PROBLEM 1
Linked List

Implement an algorithm to delete a node in the middle of a singly linked list, given only access to that node.

EXAMPLE
Input: the node c from the linked list a->b->c->d->e
Result: nothing is returned, but the new linked list looks like a- >b- >d->e

---

PROBLEM 2
Tower of Hanoi

In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

(1) Only one disk can be moved at a time.
(2) A disk is slid off the top of one tower onto the next tower.
(3) A disk can only be placed on top of a larger disk.
Write a program to move the disks from the first tower to the last using stacks.

PROBLEM 3
Dominator - Find an index of an array such that its value occurs at more than half of indices in the array.

A zero-indexed array A consisting of N integers is given. The *dominator* of array A is the value that occurs in more than half of the elements of A.
For example, consider array A such that
A[0] = 3 A[1] = 4 A[2] = 3
A[3] = 2 A[4] = 3 A[5] = -1
A[6] = 3 A[7] = 3
The dominator of A is 3 because it occurs in 5 out of 8 elements of A (namely in those with indices 0, 2, 4, 6 and 7) and 5 is more than a half of 8.
Write a function
```
class Solution { public int solution(int[] A); }
```

that, given a zero-indexed array A consisting of N integers, returns index of any element of array A in which the dominator of A occurs. The function should return −1 if array A does not have a dominator.
Assume that:

- N is an integer within the range [0..100,000];
- each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].

For example, given array A such that
A[0] = 3 A[1] = 4 A[2] = 3
A[3] = 2 A[4] = 3 A[5] = -1
A[6] = 3 A[7] = 3
the function may return 0, 2, 4, 6 or 7, as explained above.
Complexity:

- expected worst-case time complexity is O(N);
- expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

PROBLEM 4
Ladder - Count the number of different ways of climbing to the top of a ladder

You have to climb up a ladder. The ladder has exactly N rungs, numbered from 1 to N. With each step, you can ascend by one or two rungs. More precisely:

- with your first step you can stand on rung 1 or 2,
- if you are on rung K, you can move to rungs K + 1 or K + 2,
- finally you have to stand on rung N.

Your task is to count the number of different ways of climbing to the top of the ladder.
For example, given N = 4, you have five different ways of climbing, ascending by:

- 1, 1, 1 and 1 rung,
- 1, 1 and 2 rungs,
- 1, 2 and 1 rung,
- 2, 1 and 1 rungs, and
- 2 and 2 rungs.

Given N = 5, you have eight different ways of climbing, ascending by:

- 1, 1, 1, 1 and 1 rung,
- 1, 1, 1 and 2 rungs,
- 1, 1, 2 and 1 rung,
- 1, 2, 1 and 1 rung,
- 1, 2 and 2 rungs,
- 2, 1, 1 and 1 rungs,

- 2, 1 and 2 rungs, and
- 2, 2 and 1 rung.

The number of different ways can be very large, so it is sufficient to return the result modulo 2^P, for a given integer P.
Write a function:
`class Solution { public int[] solution(int[] A, int[] B); }`
that, given two non-empty zero-indexed arrays A and B of L integers, returns an array consisting of L integers specifying the consecutive answers; position I should contain the number of different ways of climbing the ladder with A[I] rungs modulo $2^{B[I]}$.
For example, given L = 5 and:
A[0] = 4 B[0] = 3
A[1] = 4 B[1] = 2
A[2] = 5 B[2] = 4
A[3] = 5 B[3] = 3
A[4] = 1 B[4] = 1
the function should return the sequence [5, 1, 8, 0, 1], as explained above.
Assume that:

- L is an integer within the range [1..30,000];
- each element of array A is an integer within the range [1..L];
- each element of array B is an integer within the range [1..30].

Complexity:

- expected worst-case time complexity is O(L);
- expected worst-case space complexity is O(L), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

PROBLEM 5
Game Of Life

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours comes to life.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules Simultaneously to every cell in the seed — births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

Problem.

The inputs below represent the cells in the universe as X or - . X is a alive cell. - is a dead cell or no cell. The below inputs provide the provide pattern or initial cells in the universe. The output is the state of the system in the next tick (one run of the application of all the rules), represented in the same format.

Input A:
(Block pattern)

```
X X
X X
```

Output A:

```
X X
X X
```

Input B
(Boat pattern)

```
X X -
X - X
- X -
```

Output B

```
X X -
X - X
- X -
```

Input C
(Blinker pattern)

```
- X -
- X -
- X -
```

Output C

```
- - -
X X X
- - -
```

Input D
(Toad pattern)

```
- X X X
X X X -
```

Output D

```
- - X -
X - - X
X - - X
- X - -
```