8 × 8 좌표 평면상에서 나이트의 위치가 주어졌을 때 나이트가 이동할 수 있는 경우의 수를 출력하는 프로그램을 작성하세요. 왕실의 정원에서 행 위치를 표현할 때는 1부터 8로 표현하며, 열 위치를 표현할 때는 a부터 h로 표현합니다.
  - c2에 있을 때 이동할 수 있는 경우의 수는 **6가지**입니다.



〈문제〉 왕실의 나이트: 문제 조건

난이도 ●○○ | **풀이 시간** 20분 | **시간 제한** 1초 | **메모리 제한** 128MB

입력 조건 • 첫째 줄에 8 × 8 좌표 평면상에서 현재 나이트가 위치한 곳의 좌표를 나타내는 두 문자로 구성된 문자열이 입력된다. 입력 문자는 a1처럼 열과 행으로 이뤄진다.

출력 조건 • 첫째 줄에 나이트가 이동할 수 있는 경우의 수를 출력하시오.

입력 예시

```
a1
```

출력 예시

```
2
```

```
def knight(input):
    result = 0 # 이동 한 횟수
    # 상우, 우상, 우하, 하우, 하좌, 좌하, 좌상, 상좌에 해당하는 좌표 이동 값
    direction = [(1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1),
(-1, 2)]

    # 가져오는 값의 첫 번째는 문자이므로, 딕셔너리를 통해 위치를 매핑
    replace_direction = {'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5, 'f' : 6, 'g'
: 7, 'h' : 8}
    x, y = replace_direction[list(input)[0]], int(list(input)[1])

    for d in range(len(direction)): # 모든 방향을 돌면서

        # 새로운 x, y 좌표를 각각 더해준다.
        new_dx, new_dy = x + direction[d][0], y + direction[d][1]

        # 이전 완전탐색과는 다르게 8x8 배열 안에서 막히는 곳 없이 모두 해당하는 조건만
```

```
count해야 한다.
        if 1 <= new_dx <= 8 and 1 <= new_dy <= 8:
            result += 1

    print(result)
```

시간 복잡도

- 입력값과 무관하게 정해진 direction 안에서만 순회하므로 O(1)의 시간복잡도를 가진다.

> Q4. 문자열 재정렬

〈문제〉 문자열 재정렬: 문제 설명

- 알파벳 대문자와 숫자(0 ~ 9)로만 구성된 문자열이 입력으로 주어집니다. 이때 모든 알파벳을 오름차순으로 정렬하여 이어서 출력한 뒤에, 그 뒤에 모든 숫자를 더한 값을 이어서 출력합니다.
- 예를 들어 K1KA5CB7이라는 값이 들어오면 ABCKK13을 출력합니다.

〈문제〉 문자열 재정렬: 문제 해결 아이디어

- 요구사항대로 충실히 구현하면 되는 문제입니다.
- 문자열이 입력되었을 때 문자를 하나씩 확인합니다.
  - 숫자인 경우 따로 합계를 계산합니다.
  - 알파벳은 경우 별도의 리스트에 저장합니다.
- 결과적으로 **리스트에 저장된 알파벳을 정렬해 출력하고, 합계를 뒤에 붙여 출력하면 정답**입니다.

```
def replacement(input):
    visited = [] # 문자열을 담고, 정렬 할 배열 생성
    sum = 0 # 숫자 합계를 담을 배열 생성
    for i in input: # 들어오는 문자열을 순회하면서
        if i.isalpha(): # 만약 문자열이면
            visited.append(i) # 빈 배열에 문자열을 대입
        else: # 숫자라면 sum에 합계를 추가
            sum += int(i)
    visited.sort() # 문자열을 오름차순으로 정렬하고
    visited.append(str(sum)) # 숫자를 문자형으로 바꿔 배열에 넣어 준 다음

    print(''.join(visited)) # 하나의 문자로 출력 해 준다.
```

시간 복잡도

- 받아오는 input의 길이만큼 반복하므로 O(N)만큼의 시간 복잡도를 가진다.

구현 템플릿

- 구현 템플릿은 좌표연산과 배열의 입출력, 치환을 고정 템플릿으로 사용한다.

```
# N x N 행렬에서 한 칸씩 움직 일 때는 아래 템플릿을 고정으로 사용한다.
def imp1():

    # 입력값 n은 범위, walk는 방향정보
    n = 5
    walk = ['R', 'R', 'R', 'U', 'D', 'D']

    direction = ['L', 'R', 'U', 'D'] # 방향을 지시하는 문자열
    dx = [0, 0, -1, 1] # 각 방향마다 +- 할 좌표의 움직임
    dy = [-1, 1, 0, 0] # 문제에서 주는 x, y축 == 가로, 세로 축을 알아서 설정한다.
    x, y = 0, 0 # 시작 좌표.

    for w in walk: # 정해진 이동 방향을 돌면서
        for d in range(len(direction)): # 방향 목록중에
            if w == direction[d]: # 해당하는 방향이 나오면
                new_dx = x + dx[d] # 미리 설정한 좌표만큼 더해준다.
                new_dy = y + dy[d]

        # 밖을 벗어나거나, 더 이상 물러 날 수 없을 때는 넘어간다.
        if new_dx > n or new_dx < 1 or new_dy > n or new_dy < 1:
            continue
        # 새로운 좌표를 기존의 좌표에 대입한다.
        x, y = new_dx, new_dy

    print(x, y)
```

```
# 왕실의 나이트 문제의 경우 1칸 씩 이동하는게 아닌, 이동 범위가 주어 진 경우 직접 할당한다.
# 상우, 우상, 우하, 하우, 하좌, 좌하, 좌상, 상좌에 해당하는 좌표 이동 값
direction = [(1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1), (-1, 2)]

# 이 후 한 칸씩 이동하는 형태와 같이 이동하는 좌표와 범위 연산을 수행한다.
```

```
# 문자인지 숫자인지 체크하는 isalpha()는 문자 + 숫자가 혼합된 문자열 처리에서 유용하다.
# ord(문자)는 문자에 해당하는 유니코드 정수를,
# chr(정수)는 문자에 해당하는 유니코드 문자를 반환하므로 이 또한 문자열 처리에서 유용하다.
# 문자열 연산이나 탐색의 경우, 탐색 한 문자열을 담을 수 있는 배열을 선언한다.
def replacement():
    # 입력받는 input은 문자와 숫자의 혼합이다.
    input = '1KK23PD'

    visited = [] # 문자열을 담고, 정렬 할 배열 생성
```

```
    for i in input: # 들어오는 문자열을 순회하면서

        # some logic
        # cur_node =  ord(i) # 문자에 대응하는 숫자가 필요하다면
        # cur_node =  chr(i) # 숫자에 대응하는 문자가 필요하다면

        if i.isalpha(): # 만약 문자열이면
            visited.append(i) # 빈 배열에 문자열을 대입
            # visited.pop() # pop, insert, del 등의 배열 입출력 연산을 이용한다.
        else:
            # some logic
            continue
```