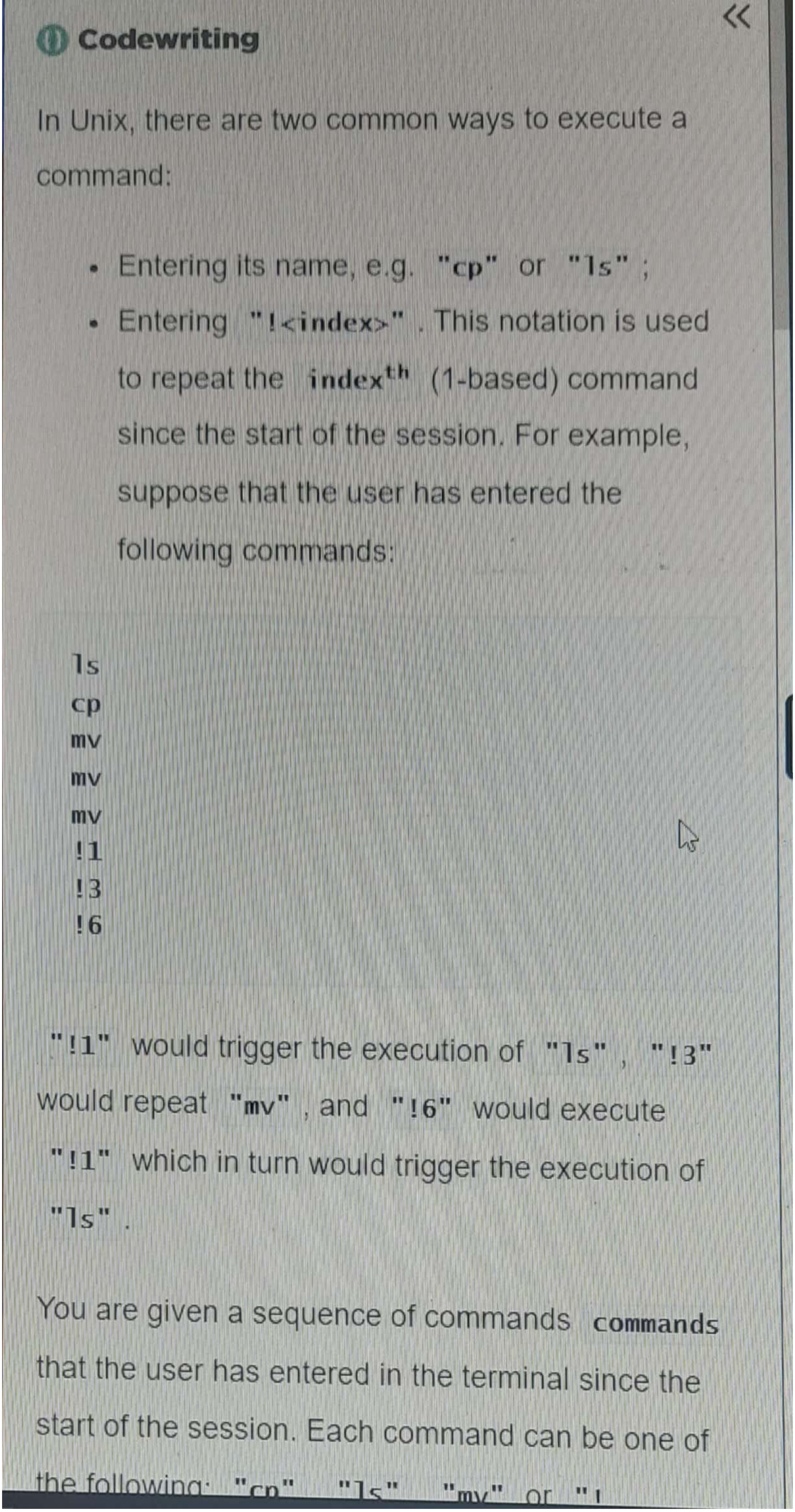


Codewriting

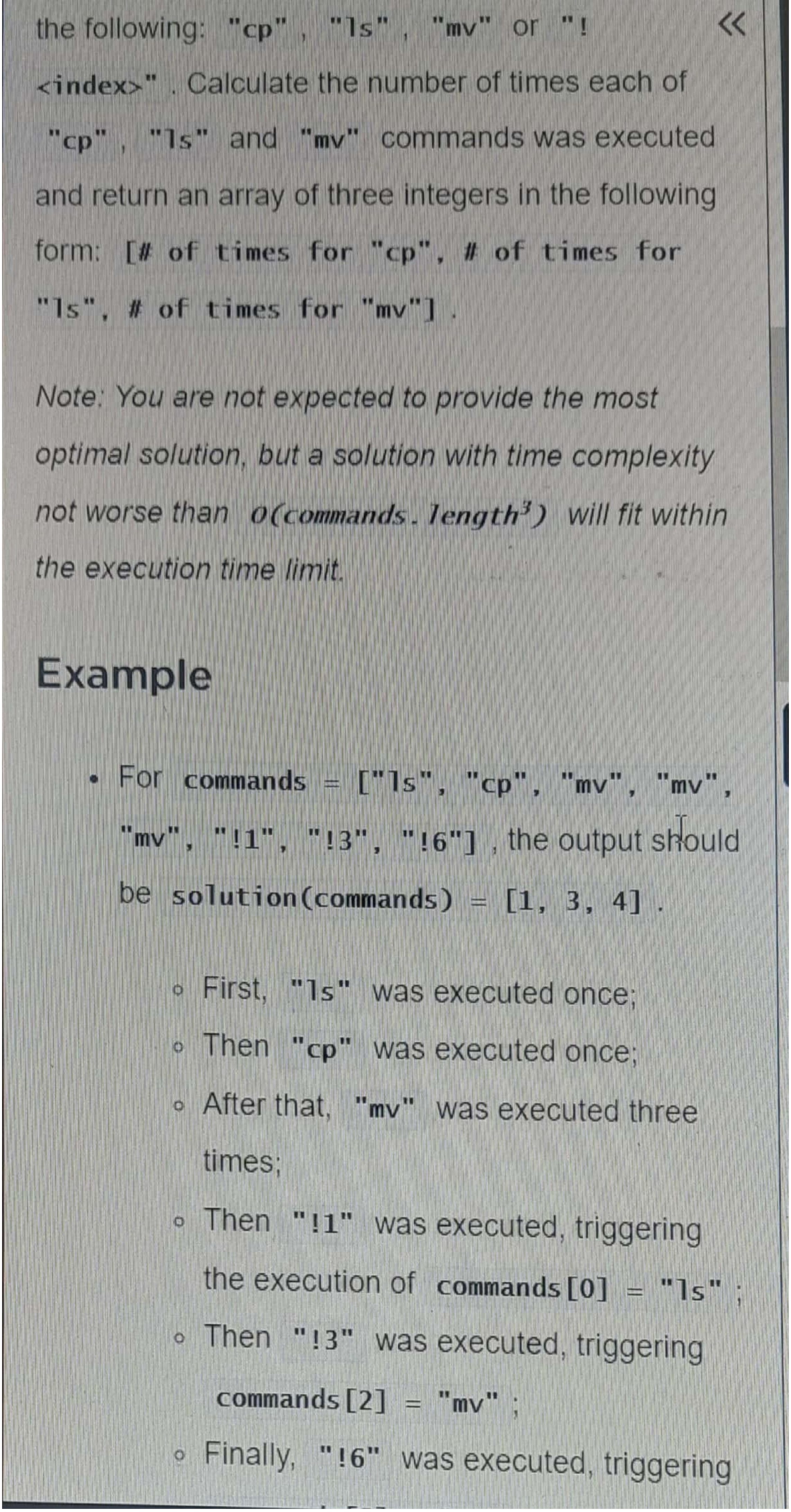
In Unix, there are two common ways to execute a command:

- Entering its name, e.g. `"cp"` or `"ls"` ;
- Entering `"!<index>"` . This notation is used to repeat the `index`th (1-based) command since the start of the session. For example, suppose that the user has entered the following commands:

```
ls
cp
mv
mv
mv
!1
!3
!6
```

`"!1"` would trigger the execution of `"ls"` , `"!3"` would repeat `"mv"` , and `"!6"` would execute `"!1"` which in turn would trigger the execution of `"ls"` .

You are given a sequence of commands `commands` that the user has entered in the terminal since the start of the session. Each command can be one of the following: `"cp"` , `"ls"` , `"mv"` or `"!`

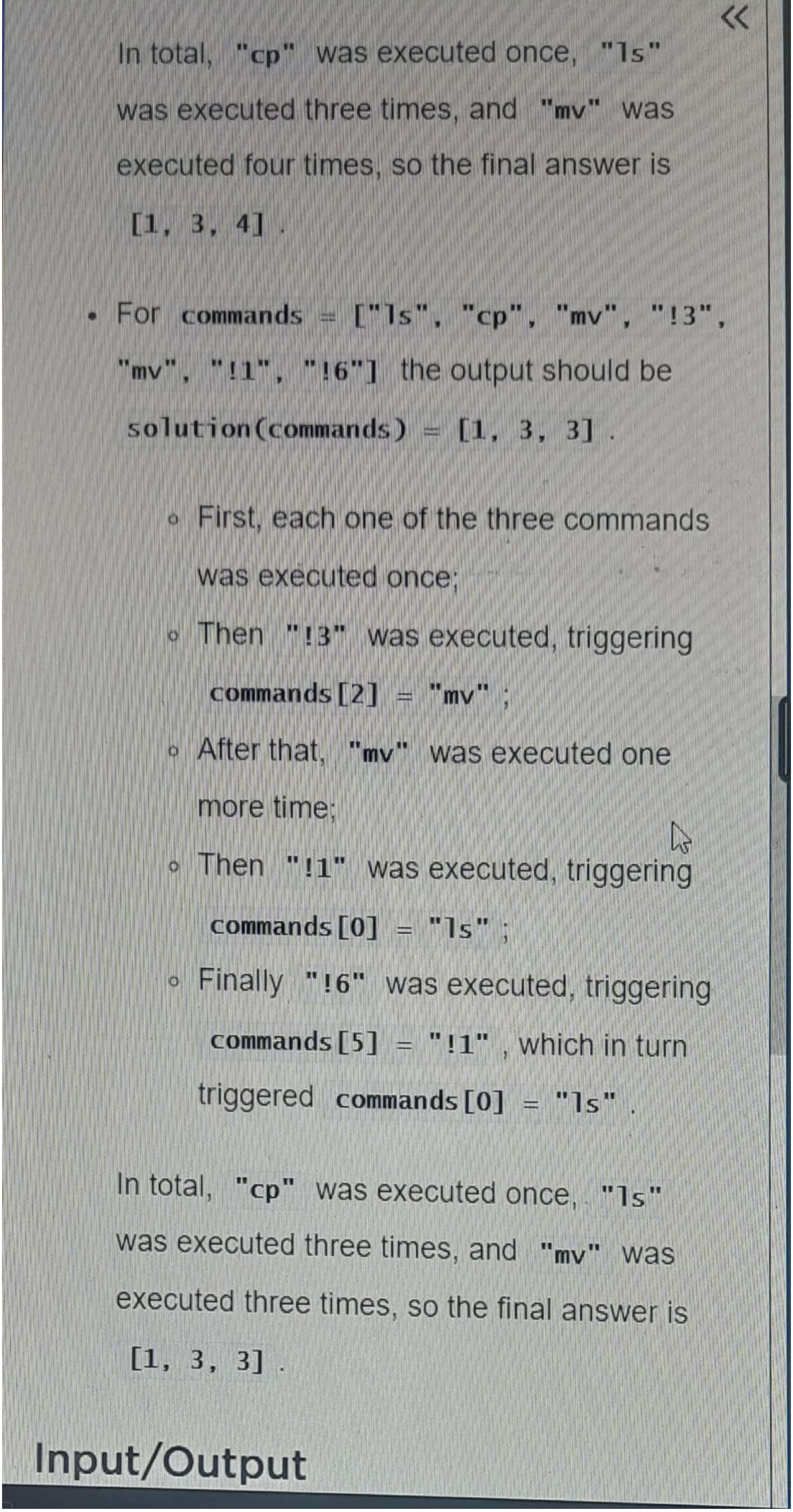


the following: `"cp"` , `"ls"` , `"mv"` or `"!<index>"` . Calculate the number of times each of `"cp"` , `"ls"` and `"mv"` commands was executed and return an array of three integers in the following form: `[# of times for "cp", # of times for "ls", # of times for "mv"]` .

Note: You are not expected to provide the most optimal solution, but a solution with time complexity not worse than $O(commands.length^3)$ *will fit within the execution time limit.*

Example

- For `commands = ["ls", "cp", "mv", "mv", "mv", "!1", "!3", "!6"]` , the output should be `solution(commands) = [1, 3, 4]` .
  - First, `"ls"` was executed once;
  - Then `"cp"` was executed once;
  - After that, `"mv"` was executed three times;
  - Then `"!1"` was executed, triggering the execution of `commands[0] = "ls"` ;
  - Then `"!3"` was executed, triggering `commands[2] = "mv"` ;
  - Finally, `"!6"` was executed, triggering

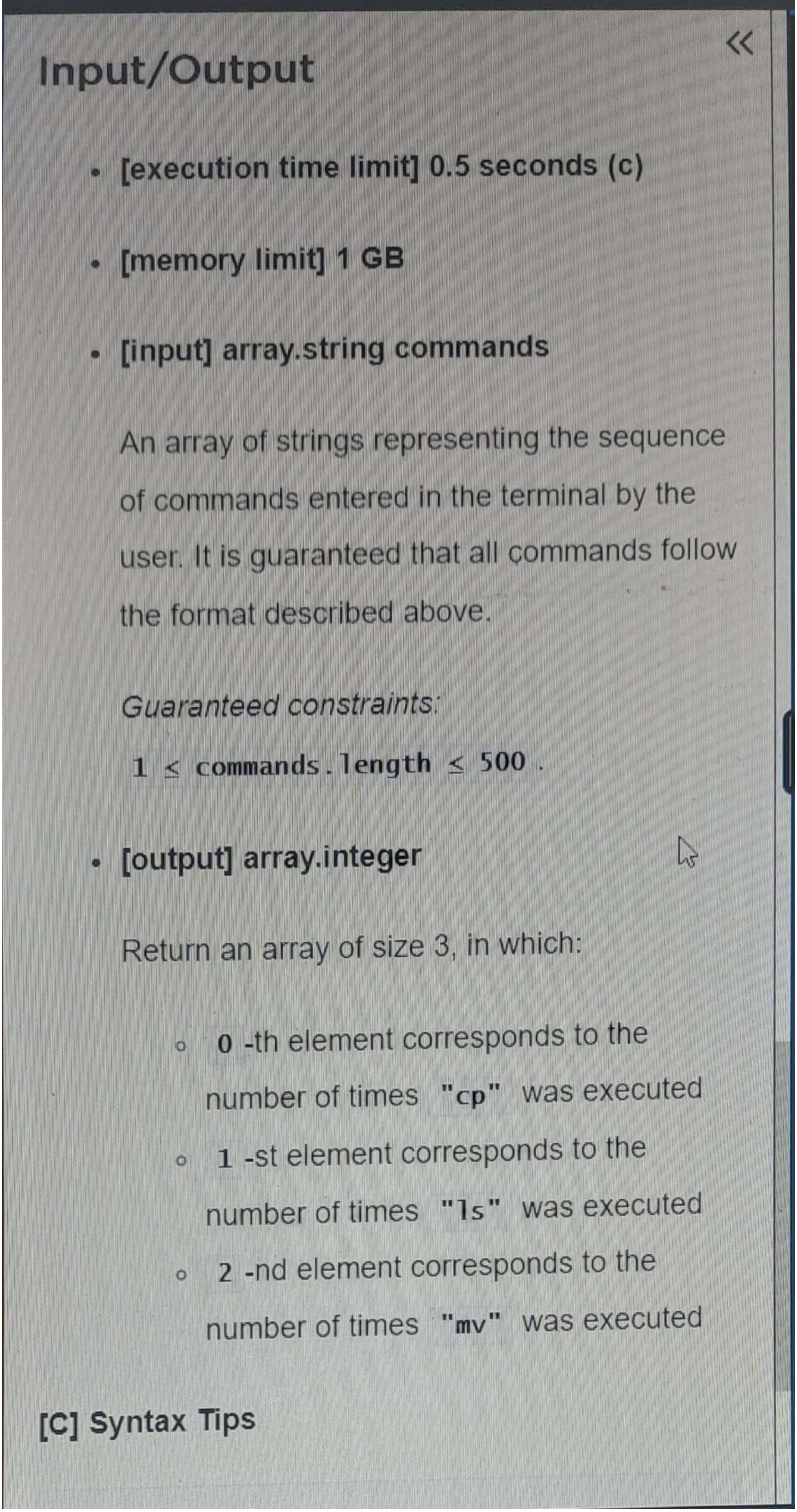


In total, `"cp"` was executed once, `"ls"` was executed three times, and `"mv"` was executed four times, so the final answer is `[1, 3, 4]`.

- For `commands = ["ls", "cp", "mv", "!3", "mv", "!1", "!6"]` the output should be `solution(commands) = [1, 3, 3]`.
  - First, each one of the three commands was executed once;
  - Then `"!3"` was executed, triggering `commands[2] = "mv"`;
  - After that, `"mv"` was executed one more time;
  - Then `"!1"` was executed, triggering `commands[0] = "ls"`;
  - Finally `"!6"` was executed, triggering `commands[5] = "!1"`, which in turn triggered `commands[0] = "ls"`.

In total, `"cp"` was executed once, `"ls"` was executed three times, and `"mv"` was executed three times, so the final answer is `[1, 3, 3]`.

Input/Output



Input/Output

- **[execution time limit] 0.5 seconds (c)**

- **[memory limit] 1 GB**

- **[input] array.string commands**

  An array of strings representing the sequence of commands entered in the terminal by the user. It is guaranteed that all commands follow the format described above.

  Guaranteed constraints:

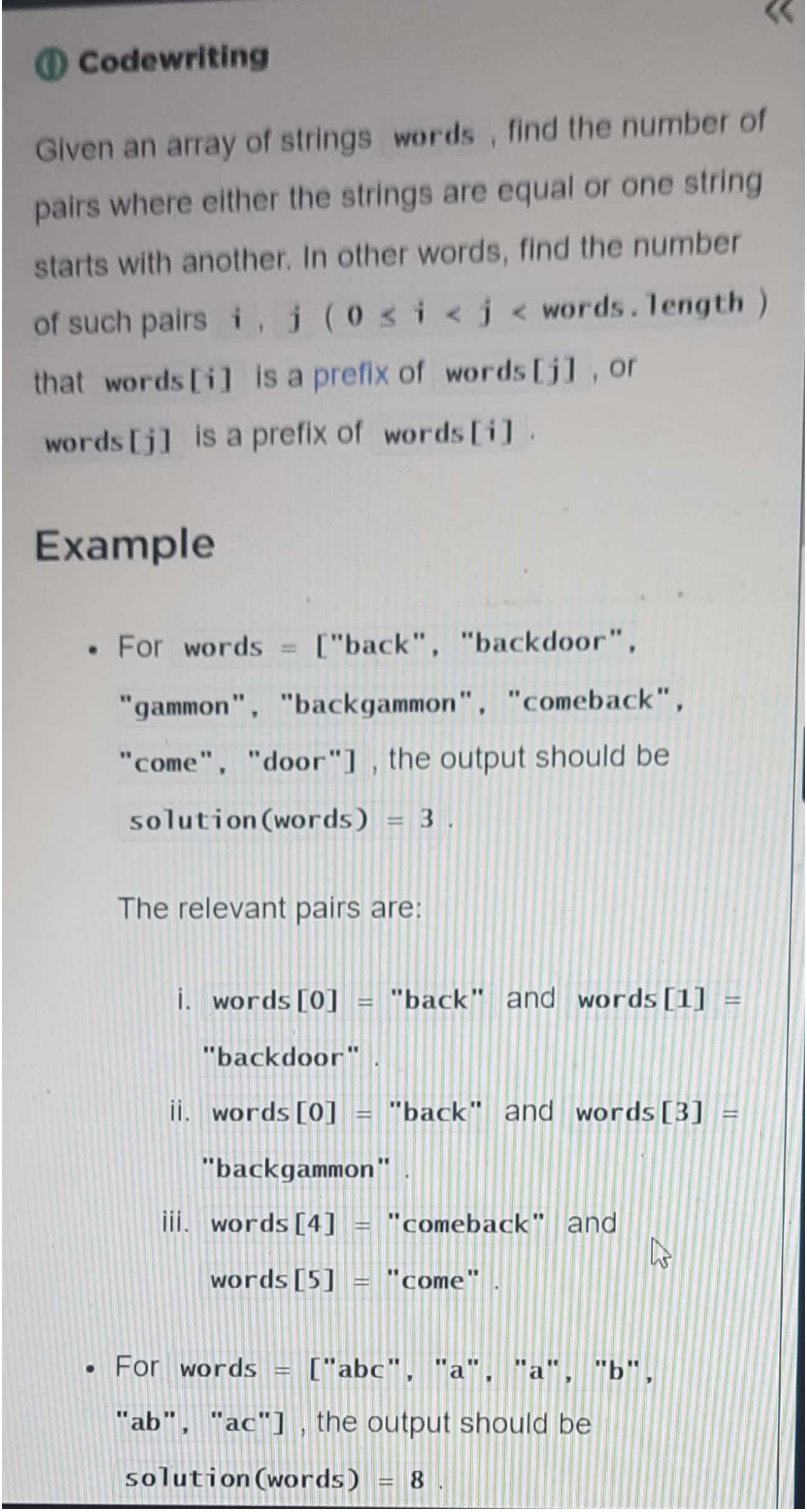
  `1 ≤ commands.length ≤ 500` .

- **[output] array.integer**

  Return an array of size 3, in which:

  - `0` -th element corresponds to the number of times `"cp"` was executed
  - `1` -st element corresponds to the number of times `"ls"` was executed
  - `2` -nd element corresponds to the number of times `"mv"` was executed

[C] Syntax Tips



Codewriting

Given an array of strings `words`, find the number of pairs where either the strings are equal or one string starts with another. In other words, find the number of such pairs `i`, `j` (`0 ≤ i < j < words.length`) that `words[i]` is a prefix of `words[j]`, or `words[j]` is a prefix of `words[i]`.

Example

- For `words = ["back", "backdoor", "gammon", "backgammon", "comeback", "come", "door"]`, the output should be `solution(words) = 3`.

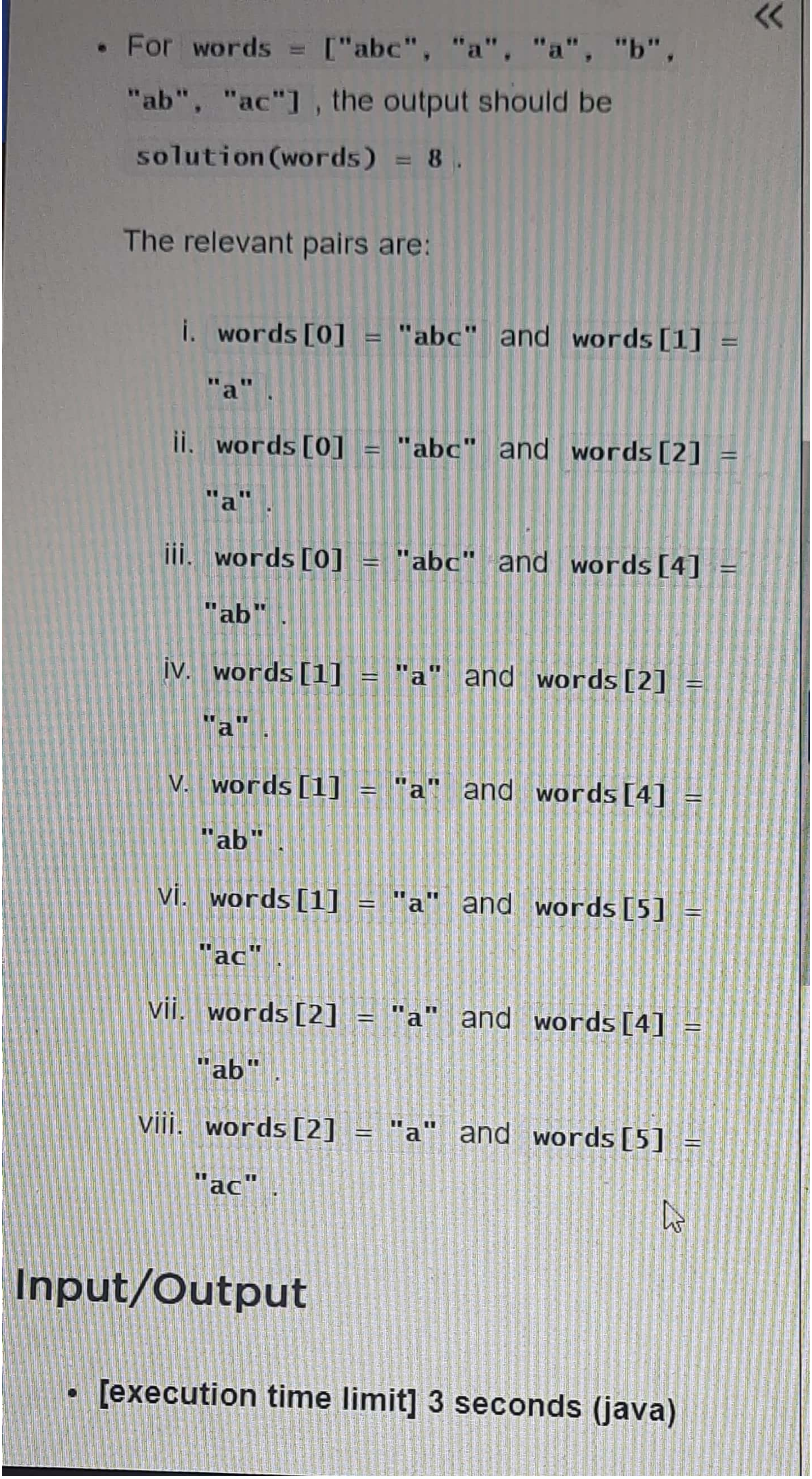
  The relevant pairs are:

  i. `words[0] = "back"` and `words[1] = "backdoor"`.

  ii. `words[0] = "back"` and `words[3] = "backgammon"`.

  iii. `words[4] = "comeback"` and `words[5] = "come"`.

- For `words = ["abc", "a", "a", "b", "ab", "ac"]`, the output should be `solution(words) = 8`.



- For `words = ["abc", "a", "a", "b", "ab", "ac"]` , the output should be `solution(words) = 8` .

The relevant pairs are:

i. `words[0] = "abc"` and `words[1] = "a"` .

ii. `words[0] = "abc"` and `words[2] = "a"` .

iii. `words[0] = "abc"` and `words[4] = "ab"` .

iv. `words[1] = "a"` and `words[2] = "a"` .

v. `words[1] = "a"` and `words[4] = "ab"` .

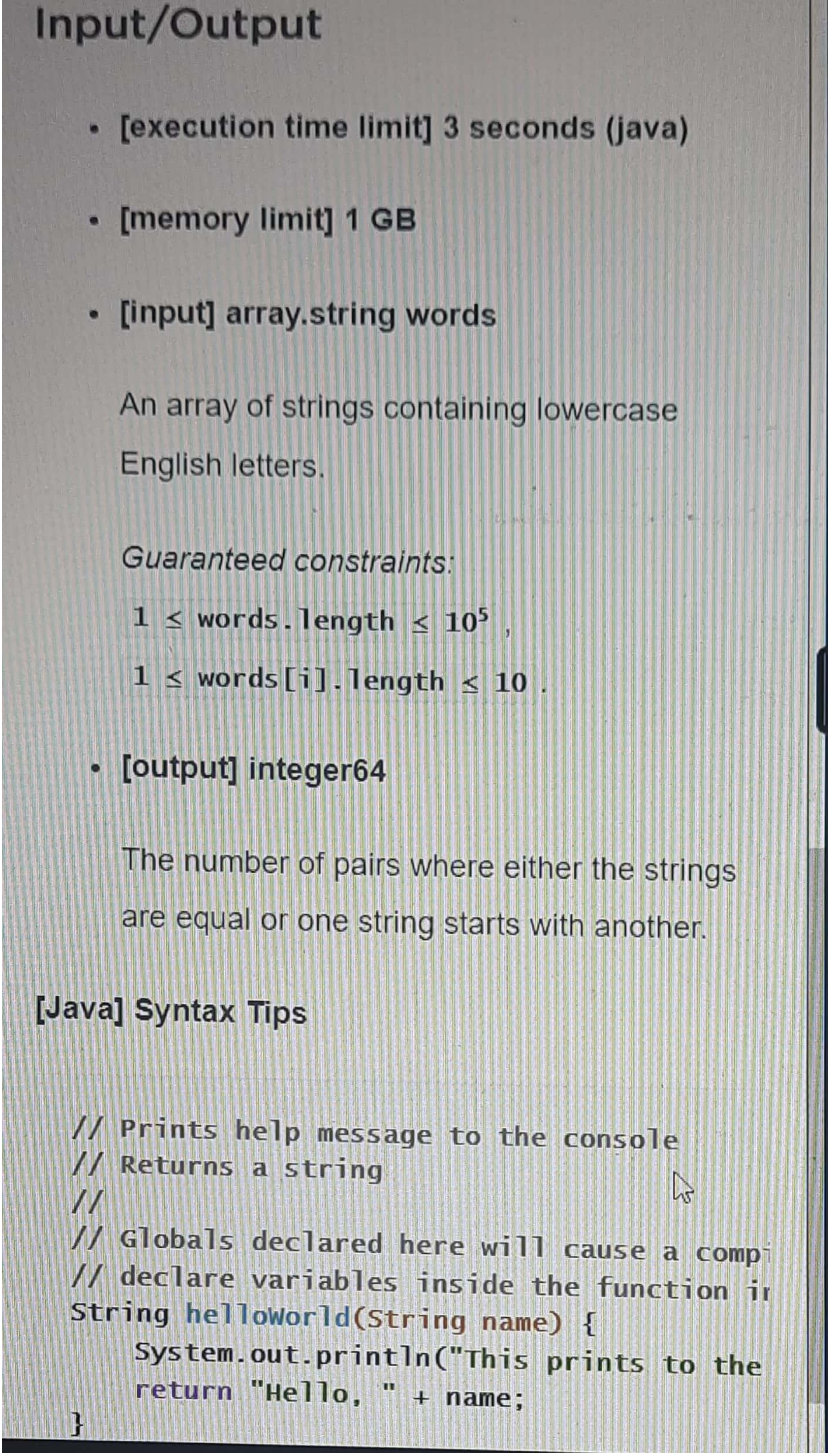
vi. `words[1] = "a"` and `words[5] = "ac"` .

vii. `words[2] = "a"` and `words[4] = "ab"` .

viii. `words[2] = "a"` and `words[5] = "ac"` .

Input/Output

- **[execution time limit] 3 seconds (java)**



Input/Output

- **[execution time limit] 3 seconds (java)**

- **[memory limit] 1 GB**

- **[input] array.string words**

  An array of strings containing lowercase English letters.

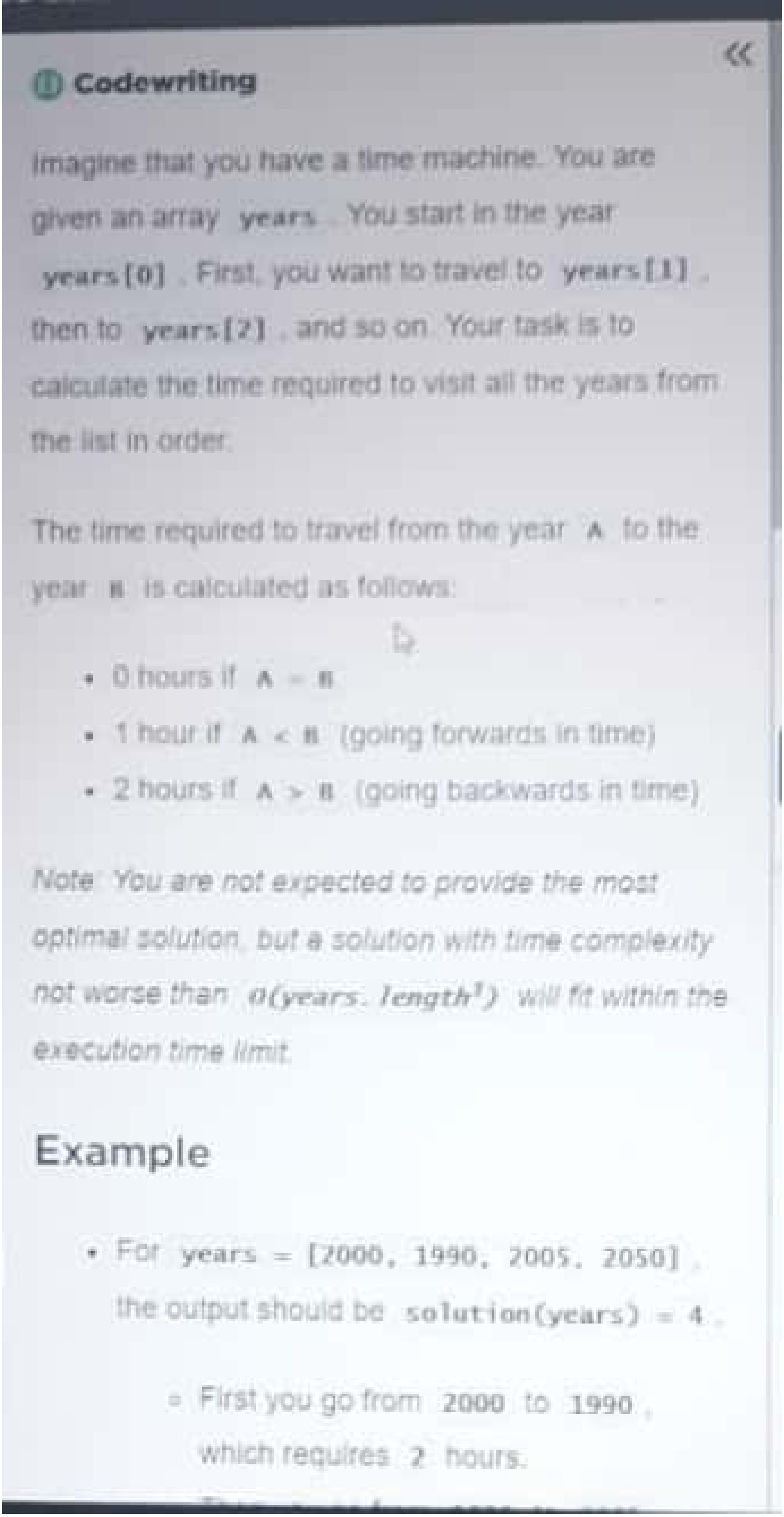
  Guaranteed constraints:
  `1 ≤ words.length ≤ 10`^{5} ,
  `1 ≤ words[i].length ≤ 10` .

- **[output] integer64**

  The number of pairs where either the strings are equal or one string starts with another.

[Java] Syntax Tips

```
// Prints help message to the console
// Returns a string
//
// Globals declared here will cause a compi
// declare variables inside the function i
String helloWorld(String name) {
    System.out.println("This prints to the
    return "Hello, " + name;
}
```



Codewriting

Imagine that you have a time machine. You are given an array `years`. You start in the year `years[0]`. First, you want to travel to `years[1]`, then to `years[2]`, and so on. Your task is to calculate the time required to visit all the years from the list in order.

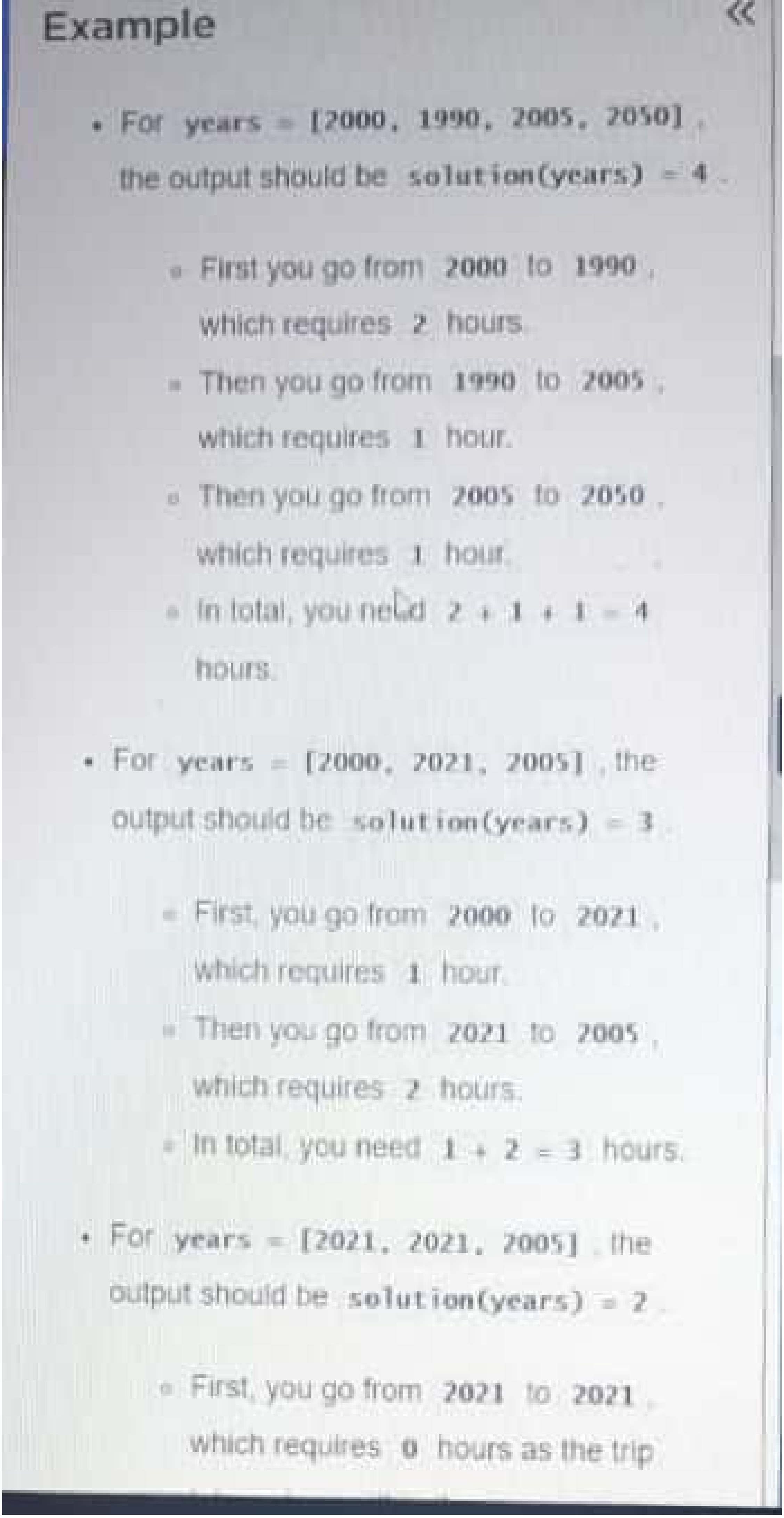
The time required to travel from the year `A` to the year `B` is calculated as follows:

- 0 hours if `A == B`
- 1 hour if `A < B` (going forwards in time)
- 2 hours if `A > B` (going backwards in time)

Note: You are not expected to provide the most optimal solution, but a solution with time complexity not worse than $O(years.length^2)$ will fit within the execution time limit.

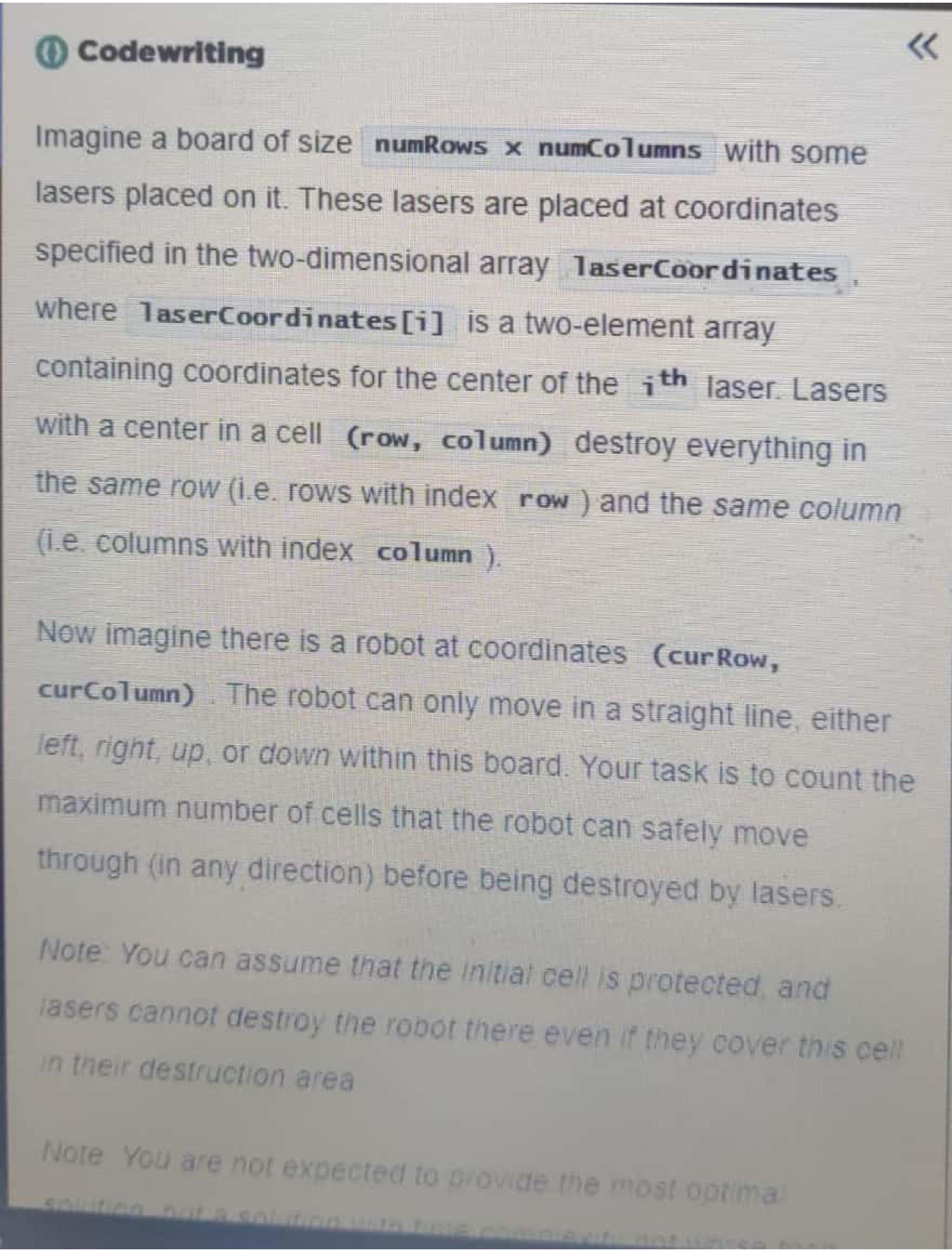
Example

- For `years = [2000, 1990, 2005, 2050]`, the output should be `solution(years) = 4`.
  - First you go from `2000` to `1990`, which requires `2` hours.



Example

- For `years = [2000, 1990, 2005, 2050]`, the output should be `solution(years) = 4`.
  - First you go from `2000` to `1990`, which requires `2` hours.
  - Then you go from `1990` to `2005`, which requires `1` hour.
  - Then you go from `2005` to `2050`, which requires `1` hour.
  - In total, you need `2 + 1 + 1 = 4` hours.
- For `years = [2000, 2021, 2005]`, the output should be `solution(years) = 3`.
  - First, you go from `2000` to `2021`, which requires `1` hour.
  - Then you go from `2021` to `2005`, which requires `2` hours.
  - In total, you need `1 + 2 = 3` hours.
- For `years = [2021, 2021, 2005]`, the output should be `solution(years) = 2`.
  - First, you go from `2021` to `2021`, which requires `0` hours as the trip



Codewriting

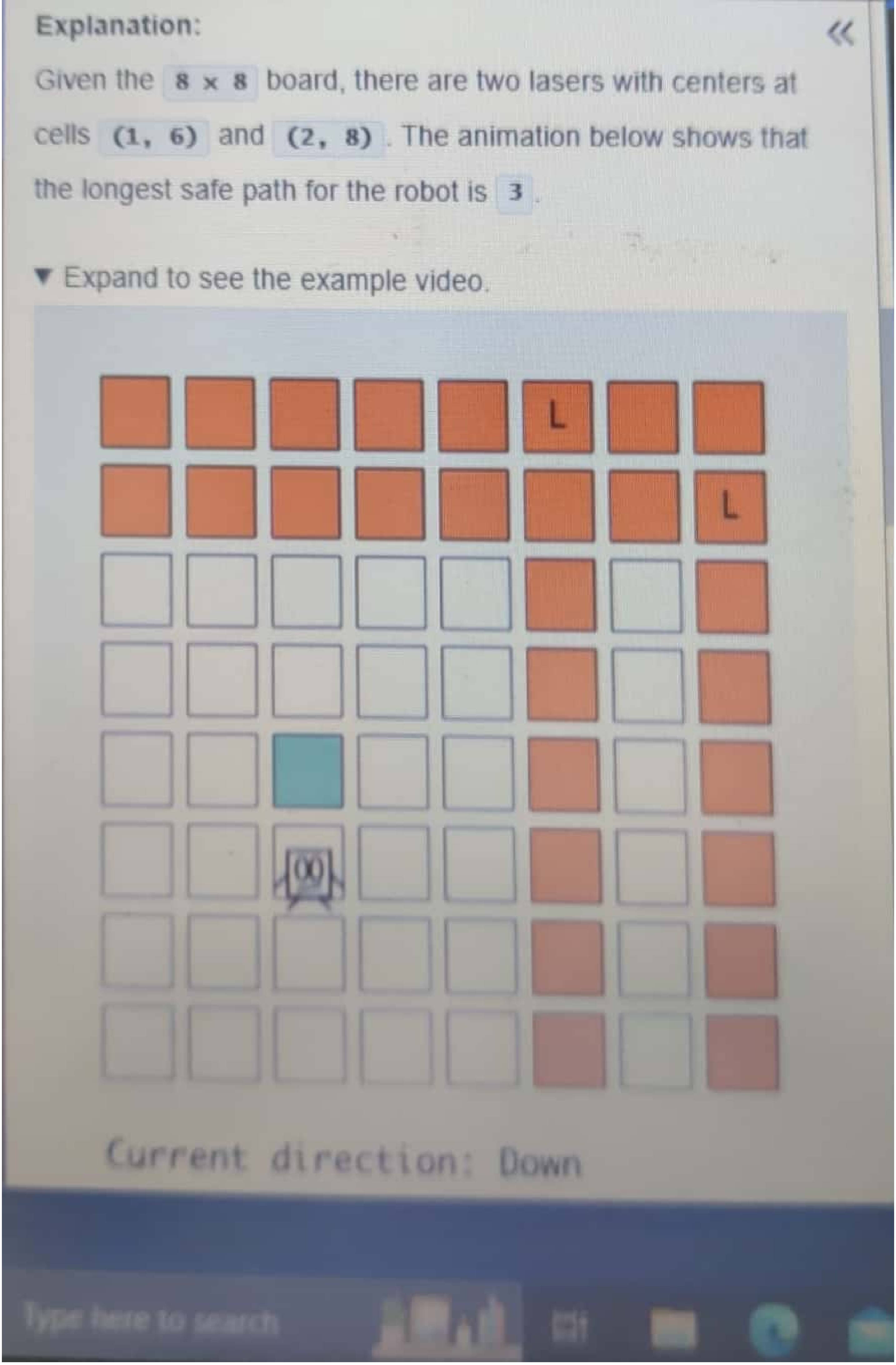
Imagine a board of size `numRows x numColumns` with some lasers placed on it. These lasers are placed at coordinates specified in the two-dimensional array `laserCoordinates`, where `laserCoordinates[i]` is a two-element array containing coordinates for the center of the i^{th} laser. Lasers with a center in a cell `(row, column)` destroy everything in the *same row* (i.e. rows with index `row`) and the *same column* (i.e. columns with index `column`).

Now imagine there is a robot at coordinates `(curRow, curColumn)`. The robot can only move in a straight line, either *left*, *right*, *up*, or *down* within this board. Your task is to count the maximum number of cells that the robot can safely move through (in any direction) before being destroyed by lasers.

Note: You can assume that the initial cell is protected, and lasers cannot destroy the robot there even if they cover this cell in their destruction area.

Note: You are not expected to provide the most optimal solution, but a solution with time complexity not worse than

Scanned with CamScanner

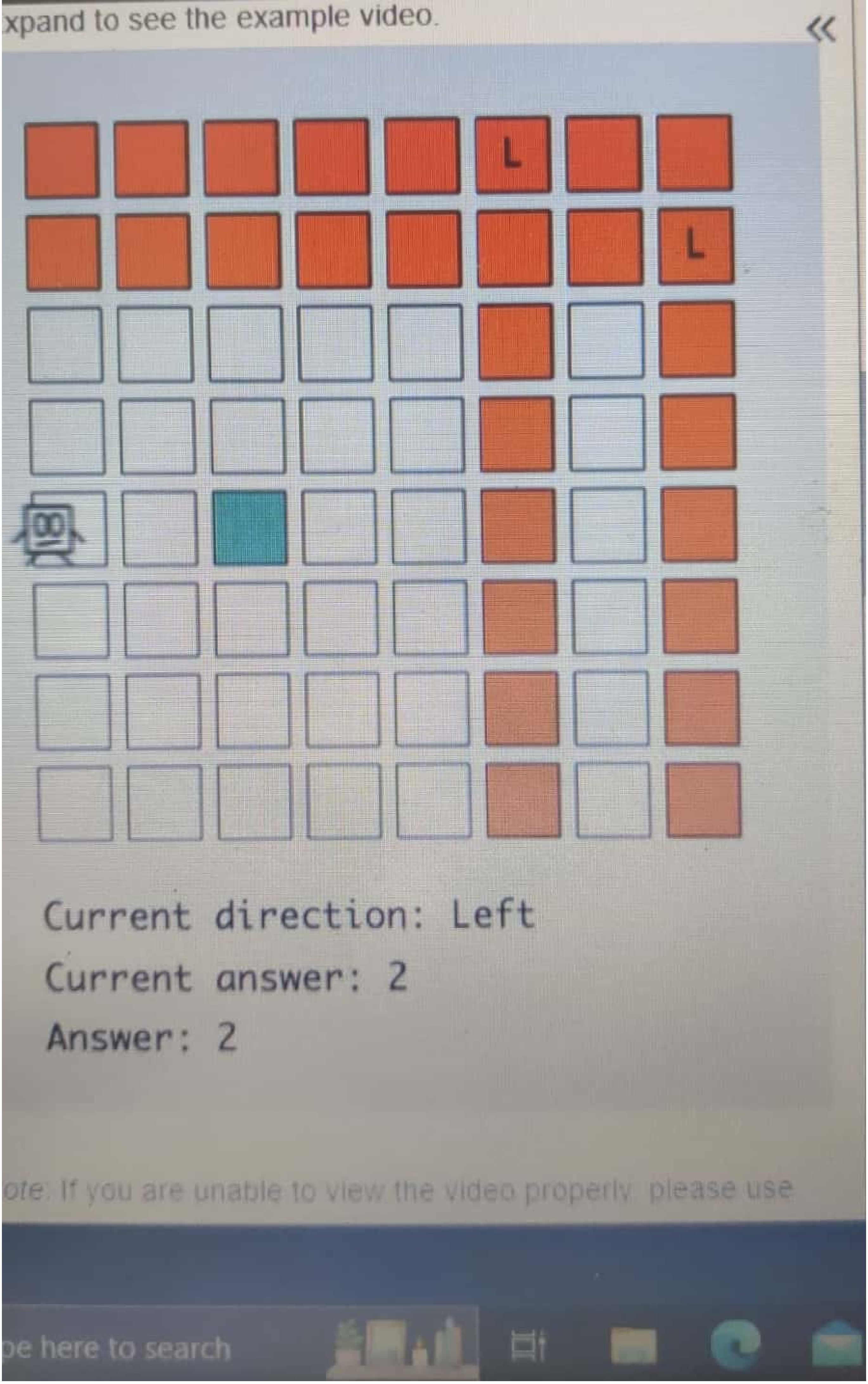


Explanation:

Given the `8 x 8` board, there are two lasers with centers at cells `(1, 6)` and `(2, 8)`. The animation below shows that the longest safe path for the robot is `3`.

▼ Expand to see the example video.

Current direction: Down



xpand to see the example video.

```
Current direction: Left
Current answer: 2
Answer: 2
```

ote: If you are unable to view the video properly, please use

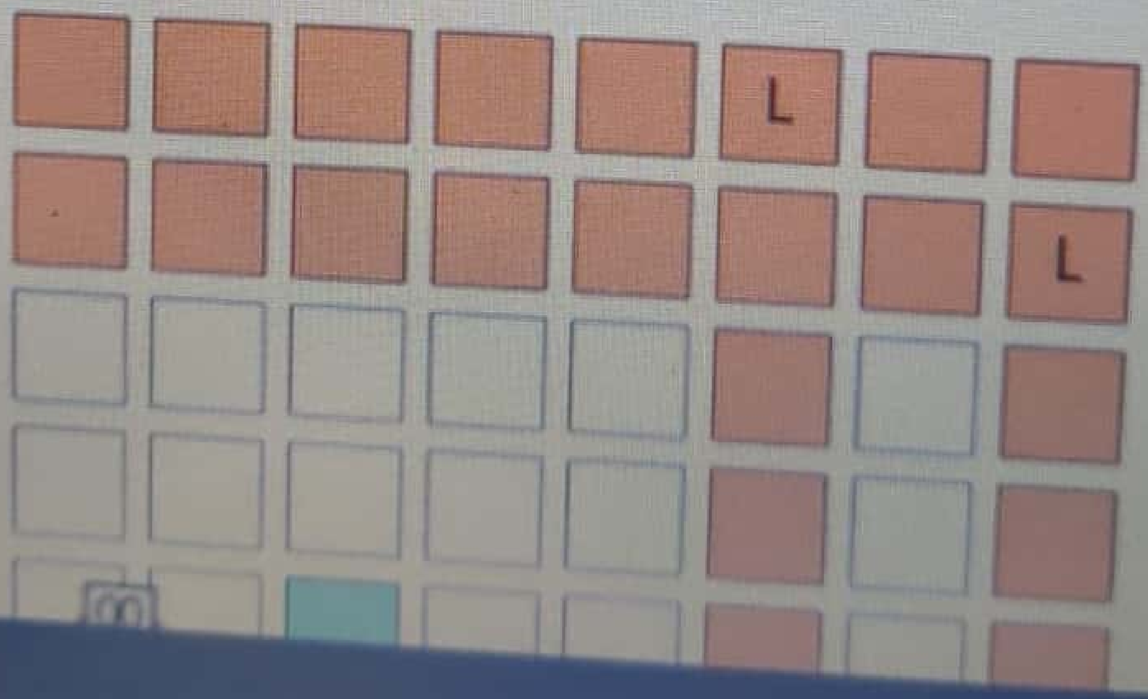
Example

For `numRows = 8`, `numColumns = 8`, `curRow = 5`, `curColumn = 3`, and `laserCoordinates = [[1, 6], [2, 8]]`, the output should be `solution(numRows, numColumns, curRow, curColumn, laserCoordinates) = 3`.

Explanation:

Given the `8 x 8` board, there are two lasers with centers at cells `(1, 6)` and `(2, 8)`. The animation below shows that the longest safe path for the robot is `3`.

▼ Expand to see the example video.



Type here to search