1003. fibonacci.c

fibonacci 함수를 단순 재귀호출시키면 너무 많은 시간이 소요됨.

시간 초과(백준), VSCODE에서는 컴파일은 되나 역시 많은 시간이 필요.

이 때는 복잡한 문제를 나누어 해결하는 동적 계획법(Dynamic Programming)의 방법중 하나인

메모이제이션(Memoization) 기법을 사용하면

재귀호출로 반복되는 $O(2^n)$의 실행횟수가 $O(n)$으로 줄어든다.

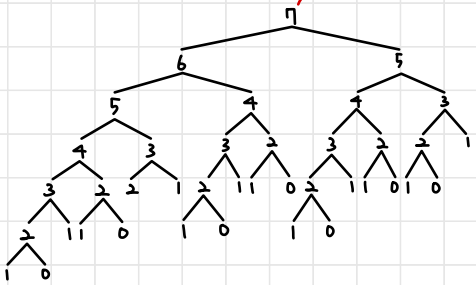
본래 피보나치 재귀호출 함수(return 0와 return 1 횟수 세기)

```
void fi(int n, int *z, int *o){
    if(n==0) (*z)++;
    else if(n==1) (*o)++;
    else fi(n-1, z, o)+fi(n-2, z, o);
}
```

개별적으로 무수히 많은 재귀호출 실행

ex) fi(7)인 경우

n이 커질수록, 재귀호출은 걷잡을 수 없이 커짐



따라서 n이 커질수록 스택(Stack) 영역이 부족해짐.

→ 메모리 낭비 + 재귀호출로 인한 실행속도 매우 나쁨.

이 때 Memoization 기법을 사용.

Memoization(메모이제이션)

반복 계산 시, 계산한 결과를 기억해두었다가 이후 같은 연산 시 기억해둔 값을 이용

조건 : N은 40 이하의 자연수

int memo[41] ← 0 ~ 40까지 인덱스의 결괏값을 저장하는 배열.

이번 문제에서는 zero[41] 배열과 one[41] 배열 생성.

변경된 피보나치 함수

```
int zero[41];
int one[41];
```
전역변수, 자동 초기화

```
void fi(int n) {
    if(n==0) zero[n]=1;
    if(n==1) one[n]=1;
    if(zero[n]==0 && one[n]==0) {
        fi(n-2);
        fi(n-1);
        zero[n]=zero[n-1]+zero[n-2];
        one[n]=one[n-1]+one[n-2];
    }
```

Base case

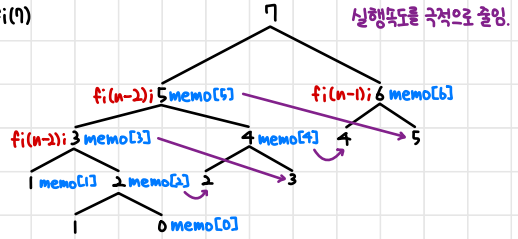
→ 재귀호출을 반복하여 작은 인덱스부터 채워나감.

→ 이전에 저장된 인덱스가 있음.

인덱스를 저장하는 알고리즘.

ex) fi(7)

실행속도를 극적으로 줄임.



1018. Chess.py

체스판의 크기는 m × n이고, 이 보드를 잘라 8×8 크기의 체스판을 만드려고 한다.

→ 중첩 for문의 range를 n-7, m-7로 잘라 보드의 시작점을 고정하고,

이후의 중첩 for문의 range를 (i, i+8), (j, j+8)로 두어 8, 8만큼 index를 증가시키며 탐색한다.

ex) n=10, m=13일 때

```
for i in range(n-7)      # 0~3  ┐ 시작위치 지정
    for j in range(m-7)  # 0~6  ┘
        ⋮
                                  3, 10 ┐ 시작위치~시작위치+7
    for a in range(i, i+8)            ┘ = 0~7(8개의 index)
        for b in range(j, j+8)        6, 13
```

이후 반복문을 (n-7)·(m-7)·8·8번 반복하며 무늬를 체크한다.

(a+b)%2==0를 검사하여 현재 인덱스의 위치가 짝수/홀수인지 판단하는데,

초기값을 짝수가 흰색무늬(white), 홀수가 흑색무늬(black)인 것으로 설정하면,

현재위치(짝수 인덱스)가 'W'가 아닐 때 white 변수를 1 증가시켜 바꿔야 하는 무늬값을 증가시킨다 (반대도 동일)

따라서 자연스레 짝수 인덱스가 흑색이라고 가정하면(else) black이 1 증가하게 된다.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3

(a+b)%2==0, 짝수가 흰색이면

자연스럽게 홀수는 흑색.

* 짝수를 흰색으로 설정하였는데 W가 아니면 white 1 증가

else → 짝수를 흑색으로 설정하였는데 B가 아니면 black 1 증가

이렇게 white와 black 변수를 모두 증가시킨 후, 어느 무늬로 먼저 시작했는지와 관계없이 (단순 최소값 계산)

모두 리스트에 넣어준 후, 마지막에 리스트 중 최소값을 출력해주면 된다.

```python
n, m = map(int, input().split()) # m*n 정수 입력
chess = [] # 보드의 무늬(문자열)을 입력받을 리스트 생성
for i in range(n): # n(높이)만큼 문자열을 입력받아
    chess.append(input()) # chess 리스트에 할당함(append)
startColor = [] # 시작 무늬에 따른 무늬 변경횟수를 저장할 startColor 리스트 생성

for i in range(n-7):
    for j in range(m-7): # 보드를 8*8 크기로 만듦(n-7 = 0~7의 인덱스를 가짐)
        white = 0
        black = 0 # 보드의 왼쪽 위의 시작점부터 변경해야 하는 무늬를 기록하기 위한 변수 생성
        for a in range(i, i+8):
            for b in range(j, j+8): # 0~7의 인덱스(8*8)를 체크함
                if(a+b)%2==0: # 짝수 인덱스 위치에 있을 때
                    if chess[a][b]!='W': # 흰색 무늬가 아니면
                        white+=1 # 무늬를 변경해야 하므로 white를 1 증가
                    else: # 반대로 흑색 무늬가 아니면
                        black+=1 # 무늬를 변경해야 하므로 black을 1 증가
                else:
                    if chess[a][b]!='B':
                        white+=1
                    else:
                        black+=1 # 홀수 인덱스 위치에 있을 때도 흰색/흑색에 따라 체크
        startColor.append(white) # 흰색 무늬로 시작했을 때 변경횟수를 리스트에 저장
        startColor.append(black) # 흑색 무늬로 시작했을 때 변경횟수를 리스트에 저장
print(min(startColor)) # 리스트에 있는 변경횟수 중 최소값을 출력
```

* 위 문제는 브루트포스(Brute-Force) 알고리즘을 사용한다.

다른 말로는 '무차별 대입', '완전탐색'이라 부르며, 가능한 모든 경우의 수를 체크하는

난폭한(Brute), 비효율적이지만 정확도 100%를 가지는 알고리즘이다.