数据结构
Data Structures

Chapter 3 Array and Matrix

Prof. Yitian Shao
School of Computer Science and Technology

Array and Matrix

Course Overview

- Vector
  - Differences between array and vector
  - Variable definition and assignment
  - Accessing elements of a vector
  - Modify a vector
- Matrix
  - Variable definition and assignment
  - Accessing elements of a matrix
  - Modify a matrix

The course content is developed partially based on Stanford CS106B. Copyright (C) Stanford Computer Science and Tyler Conklin, licensed under Creative Commons Attribution 2.5 License.

Vector: Array with dynamic size

- **Vector** is a collection of elements, like a dynamically-resizing array
- C++ arrays cannot be easily resized.
- C++ lets you index out of the array bounds (garbage memory) without necessarily crashing or warning.
- Array does not support: inserting/deleting elements into the front/middle/back of the array, reversing, sorting the elements, searching for a given value

Vector: Array with dynamic size

- Vector is part of the C++ STL (`#include<vector>  using namespace std`)
- Include the **data type of elements** in the `<>` brackets

Vector members

Element access

at(i)	access element i with bounds checking
operator[]	access specified element
front()	access the first element
back()	access the last element

Modifiers

clear()	clears the contents
insert(i, value)	inserts elements at position i
erase(i)	erases elements at position i
push_back(value)	adds an element to the end
pop_back()	removes the last element

Capacity

empty()	checks whether the container is empty
size()	returns the number of elements

Iterators

begin()	returns an iterator to the beginning
end()	returns an iterator to the end

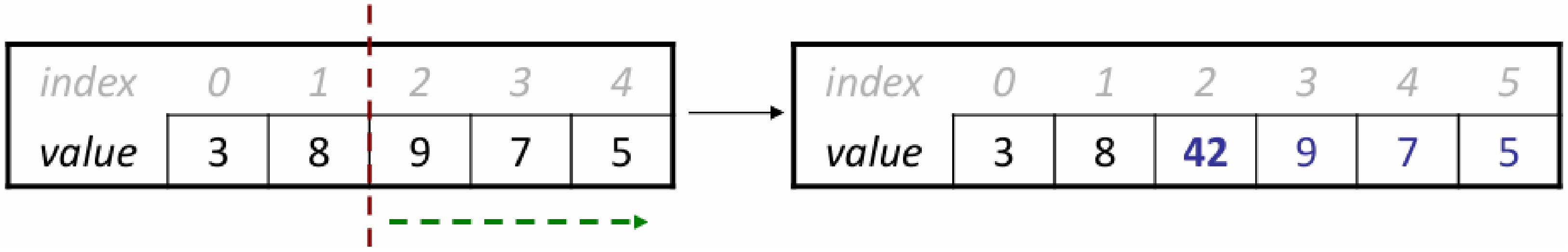
Iterating over a vector

```
vector<string> vec {“A", “B", “C"};
for (int i = 0; i < vec.size(); i++) {          // Prints off each element on its own line
   cout << vec[i] << endl;
}
for (int i = vec.size() - 1; i >= 0; i--) {          // Same thing as above but backwards
   cout << vec[i] << endl;
}
for (string v : vec) {          // "for-each" loop
   cout << v << endl;
}
```

Vector insert/remove

```
vector<int> vec {3, 8, 9, 7, 5};
```

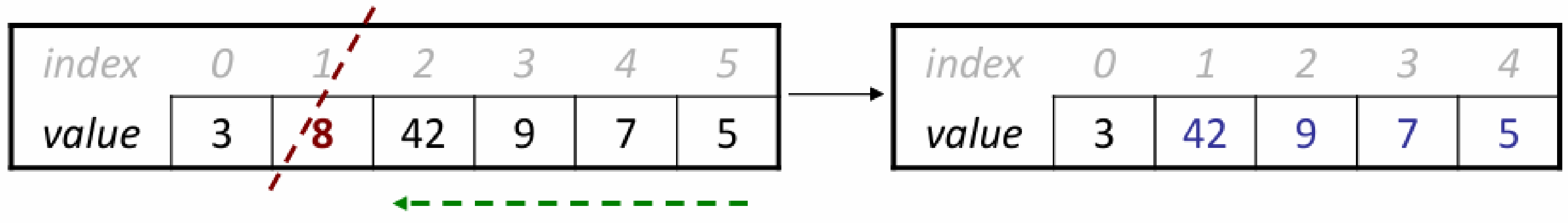
- vec.insert(2, 42); //shift elements right to make room for the new element



index	0	1	2	3	4
value	3	8	9	7	5

index	0	1	2	3	4	5
value	3	8	42	9	7	5

- vec.erase(1); //shift elements left to cover the space left by the removed element



index	0	1	2	3	4	5
value	3	8	42	9	7	5

index	0	1	2	3	4
value	3	42	9	7	5

(The more elements to shift, the slower these operations will be)

Vector exercise A

- Write a function countInRange that accepts a vector<int>, a min, and a max. It returns the number of values in the vector that fall within the range inclusive.
- So if a vector vec contained {0, 5, -21, -4, 7} and min = 2 and max = 12, the function would return 2.

```
class Solution {
public:
    int countInRange (vector<int>& vec, int min, int max) {

        // Implement your solution

    }
};
```

How to test your solution

```
class Solution {
public:
    int countInRange (vector<int>& vec, int min, int max) {

        // Implement your solution

    }
};
```

```
int main()
{
   Solution sol;

   vector<int>  test1 {0, 5, -21, -4, 7};
   cout << "Answer of Test1 is " << sol.countInRange(test1, 2, 12) << endl;

   vector<int>  test2 {0, 2, 4, 15, 3};
   cout << "Answer of Test2 is " << sol.countInRange(test2, 2, 12) << endl;
}
```

Vector exercise A: solution

```cpp
class Solution {
public:
    int countInRange (vector<int>& vec, int min, int max) {

        int count = 0;

        for(int v : vec)
        {
            if(v >= min && v <= max)
                count++;
        }

        return count;

    }
};
```

Vector exercise B

- Write a function `removeAll` that accepts a vector of strings, and a target string. It removes any strings in the vector that equal the target string.
- So if vec contained {“Youre”, “a”, “hairy”,“wizard”, “hairy”} and target = “hairy”, vec should equal {“Youre”, “a”, “wizard”} .

```
class Solution {
public:
    void removeAll (vector<string>& vec, string target) {

        // Implement your solution

    }
};
```

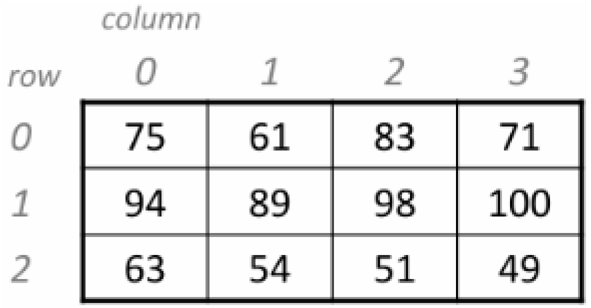
Vector exercise B: solution

```
class Solution {
public:
    void removeAll (vector<string>& vec, string target) {

       for (int i = vec.length()- 1; i >= 0; i--) {
               if (vec[i] == target) {
                      vec.erase(i);
               }
       }

    }
};
```

Matrix

- A matrix is two-dimensional array
- Implement a matrix of integers using C++ array (fixed size)

```cpp
int matrix[3][4] = {
     {75, 61, 83, 71},
     {94, 89, 98, 100},
     {63, 54, 51, 49}
 };
```



	column			
row	0	1	2	3
0	75	61	83	71
1	94	89	98	100
2	63	54	51	49

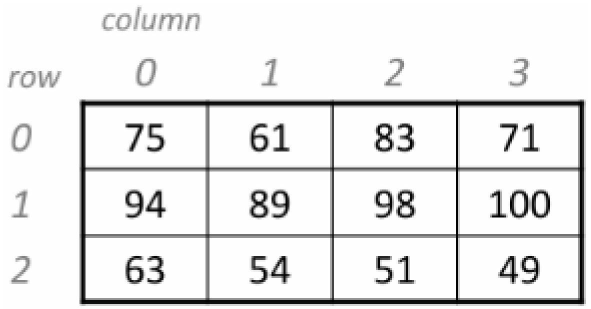
- Accessing element at row 1, column 2

```
matrix[1][2]
```

Matrix (dynamic size)

- Implement a matrix of integers using C++ vector

```
vector<vector<int>> matrix = {
      {75, 61, 83, 71},
      {94, 89, 98, 100},
      {63, 54, 51, 49}
 };
```

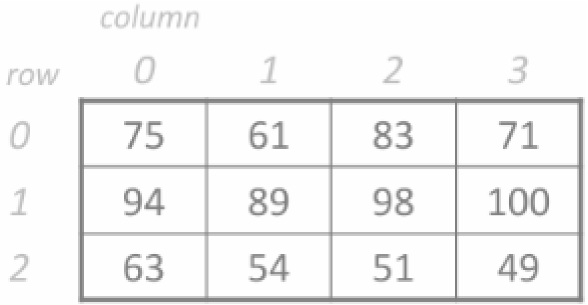


	column			
row	0	1	2	3
0	75	61	83	71
1	94	89	98	100
2	63	54	51	49

Matrix (dynamic size)

- Implement a matrix of integers using C++ vector

```
vector<vector<int>> matrix = {
      {75, 61, 83, 71},
      {94, 89, 98, 100},
      {63, 54, 51, 49}
 };
```



	column			
row	0	1	2	3
0	75	61	83	71
1	94	89	98	100
2	63	54	51	49

- Print out all elements of this matrix by **looping** through rows and columns

```
for (int i = 0; i < matrix.size(); i++) {
    for (int j = 0; j < matrix[i].size(); j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}
```

Matrix: Insert Elements

```
int main() {
    vector<vector<int>> v = {{1, 2, 3},
                             {4, 5, 6}};

    // Insert a new row at the end
    v.push_back({7, 8, 9});

    // Insert value in 2nd row at 2nd position
    v[1].insert(v[1].begin() + 1, 10);

    for (int i = 0; i < v.size(); i++) {
        for (int j = 0; j < v[i].size(); j++) {
            cout << v[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Matrix: Delete Elements

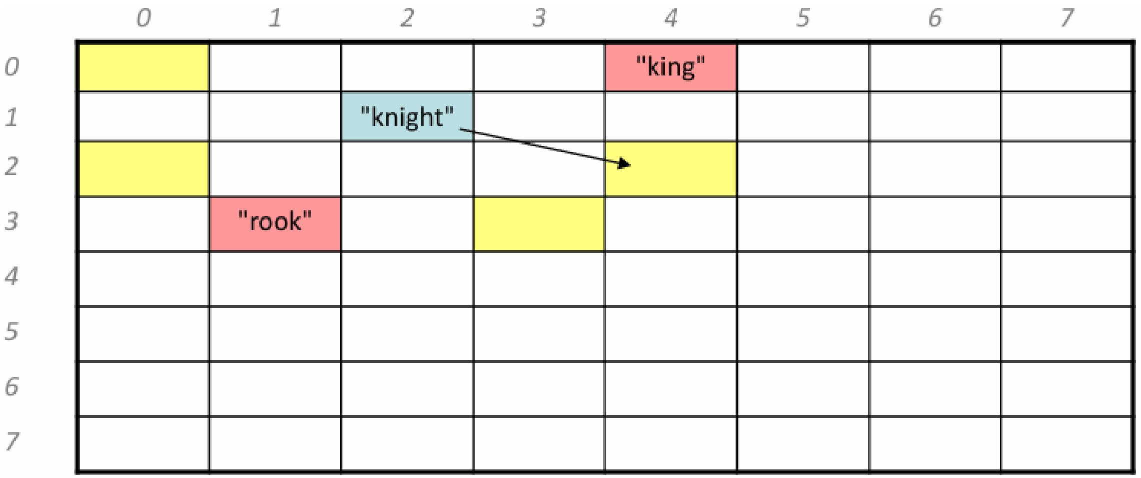
```
int main() {
    vector<vector<int>> v = {{1, 2, 3},
                             {4, 5, 6}};

    // Delete the second row
    v.erase(v.begin() + 1);

    // Delete second element in first row
    v[0].erase(v[0].begin() + 1);

    for (int i = 0; i < v.size(); i++) {
        for (int j = 0; j < v[i].size(); j++) {
            cout << v[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Matrix exercise A

- Write a function knightCanMove that accepts a matrix and two row/column pairs (r1, c1), (r2, c2) as parameters, and returns true if there is a knight at chess board square (r1, c1) that can legally move to empty square (r2, c2).

- Recall that a knight makes an "L" shaped move, going 2 squares in one dimension and 1 square in the other.

- knightCanMove(board, 1, 2, 2, 4) returns true



	0	1	2	3	4	5	6	7
0	🟨				"king"			
1			"knight"					
2	🟨				🟨			
3		"rook"		🟨				
4								
5								
6								
7								

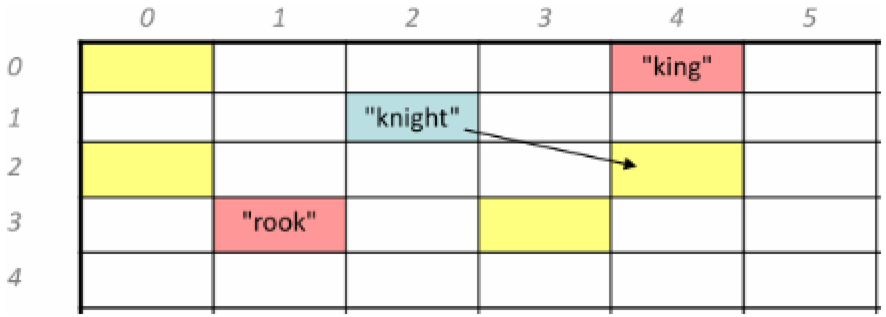
```cpp
class Solution {
public:
    int knightCanMove (vector<vector<int>>& board, int r1, int c1, int r2, int c2) {
        // Implement your solution
    }
};
```

Matrix exercise A: problem break down

- Write a function `knightCanMove` that accepts a matrix and two row/column pairs (r1, c1), (r2, c2) as parameters, and returns true if there is a knight at chess board square (r1, c1) that can legally move to empty square (r2, c2).

1. Are the given chess board row/column pairs valid?
2. Is there a knight at chess board square (r1, c1)
3. Is the square (r2, c2) empty?
4. Is the movement from (r1, c1) to (r2, c2) valid?

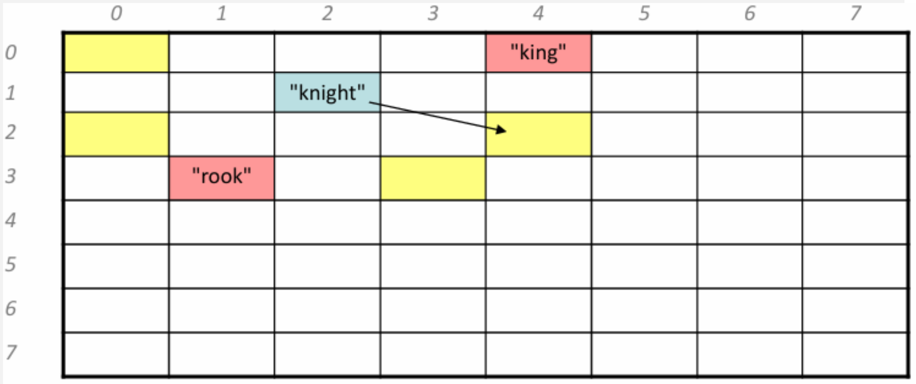
   (going 2 squares in one dimension and 1 square in the other)



	0	1	2	3	4	5
0	🟨				"king"	
1			"knight"			
2	🟨				🟨	
3		"rook"		🟨		
4						

Matrix exercise A: assistive function

```
class Solution {
public:
    bool inBounds (vector<vector<int>>& board, int r, int c) {
        if(r < 0 || r >= board.size()) {
              return false;
       }
        if(c < 0 || c= board[r].size()) {
              return false;
       }
       return true;
    }



    int knightCanMove (vector<vector<int>>& board, int r1, int c1, int r2, int c2) {
       // Implement your solution
    }
};
```

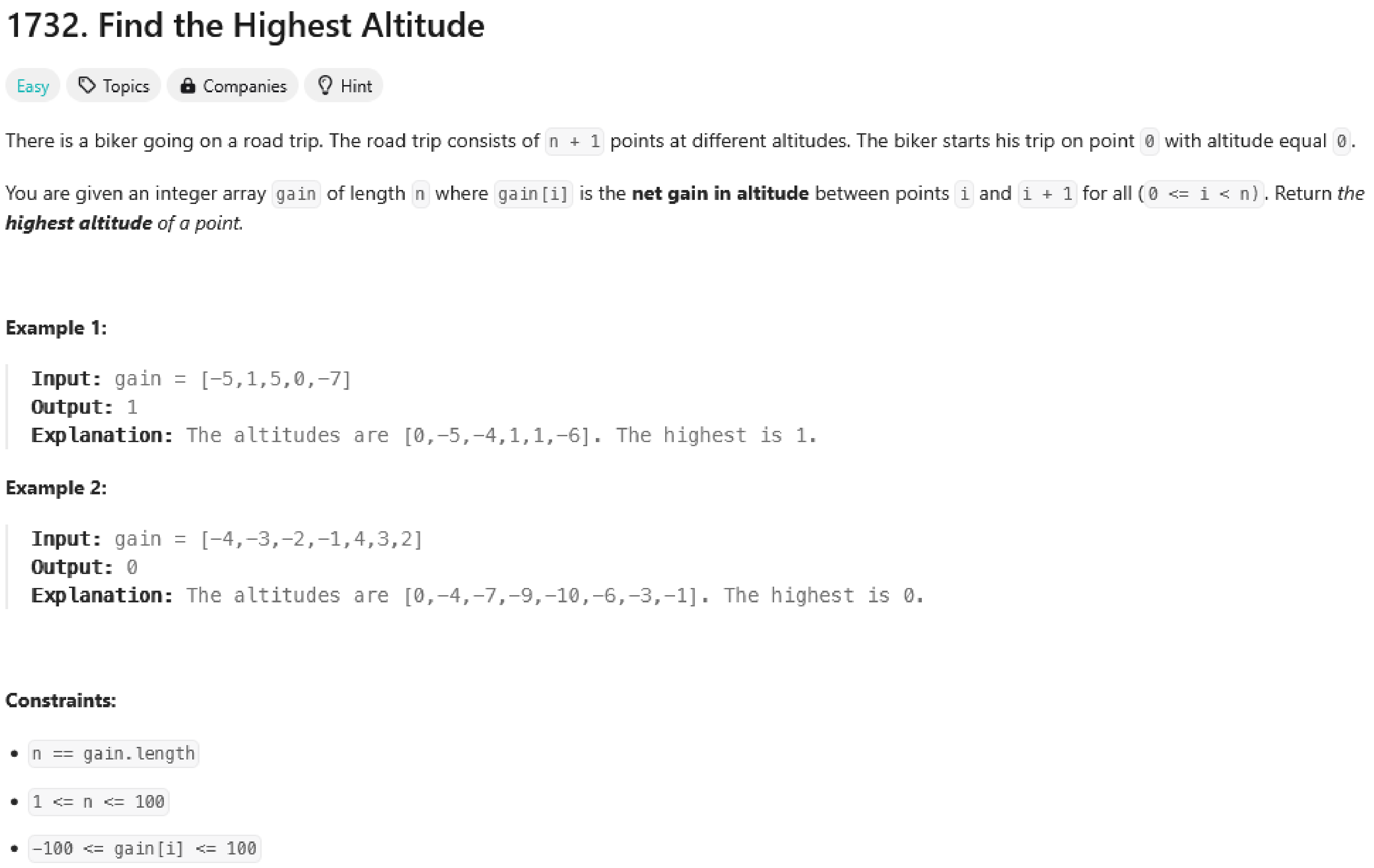


Matrix exercise A: solution

```
class Solution {
public:
    int knightCanMove (vector<vector<int>>& board, int r1, int c1, int r2, int c2) {
       if (!inBounds(board, r1, c1) || !inBounds(board, r2, c2)) {
              return false;
        }

        if (board[r1][c1] != "knight" || board[r2][c2] != "") {
              return false;
        }

        int dr = abs(r1 - r2), dc = abs(c1 - c2);
        if (!((dr == 1 && dc == 2) || (dr == 2 && dc == 1))) {
              return false;
        }
        return true;
    }
};
```

Exercise 3.1

- Complete LeetCode 1732



1732. Find the Highest Altitude

There is a biker going on a road trip. The road trip consists of `n + 1` points at different altitudes. The biker starts his trip on point `0` with altitude equal `0`.

You are given an integer array `gain` of length `n` where `gain[i]` is the **net gain in altitude** between points `i` and `i + 1` for all (`0 <= i < n)`. Return *the* ***highest altitude*** *of a point.*

Example 1:

```
Input: gain = [-5,1,5,0,-7]
Output: 1
Explanation: The altitudes are [0,-5,-4,1,1,-6]. The highest is 1.
```

Example 2:

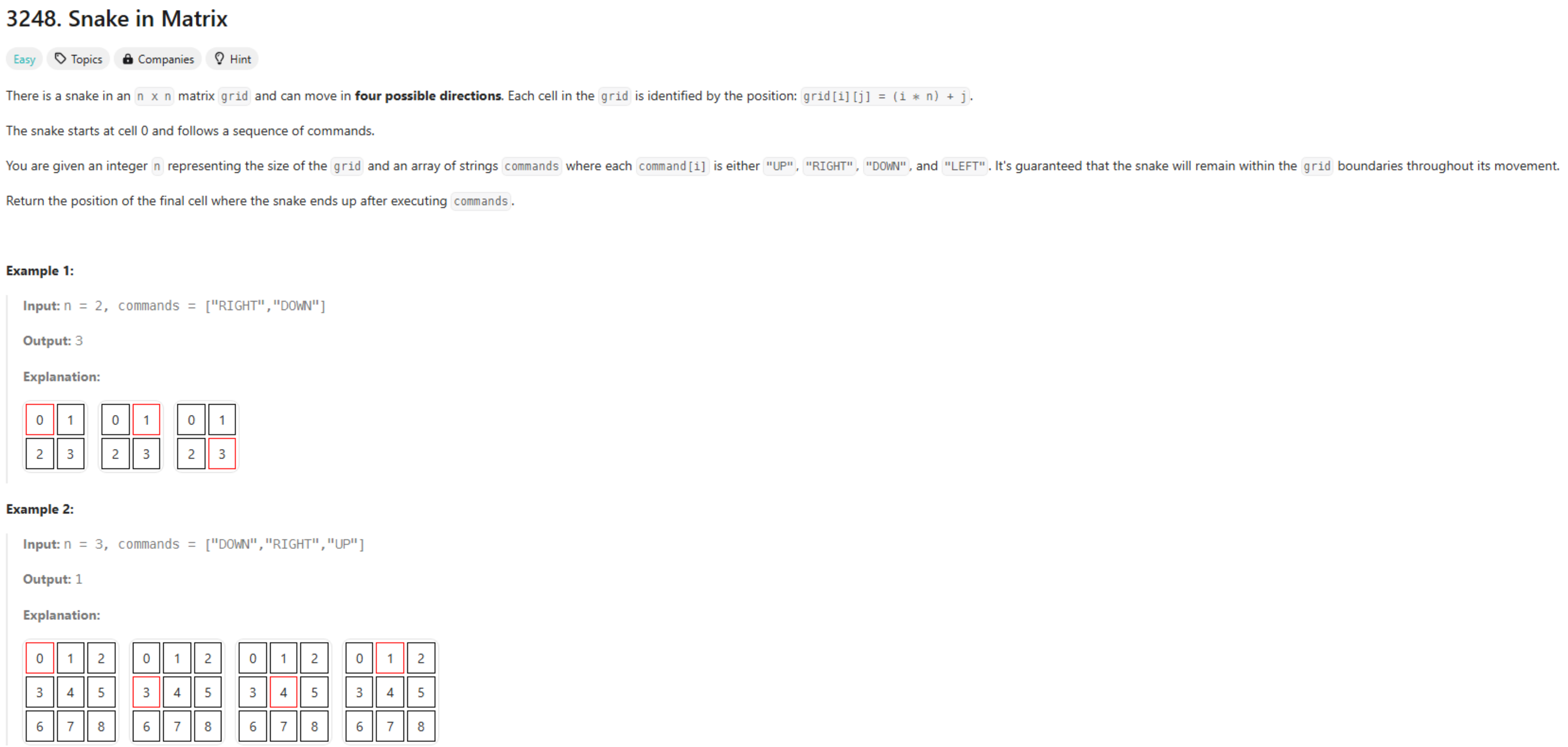
```
Input: gain = [-4,-3,-2,-1,4,3,2]
Output: 0
Explanation: The altitudes are [0,-4,-7,-9,-10,-6,-3,-1]. The highest is 0.
```

Constraints:

- `n == gain.length`
- `1 <= n <= 100`
- `-100 <= gain[i] <= 100`

Exercise 3.2

- Complete LeetCode 3248



3248. Snake in Matrix

There is a snake in an `n x n` matrix `grid` and can move in **four possible directions**. Each cell in the `grid` is identified by the position: `grid[i][j] = (i * n) + j`.

The snake starts at cell 0 and follows a sequence of commands.

You are given an integer `n` representing the size of the `grid` and an array of strings `commands` where each `command[i]` is either `"UP"`, `"RIGHT"`, `"DOWN"`, and `"LEFT"`. It's guaranteed that the snake will remain within the `grid` boundaries throughout its movement.

Return the position of the final cell where the snake ends up after executing `commands`.

Example 1:

> **Input:** `n = 2, commands = ["RIGHT","DOWN"]`
>
> **Output:** 3
>
> **Explanation:**

0	1
2	3

0	1
2	3

0	1
2	3

Example 2:

> **Input:** `n = 3, commands = ["DOWN","RIGHT","UP"]`
>
> **Output:** 1
>
> **Explanation:**

0	1	2
3	4	5
6	7	8

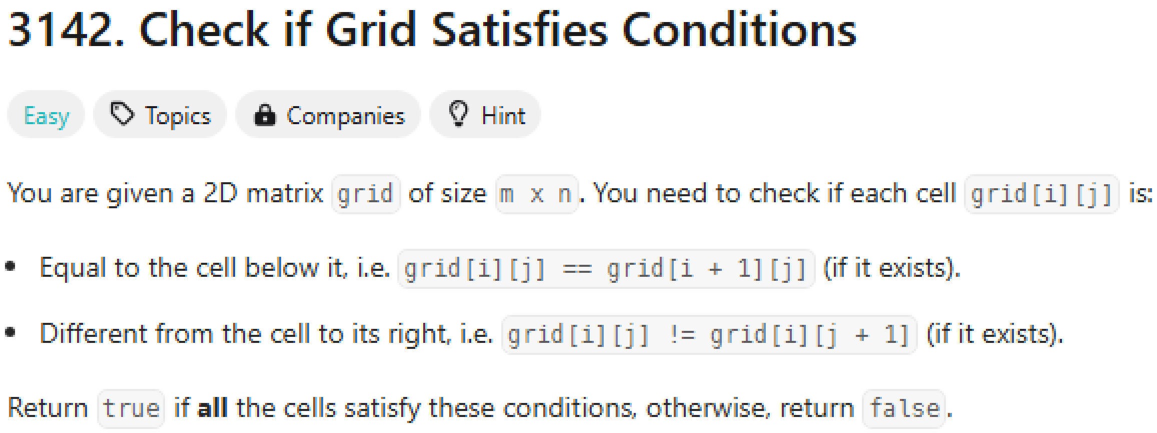
0	1	2
3	4	5
6	7	8

0	1	2
3	4	5
6	7	8

0	1	2
3	4	5
6	7	8

Exercise 3.3

- Complete LeetCode 3142



3142. Check if Grid Satisfies Conditions

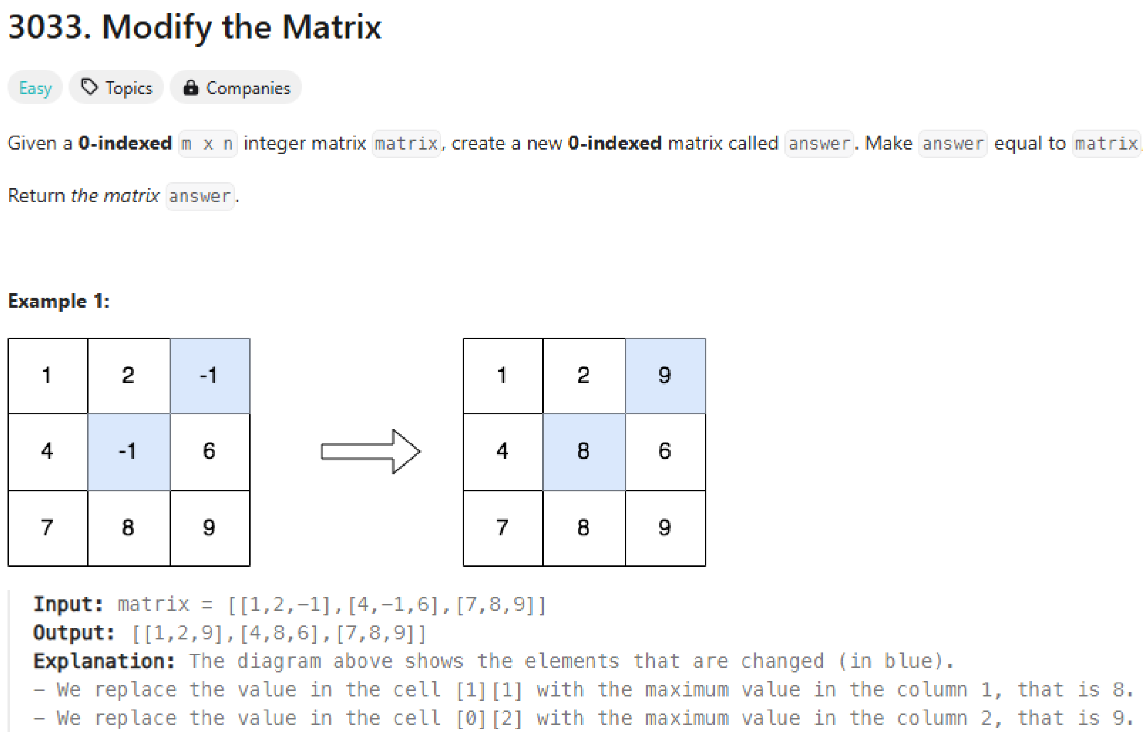
You are given a 2D matrix `grid` of size `m x n`. You need to check if each cell `grid[i][j]` is:

- Equal to the cell below it, i.e. `grid[i][j] == grid[i + 1][j]` (if it exists).
- Different from the cell to its right, i.e. `grid[i][j] != grid[i][j + 1]` (if it exists).

Return `true` if **all** the cells satisfy these conditions, otherwise, return `false`.

Exercise 3.4

- Complete LeetCode 3033



3033. Modify the Matrix

Easy · Topics · Companies

Given a **0-indexed** `m x n` integer matrix `matrix`, create a new **0-indexed** matrix called `answer`. Make `answer` equal to `matrix`, then replace each element with the value `-1` with the **maximum** element in its respective column.

Return *the matrix* `answer`.

Example 1:

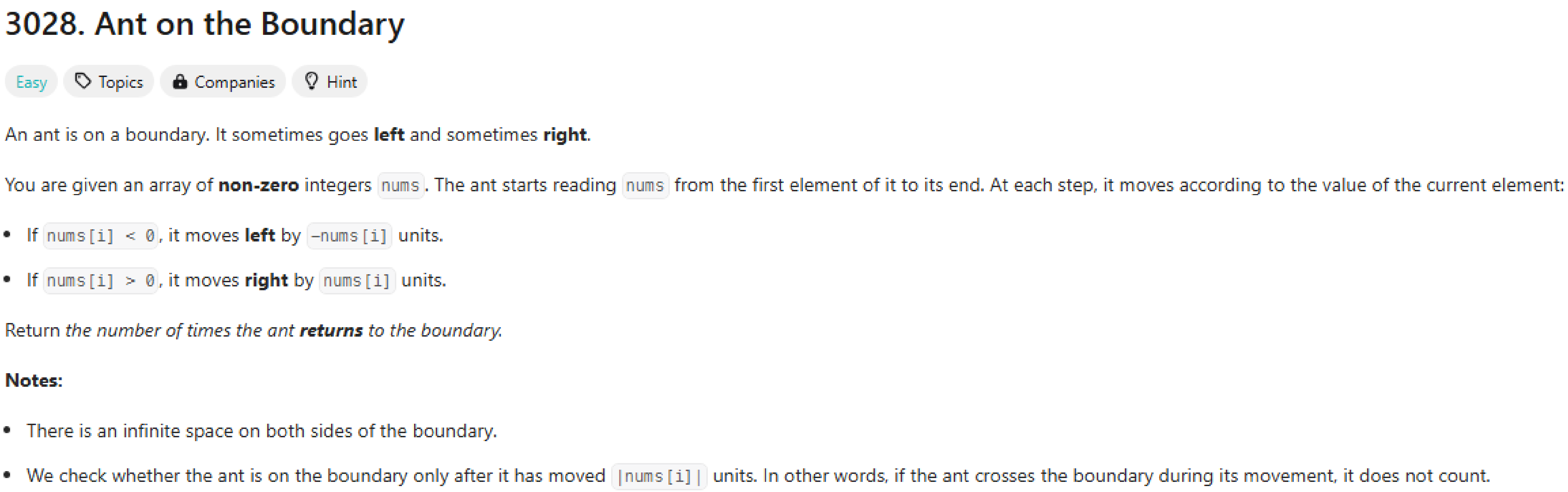
1	2	-1
4	-1	6
7	8	9

1	2	9
4	8	6
7	8	9

```
Input: matrix = [[1,2,-1],[4,-1,6],[7,8,9]]
Output: [[1,2,9],[4,8,6],[7,8,9]]
Explanation: The diagram above shows the elements that are changed (in blue).
- We replace the value in the cell [1][1] with the maximum value in the column 1, that is 8.
- We replace the value in the cell [0][2] with the maximum value in the column 2, that is 9.
```

Exercise 3.5

- Complete LeetCode 3028



3028. Ant on the Boundary

An ant is on a boundary. It sometimes goes **left** and sometimes **right**.

You are given an array of **non-zero** integers `nums`. The ant starts reading `nums` from the first element of it to its end. At each step, it moves according to the value of the current element:

- If `nums[i] < 0`, it moves **left** by `-nums[i]` units.
- If `nums[i] > 0`, it moves **right** by `nums[i]` units.

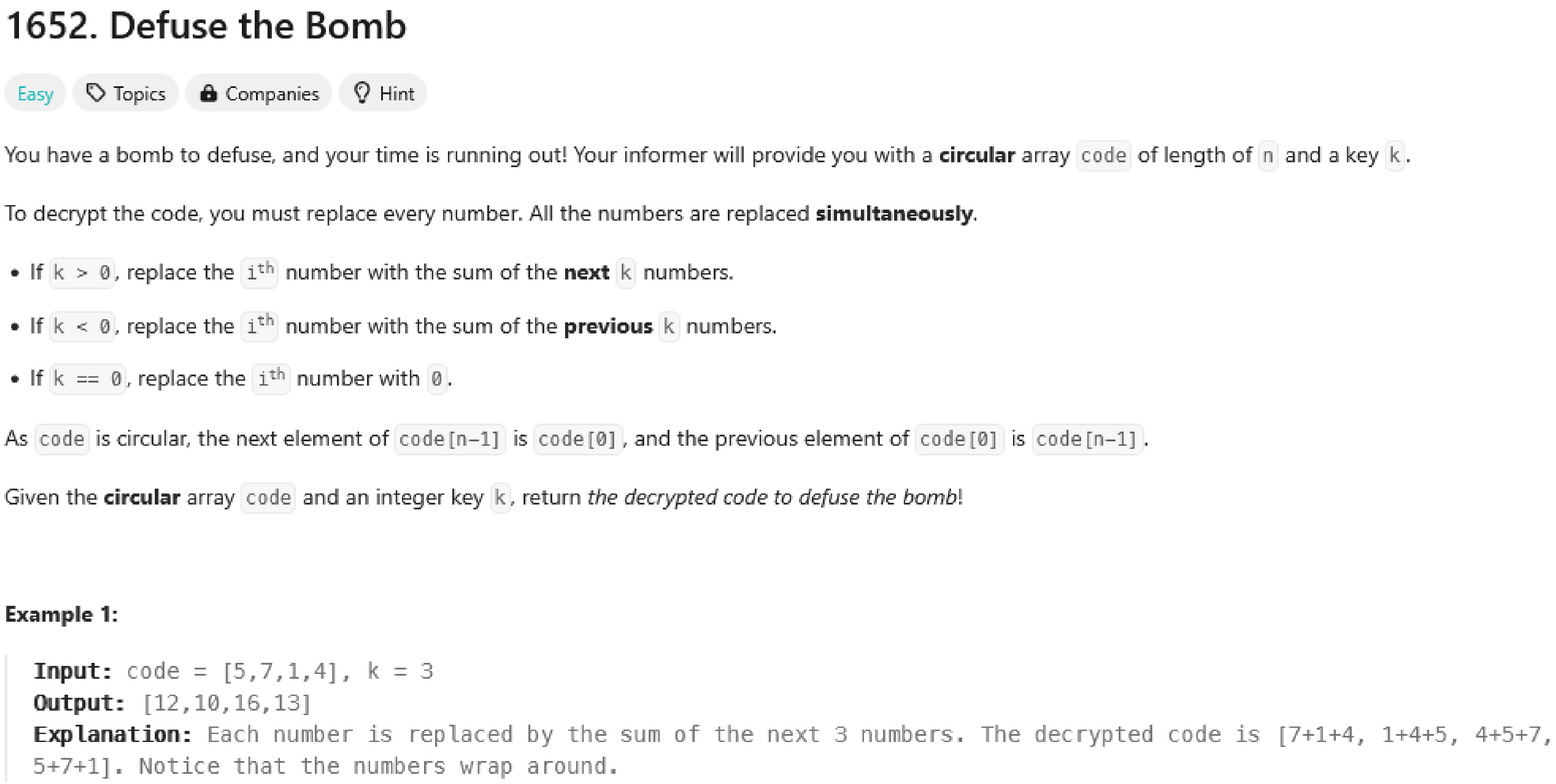
Return *the number of times the ant **returns** to the boundary.*

Notes:

- There is an infinite space on both sides of the boundary.
- We check whether the ant is on the boundary only after it has moved `|nums[i]|` units. In other words, if the ant crosses the boundary during its movement, it does not count.

Exercise 3.6

- Complete LeetCode 1652



1652. Defuse the Bomb

You have a bomb to defuse, and your time is running out! Your informer will provide you with a **circular** array `code` of length of `n` and a key `k`.

To decrypt the code, you must replace every number. All the numbers are replaced **simultaneously**.

- If `k > 0`, replace the i^{th} number with the sum of the **next** `k` numbers.
- If `k < 0`, replace the i^{th} number with the sum of the **previous** `k` numbers.
- If `k == 0`, replace the i^{th} number with `0`.

As `code` is circular, the next element of `code[n-1]` is `code[0]`, and the previous element of `code[0]` is `code[n-1]`.

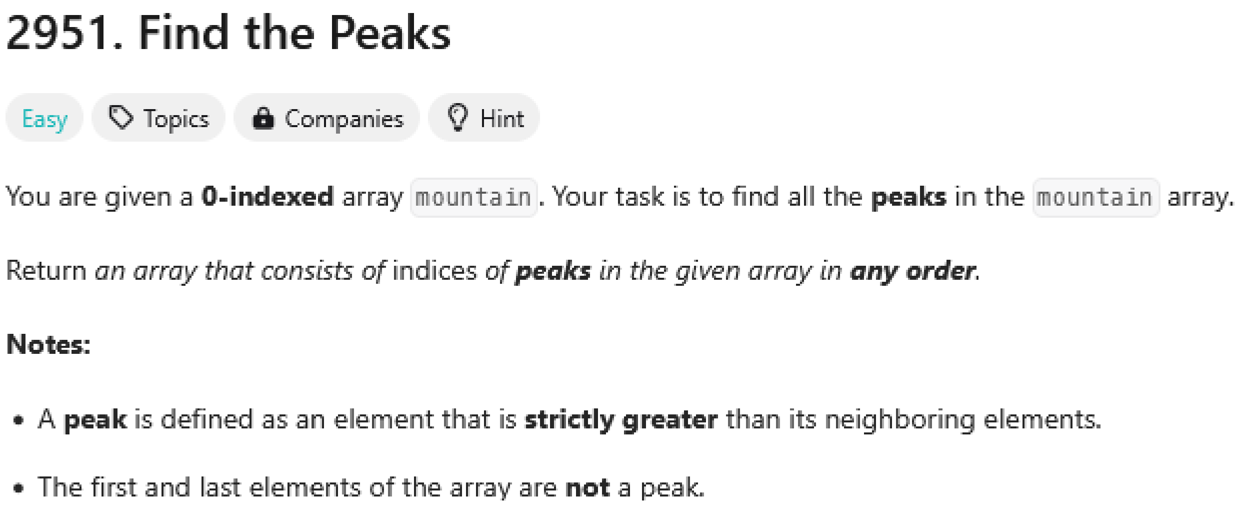
Given the **circular** array `code` and an integer key `k`, return *the decrypted code to defuse the bomb*!

Example 1:

```
Input: code = [5,7,1,4], k = 3
Output: [12,10,16,13]
Explanation: Each number is replaced by the sum of the next 3 numbers. The decrypted code is [7+1+4, 1+4+5, 4+5+7,
5+7+1]. Notice that the numbers wrap around.
```

Exercise 3.7

- Complete LeetCode 2951



2951. Find the Peaks

You are given a **0-indexed** array `mountain`. Your task is to find all the **peaks** in the `mountain` array.

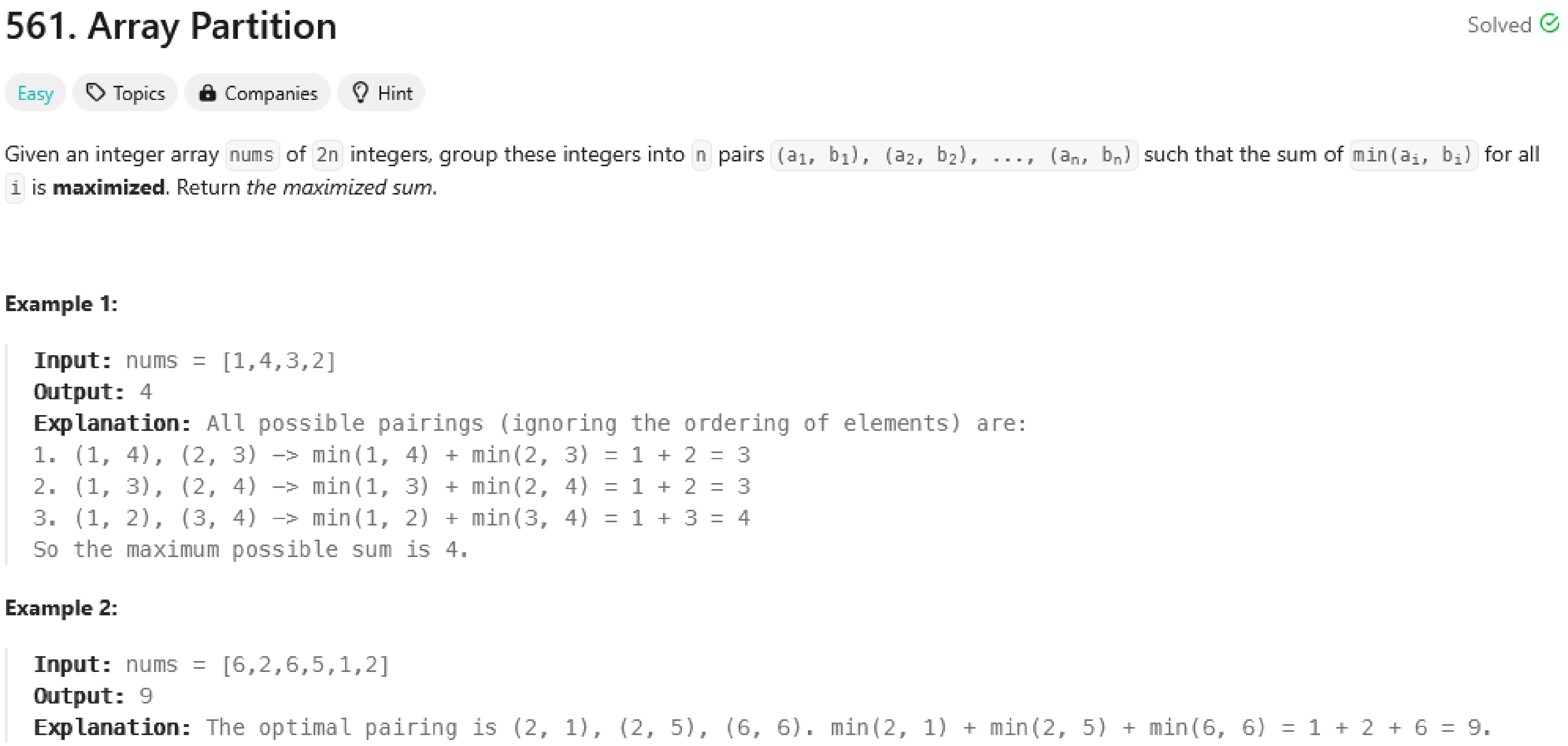
Return *an array that consists of* indices *of* ***peaks*** *in the given array in* ***any order***.

Notes:

- A **peak** is defined as an element that is **strictly greater** than its neighboring elements.
- The first and last elements of the array are **not** a peak.

Exercise 3.8

- Complete LeetCode 561



561. Array Partition

Given an integer array `nums` of `2n` integers, group these integers into `n` pairs $(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)$ such that the sum of $\min(a_i, b_i)$ for all `i` is **maximized**. Return *the maximized sum*.

Example 1:

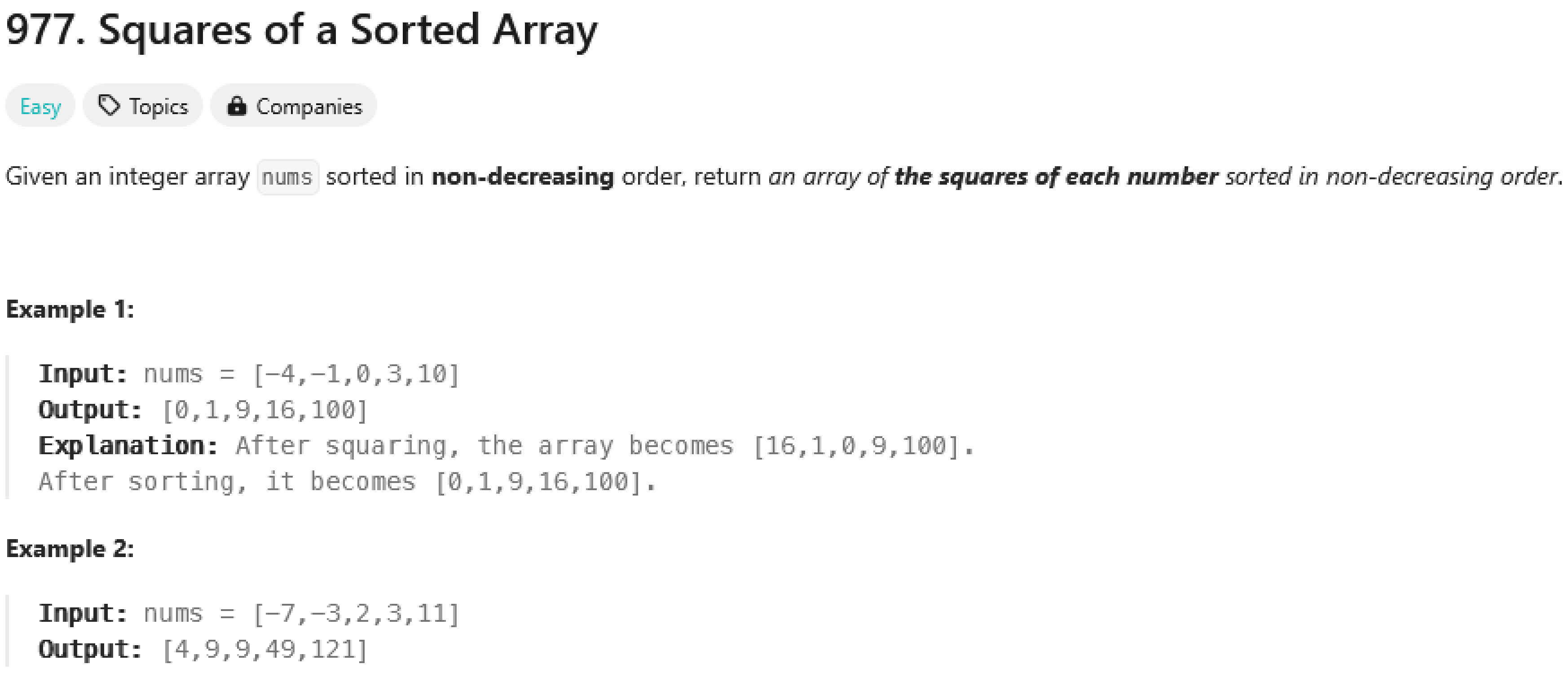
```
Input: nums = [1,4,3,2]
Output: 4
Explanation: All possible pairings (ignoring the ordering of elements) are:
1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3
2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3
3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4
So the maximum possible sum is 4.
```

Example 2:

```
Input: nums = [6,2,6,5,1,2]
Output: 9
Explanation: The optimal pairing is (2, 1), (2, 5), (6, 6). min(2, 1) + min(2, 5) + min(6, 6) = 1 + 2 + 6 = 9.
```

Exercise 3.9

- Complete LeetCode 977



977. Squares of a Sorted Array

Given an integer array `nums` sorted in **non-decreasing** order, return *an array of* ***the squares of each number*** *sorted in non-decreasing order*.

Example 1:

```
Input: nums = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
After sorting, it becomes [0,1,9,16,100].
```

Example 2:

```
Input: nums = [-7,-3,2,3,11]
Output: [4,9,9,49,121]
```