Problem 1223: Dice Roll Simulation

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

A die simulator generates a random number from

1

to

6

for each roll. You introduced a constraint to the generator such that it cannot roll the number

i

more than

rollMax[i]

(

1-indexed

) consecutive times.

Given an array of integers

rollMax

and an integer

n

, return

the number of distinct sequences that can be obtained with exact

n

rolls

. Since the answer may be too large, return it

modulo

10

9

+ 7

.

Two sequences are considered different if at least one element differs from each other.

Example 1:

Input:

n = 2, rollMax = [1,1,2,2,2,3]

Output:

34

Explanation:

There will be 2 rolls of die, if there are no constraints on the die, there are 6 * 6 = 36 possible combinations. In this case, looking at rollMax array, the numbers 1 and 2 appear at most once consecutively, therefore sequences (1,1) and (2,2) cannot occur, so the final answer is 36-2 = 34.

Example 2:

Input:

n = 2, rollMax = [1,1,1,1,1,1]

Output:

30

Example 3:

Input:

n = 3, rollMax = [1,1,1,2,2,3]

Output:

181

Constraints:

1 <= n <= 5000

rollMax.length == 6

1 <= rollMax[i] <= 15

Code Snippets

C++:

```
class Solution {
public:
int dieSimulator(int n, vector<int>& rollMax) {

}
};
```

Java:

```
class Solution {
public int dieSimulator(int n, int[] rollMax) {

}
}
```

Python3:

```
class Solution:
def dieSimulator(self, n: int, rollMax: List[int]) -> int:
```

Python:

```
class Solution(object):
def dieSimulator(self, n, rollMax):
"""
:type n: int
:type rollMax: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[]} rollMax
* @return {number}
*/
var dieSimulator = function(n, rollMax) {

};
```

TypeScript:

```
function dieSimulator(n: number, rollMax: number[]): number {

};
```

C#:

```
public class Solution {
public int DieSimulator(int n, int[] rollMax) {

}
}
```

C:

```
int dieSimulator(int n, int* rollMax, int rollMaxSize) {

}
```

Go:

```
func dieSimulator(n int, rollMax []int) int {

}
```

Kotlin:

```
class Solution {
fun dieSimulator(n: Int, rollMax: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func dieSimulator(_ n: Int, _ rollMax: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn die_simulator(n: i32, roll_max: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[]} roll_max
# @return {Integer}
def die_simulator(n, roll_max)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $rollMax
* @return Integer
*/
function dieSimulator($n, $rollMax) {

}
}
```

Dart:

```
class Solution {
int dieSimulator(int n, List<int> rollMax) {

}
}
```

Scala:

```
object Solution {
def dieSimulator(n: Int, rollMax: Array[Int]): Int = {

}
```

```
}
```

Elixir:

```
defmodule Solution do
@spec die_simulator(n :: integer, roll_max :: [integer]) :: integer
def die_simulator(n, roll_max) do

end
end
```

Erlang:

```
-spec die_simulator(N :: integer(), RollMax :: [integer()]) -> integer().
die_simulator(N, RollMax) ->
.
```

Racket:

```
(define/contract (die-simulator n rollMax)
(-> exact-integer? (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Dice Roll Simulation
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int dieSimulator(int n, vector<int>& rollMax) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Dice Roll Simulation
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int dieSimulator(int n, int[] rollMax) {

}
}
```

Python3 Solution:

```
"""
Problem: Dice Roll Simulation
Difficulty: Hard
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def dieSimulator(self, n: int, rollMax: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def dieSimulator(self, n, rollMax):
"""
:type n: int
:type rollMax: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Dice Roll Simulation
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @param {number[]} rollMax
* @return {number}
*/
var dieSimulator = function(n, rollMax) {

};
```

TypeScript Solution:

```
/**
* Problem: Dice Roll Simulation
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function dieSimulator(n: number, rollMax: number[]): number {
```

```
};
```

C# Solution:

```
/*
* Problem: Dice Roll Simulation
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int DieSimulator(int n, int[] rollMax) {

}
}
```

C Solution:

```
/*
* Problem: Dice Roll Simulation
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int dieSimulator(int n, int* rollMax, int rollMaxSize) {

}
```

Go Solution:

```
// Problem: Dice Roll Simulation
// Difficulty: Hard
```

```
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func dieSimulator(n int, rollMax []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun dieSimulator(n: Int, rollMax: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func dieSimulator(_ n: Int, _ rollMax: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Dice Roll Simulation
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn die_simulator(n: i32, roll_max: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[]} roll_max
# @return {Integer}
def die_simulator(n, roll_max)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $rollMax
* @return Integer
*/
function dieSimulator($n, $rollMax) {

}
}
```

Dart Solution:

```
class Solution {
int dieSimulator(int n, List<int> rollMax) {

}
}
```

Scala Solution:

```
object Solution {
def dieSimulator(n: Int, rollMax: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec die_simulator(n :: integer, roll_max :: [integer]) :: integer
def die_simulator(n, roll_max) do

end
end
```

Erlang Solution:

```
-spec die_simulator(N :: integer(), RollMax :: [integer()]) -> integer().
die_simulator(N, RollMax) ->
.
```

Racket Solution:

```
(define/contract (die-simulator n rollMax)
(-> exact-integer? (listof exact-integer?) exact-integer?)
)
```