화성

최초로 우주 탐험을 한 사람들이 고대 이집트의 과학자들이라는 것은 잘 알려진 사실이다. 이집트인들은 투트무스 1 행성 (요즘은 화성이라고 부르는 행성)을 탐험하는 우주선을 발사했다. 화성의 표면은 $(2n+1) \times (2n+1)$ 칸으로 이루어진 격자로 표현할 수 있는데, 각 칸은 땅 또는 물이다. i 행 j 열 ($0 \le i, j \le 2 \cdot n$)에 있는 칸이 땅이라면 $s[i][j] =$'1' 이고, 물이라면 $s[i][j] =$'0' 이다.

땅을 나타내는 두 칸은, 이 둘 사이에 땅으로만 이루어진 경로가 존재한다면 연결되어 있다고 한다. 이 때 경로에 포함되는 인접한 두 칸은 변을 공유해야 한다. 화성의 섬은 땅을 나타내는 칸들의 최대 집합(maximal set)인데, 이 집합에 포함되는 어떤 두 칸도 서로 연결되어야 한다.

우주선의 임무는 화성의 섬의 개수를 세는 것이다. 이 임무는 쉽지 않은데, 고대 기술로 만든 컴퓨터라 성능에 제약이 있기 때문이다. 컴퓨터의 메모리 h는 $(2n+1) \times (2n+1)$ 크기의 2차원 배열 형태이다. 메모리의 한 칸에는길이가 100인 이진수 문자열을 저장할 수 있다. 문자열의 각 글자는 '0' (ASCII 48) 또는 '1' (ASCII 49) 이다. 처음에는, 메모리의 각 칸의 첫 비트는 해당하는 화성 지표면의 상태를 나타낸다. 즉 $h[i][j][0] = s[i][j]$ ($0 \le i, j \le 2 \cdot n$)이다. h의 다른 비트는 처음에는 '0' (ASCII 48)이다.

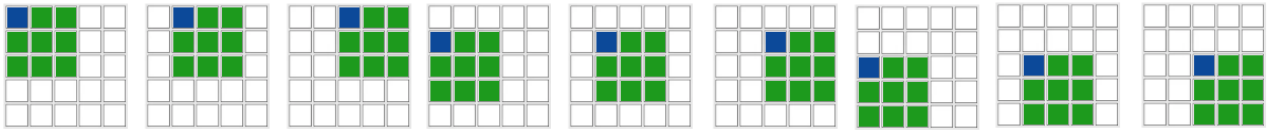
메모리에 저장된 데이터를 처리하기 위해서, 컴퓨터는 메모리에서 오직 3×3 부분을 읽을 수 있고, 이 부분의 가장 왼쪽 위 칸에 값을 덮어 쓸 수 있다. 엄밀하게 말하면, 컴퓨터는 $h[i..i+2][j..j+2]$ ($0 \le i, j \le 2 \cdot (n-1)$)의 값들을 읽을 수 있고, $h[i][j]$에 값을 쓸 수 있다. 이 과정을 **칸 (i, j)을 처리한다**라고 부르기로 하자.

컴퓨터의 한계를 극복하기 위해서, 고대 이집트 과학자들은 다음과 같이 하기로 했다.

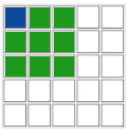
- 컴퓨터는 메모리를 n 단계로 나누어서 처리한다.
- 단계 k에서 $(0 \le k \le n-1)$, $m = 2 \cdot (n-k-1)$이라고 하자. 컴퓨터는 모든 $0 \le i, j \le m$에 대해서 칸 (i, j)을 처리하는데, i의 오름차순으로, i가 같으면 j의 오름차순으로 진행한다. 다르게 표현하면, 컴퓨터는 다음 순서로 칸들을 처리한다: $(0,0), (0,1), \cdots, (0,m)$, $(1,0), (1,1), \cdots, (1,m), \cdots, (m,0), (m,1), \cdots, (m,m)$.
- 마지막 단계에서 $(k = n-1)$, 컴퓨터는 칸 $(0,0)$을 처리한다. 이 단계를 마쳤을 때 $h[0][0]$의 값은 화성의 섬의 개수를 이진수로 표현한 값과 같아야 한다. 숫자의 가장 낮은 자릿수가 이 문자열의 첫 글자가 되는 식으로 저장된다.

다음 그림은 5×5 $(n = 2)$ 크기 메모리를 가진 컴퓨터가 메모리를 처리하는 과정을 보여준다. 파란 칸은 값이 덮어씌워지는 칸이고, 색깔이 칠해진 칸은 처리되는 부분 배열을 표시한다.

단계 0에서, 컴퓨터는 다음 순서로 부분 배열들을 처리한다.



단계 1, 컴퓨터는 단 한 개의 부분 배열을 처리한다.



당신이 할 일은 위와 같이 동작하는 컴퓨터를 이용하여 화성에 있는 섬의 수를 세는 방법을 구현하는 것이다.

상세 구현

다음 함수를 구현해야 한다.

```
string process(string[][] a, int i, int j, int k, int n)
```

- a: 3×3 크기 배열로, 처리되는 부분 배열을 나타낸다. 구체적으로는, $a = h[i..i+2][j..j+2]$이고, a의 각 원소는 정확히 길이가 100인 문자열로 각 글자는 `'0'` (ASCII 48) 또는 `'1'` (ASCII 49)이다.
- i, j: 현재 컴퓨터가 처리하는 칸의 행과 열
- k: 현재 단계
- n: 전체 단계의 수이면서, 화성 표면의 크기 $(2n+1) \times (2n+1)$을 나타내는 값
- 이 함수는 길이가 100인 이진수 문자열을 리턴해야 한다. 리턴값은 컴퓨터의 메모리 $h[i][j]$에 저장된다.
- $k = n-1$일 때 이 함수를 마지막으로 호출한다. 이 호출의 리턴값은 화성에 있는 섬의 수를 이진수로 표현한 문자열인데, 이진수의 가장 낮은 자릿수(least significant bit)가 0번 위치의 글자이고, 그 다음 낮은 자릿수가 1번 위치 글자인 식으로 저장된다.
- 이 함수는 어떤 정적 변수 또는 전역 변수와도 무관해야 하며, 리턴값은 이 함수에 전달된 파라미터에만 의존해야 한다.

각 테스트 케이스는 T개의 독립적인 시나리오(즉, 서로 다른 행성의 표면)로 이루어진다. 각 시나리오에서 당신이 구현한 함수의 동작은 시나리오들을 수행하는 순서와 무관해야 하는데, 같은 시나리오에 속한 `process` 함수 호출이 연속적으로 일어나지 않을 수 있기 때문이다. 그러나, 각각의 시나리오에서 `process` 함수의 호출 순서는 문제 설명에서 설명한 것과 같다.

특히, 각각의 테스트케이스에서, 여러분의 프로그램의 여러 인스턴스가 동시에 수행될 것이다. 메모리와 CPU 시간의 제약은 각 인스턴스의 사용량을 합친 것이다. 인스턴스 사이에서 데이터를 불법으로 교환하려는 시도는 부정행위로 간주되며, 실격될 수 있다.

특히, `process` 함수를 호출하는 동안 정적 변수나 전역 변수에 저정한 값들을 다음 호출때 사용하지 못할 수 있다.

제약조건

- $1 \le T \le 10$
- $1 \le n \le 20$
- $s[i][j]$는 `'0'`(ASCII 48)이거나 `'1'`(ASCII 49) (for all $0 \le i, j \le 2 \cdot n$)
- $h[i][j]$의 길이는 정확히 100 (for all $0 \le i, j \le 2 \cdot n$)
- $h[i][j]$의 글자들은 `'0'` (ASCII 48)이거나 `'1'` (ASCII 49) (for all $0 \le i, j \le 2 \cdot n$).

`process` 함수를 매번 호출할 때:

- $0 \le k \le n-1$
- $0 \le i, j \le 2 \cdot (n-k-1)$

부분 문제

1. (6 점) $n \le 2$
2. (8 점) $n \le 4$
3. (7 점) $n \le 6$
4. (8 점) $n \le 8$
5. (7 점) $n \le 10$
6. (8 점) $n \le 12$
7. (10 점) $n \le 14$
8. (24 점) $n \le 16$
9. (11 점) $n \le 18$
10. (11 점) $n \le 20$

예제

예제 1

$n = 1$이고 s가 다음과 같은 경우를 생각해보자.

```
'1' '0' '0'
'1' '1' '0'
'0' '0' '1'
```

이 예제에서, 행성의 표면은 3×3 칸들과 2개의 섬으로 이루어져 있다. `process` 함수를 호출하는 단계는 오직 하나이다.

단계 0에서, 그레이더는 `process` 함수를 정확히 한 번 호출한다:

```
process([["100","000","000"],["100","100","000"],["000","000","100"]],0,0,0,1)
```

h 각 칸의 처음 세 비트만 표시했다는데 주의하시오.

이 함수의 리턴값은 `"0100..."`(생략된 비트는 모두 0)이어야 하는데, 2진수 ....0010는 10진수 2이다. 96개의 0을 생략하고 `...`로 대치했다는데 주의하시오.

예제 2

$n = 2$이고 s가 다음과 같은 경우를 생각해보자.

```
'1' '1' '0' '1' '1'
'1' '1' '0' '0' '0'
'1' '0' '1' '1' '1'
'0' '1' '0' '0' '0'
'0' '1' '1' '1' '1'
```

이 예제에서, 행성의 표면은 5×5 칸과 4개의 섬으로 이루어져 있다. `process` 함수를 두 단계에 걸쳐 호출한다.

단계 0에서, 그레이더는 `process` 함수를 9 번 호출한다.

```
process([["100","100","000"],["100","100","000"],["100","000","100"]],0,0,0,2)
process([["100","000","100"],["100","000","000"],["000","100","100"]],0,1,0,2)
process([["000","100","100"],["000","000","000"],["100","100","100"]],0,2,0,2)
process([["100","100","000"],["100","000","100"],["000","100","000"]],1,0,0,2)
process([["100","000","000"],["000","100","100"],["100","000","000"]],1,1,0,2)
process([["000","000","000"],["100","100","100"],["000","000","000"]],1,2,0,2)
process([["100","000","100"],["000","100","000"],["000","100","100"]],2,0,0,2)
process([["000","100","100"],["100","000","000"],["100","100","100"]],2,1,0,2)
process([["100","100","100"],["000","000","000"],["100","100","100"]],2,2,0,2)
```

위 함수의 리턴값들이 각각 `"011", "000", "000", "111", "111", "011", "110", "010", "111"`이라고 하자. 생략된 비트는 모두 0이다. 따라서 단계 0이 끝난 뒤, h에는 다음 값들이 저장된다.

```
"011", "000", "000", "100", "100"
"111", "111", "011", "000", "000"
"110", "010", "111", "100", "100"
"000", "100", "000", "000", "000"
"000", "100", "100", "100", "100"
```

단계 1에서, 그레이더는 `process` 함수를 한 번 호출한다.

```
process([["011","000","000"],["111","111","011"],["110","010","111"]],0,0,1,2)
```

마지막으로, 이 함수의 리턴값은 `"0010000...."` (생략된 비트는 모두 0)이어야 하는데, 2진수 $....0000100$는 10진수 4이다. 93개의 0을 생략하고 `...`로 대치했다는데 주의하시오.

샘플 그레이더

샘플 그레이더는 다음 형식으로 입력을 읽는다:

- line 1: T
- block i ($0 \le i \le T-1$): 시나리오 i를 표현하는 block.
  - line 1: n
  - line $2+j$ ($0 \le j \le 2 \cdot n$): $s[j][0]$ $s[j][1]$ ... $s[j][2 \cdot n]$

샘플 그레이더는 다음 형식으로 출력한다:

- line $1 + i$ ($0 \le i \le T - 1$): i번째 시나리오에서 마지막으로 호출된 `process` 함수의 리턴값을 10진수로 표현한 것