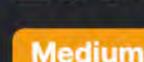
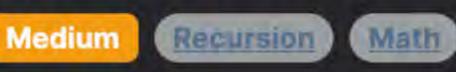
2550. Count Collisions of Monkeys on a Polygon





Leetcode Link

Problem Description

The problem describes a scenario where we have a regular convex polygon with n vertices, and each vertex is occupied by exactly one monkey. The vertices are numbered from 0 to n - 1 in a clockwise direction. The challenge is to calculate the total number of ways the monkeys can move to neighboring vertices such that at least one collision occurs. A collision is defined as either two monkeys residing on the same vertex after moving or two monkeys crossing paths along an edge. Note that a monkey can move to the vertex immediately clockwise ((i + 1) % n) or counterclockwise ((i - 1 + n) % n) to its position. Each monkey can move only once, and the final answer should be given modulo 10^9 + 7.

Intuition

All monkeys move in the clockwise direction.

To find the solution to the problem, an important observation is that a collision will not happen only in two specific scenarios:

- 2. All monkeys move in the counter-clockwise direction.
- In any other combination of movements, at least one collision is guaranteed to happen due to monkeys either ending up on the same

monkeys could decide to move. Subtracting the two scenarios where no collision occurs from the total number of possible movements gives us the total number of ways at least one collision can occur:

vertex or crossing paths. Since each monkey has two choices (clockwise or counter-clockwise), there are a total of 2ⁿ ways the

Total number of ways for at least one collision = Total ways of movement - Ways without collisions = 2^n - 2

Because the result could be very large, we compute the final answer using modulo arithmetic, specifically modulo 10^9 + 7. The

provided solution does exactly this using the modulo power function pow(2, n, mod) to calculate 2^n mod 10^9 + 7, and then

The Python solution relies on two key components of Python: the pow function and the modulo operation %.

limits. This approach elegantly handles the large number computations and efficiently computes the desired outcome with just a couple of arithmetic operations.

subtracting 2 to exclude the non-collision scenarios, followed by taking the modulo again to ensure the result is within the required

Solution Approach

The implementation of the solution involves understanding the basic properties of modular arithmetic and the power calculation. The

problem does not require complex data structures or intricate algorithms due to its nature, allowing a direct application of the mathematical insight that we derived.

• pow function: This is a built-in Python function that allows us to compute the power of a number with an optional modulus. Here, it is used to compute 2ⁿ mod (10⁹ + 7). This function is efficient for such calculations because it implements a fast exponentiation algorithm that scales well with large exponents.

• Modulo operation %: After performing the power operation, we need to ensure that we subtract two (for the two non-colliding

scenarios) in a way that respects modular arithmetic properties. The modulo operation is used again to ensure the result is within the bounds of 0 to 10^9 + 6.

The code consists of a single line within a function, which makes it quite elegant: 1 def monkeyMove(self, n: int) -> int: return (pow(2, n, mod) - 2) % mod

Here is a breakdown of what happens in this line:

```
1. We compute 2<sup>n</sup> using pow(2, n, mod). This gives us the number of all possible movements the monkeys could make by moving
  either clockwise or counterclockwise.
```

2. We subtract 2 from the result of pow(2, n, mod) to discount the scenarios where all the monkeys move in the same direction and

- no collision occurs. 3. We apply the modulo operation % mod again to ensure that the final result is expressed modulo 10^9 + 7.
- The simplicity of the approach lies in casting the problem into a binary choice for each monkey that results in a total 2ⁿ combinations and then eliminating the two combinations that do not lead to a collision. By understanding the properties of modular

arithmetic, the problem is reduced to a straightforward computation, making it an elegant example of mathematical problem-solving

Example Walkthrough Let's take n = 4 as a small example to illustrate the solution approach. This corresponds to a square where each of the four vertices is occupied by a monkey. The vertices are numbered from 0 to 3. We need to calculate the total number of ways the monkeys can move to neighboring vertices such that at least one collision occurs.

Using the intuition from the problem description, let's look at all the possible movement combinations and identify the scenarios

2. All move counter-clockwise: No collision.

where a collision will not occur.

applied to programming.

Firstly, here are all the possible movement patterns for our monkeys: 1. All move clockwise: No collision.

3. Monkey at vertex 0 moves clockwise, others move counter-clockwise: Collision occurs. 4. Monkey at vertex 1 moves clockwise, others move counter-clockwise: Collision occurs. 5. Monkey at vertex 2 moves clockwise, others move counter-clockwise: Collision occurs.

- 7. (And so on for all other combinations...)
- As described, there are 2⁴ or 16 total movement combinations for the monkeys, since each has the choice to move either clockwise

or counter-clockwise.

1 class Solution:

9

10

11

17

18

19

20

21

22

23

24

25

Out of these 16 possibilities, only 2 patterns result in no collision (see patterns 1 and 2 above). Therefore, to get the number of

6. Monkey at vertex 3 moves clockwise, others move counter-clockwise: Collision occurs.

non_collision_combinations = 2 # There are 2 non-collision scenarios. ways_with_collision = (total_combinations - non_collision_combinations) % mod # Compute the final answer with modulo. 6 print(ways_with_collision) # Output the total ways with at least one collision.

When you run this code with n = 4, you will get 14 as the number of ways at least one collision can occur (modulo $10^9 + 7$). The

logic extends to any value of n, allowing you to calculate the number of collision scenarios for any regular convex polygon with n

combinations where at least one collision occurs, we subtract these 2 non-collision scenarios from the total: 16 - 2 = 14.

2 total_combinations = pow(2, n, mod) # Compute 2^n mod (10^9 + 7) to get total possible movement combinations.

vertices in an efficient manner.

Python Solution

Define the modulo constant as BigIntegers are not efficient

Calculate 2^n using modular exponentiation, which is efficient for large powers

// This method calculates the number of ways a monkey can move, given `n` movements.

// Defining the modulo value as 1e9 + 7 to keep the result within integer limits

// It calculates the number of ways the monkey can move (minus 2 invalid ways)

// And ensures the result is non-negative after modulo operation

return (quickPower(2, numSteps) - 2 + MODULO) % MODULO;

def monkeyMove(self, n: int) -> int:

total ways = pow(2, n, MOD)

public int monkeyMove(int n) {

while (exponent > 0) {

exponent >>= 1;

final int MOD = (int) 1e9 + 7;

MOD = 10**9 + 7

We would compute this with the modulo 10^9 + 7 as follows:

1 mod = 10**9 + 7 # This is the modulo value for the problem.

```
valid_ways = (total_ways - 2) % MOD
11
12
           # Return the number of valid ways the monkey can move
           return valid_ways
Java Solution
   class Solution {
```

// If the current bit of exponent is '1', multiply the result by the current base and take modulo

// Use the quick power algorithm to calculate 2 raised to the power of `n`, reduce the result by 2, and ensure it's within th

Subtract 2 because the monkey cannot stay in the first or last column; wrap with MOD to keep result positive

// This helper method efficiently calculates (a^b) mod `mod` using the quick power algorithm. private int quickPower(long base, int exponent, int mod) { 13 // Initialize the result to 1 (identity for multiplication). 14 15 long result = 1; // Iterate as long as the exponent is greater than 0. 16

if ((exponent & 1) == 1) {

base = (base * base) % mod;

return (quickPower(2, n, MOD) - 2 + MOD) % MOD;

result = (result * base) % mod;

// Square the base and take modulo for the next bit.

// Right shift the exponent to check the next bit.

```
26
27
           // Casting the long result back to integer before returning.
28
           return (int) result;
29
30 }
31
C++ Solution
 1 class Solution {
 2 public:
       int monkeyMove(int numSteps) {
            const int MODULO = 1e9 + 7; // Constant to hold the value for modulo operation
           // Define long long alias to handle large numbers
           using Long = long long;
           // Lambda function to perform quick exponentiation (power)
10
           // This function calculates (a to the power of n) % MODULO
           auto quickPower = [&](Long base, int exponent) {
11
12
               Long result = 1;
13
               while (exponent > 0) {
                    if (exponent & 1) { // If the exponent is odd
14
                        result = (result * base) % MODULO;
15
16
17
                    base = (base * base) % MODULO; // Square the base
                   exponent >>= 1; // Divide exponent by 2
18
19
20
               return result;
21
           };
22
23
           // Calculate result using the quickPower lambda function
24
           // Formula: (2^n - 2 + MODULO) % MODULO
```

Typescript Solution

25

26

27

28

30

29 };

```
1 // Function to calculate the total number of distinct ways a monkey can move.
2 // Given 'n' steps, the monkey has 2^(n-1) possibilities for the first step
3 // and the remaining steps follow the same pattern, making the total 2^n - 2 ways.
  // The result is modulo (10^9 + 7) to keep the number within integer limits.
   function monkeyMove(n: number): number {
       // Define the modulus constant for large number calculations to ensure result is within integer bounds.
       const modulus = 10 ** 9 + 7;
       // Function to calculate (a^b) % modulus using fast exponentiation efficiently.
9
       // Handles large numbers using BigInt to avoid overflow.
10
       const quickPowerMod = (base: number, exponent: number): number => {
11
           let result = 1n; // Use BigInt for the result to handle large numbers.
12
           base = base % modulus; // Ensure base is within modulus before operations.
13
14
15
           while (exponent > 0) {
               if (exponent & 1) { // If the current exponent bit is 1, multiply to the result.
16
17
                   result = (result * BigInt(base)) % BigInt(modulus);
18
               // Square the base and reduce it modulo the modulus for the next iteration.
19
20
               base = Number((BigInt(base) * BigInt(base)) % BigInt(modulus));
21
               exponent >>>= 1; // Right shift exponent to process the next bit.
22
23
24
           return Number(result); // Convert the BigInt result back to a Number before returning.
25
       };
26
       // Use the quickPowerMod function to calculate 2^n, subtract 2 for the exact number of moves,
       // and take modulo to handle the possibility of negative results.
       return (quickPowerMod(2, n) - 2 + modulus) % modulus;
Time and Space Complexity
```

The given Python code computes the result of $2^n - 2$, modulo $10^9 + 7$. It uses the built-in pow function optimized for modular exponentiation.

27 28 29 30 } 31

Time Complexity: The primary operation of computing 2ⁿ modulo 10⁹ + 7 is performed using Python's built-in pow function. This function uses fast

Thus, the time complexity of the entire monkeyMove function is $O(\log n)$.

exponentiation.

Space Complexity: The space complexity of the code is 0(1) since it uses a constant amount of additional space. There are no data structures being

used which grow with the input size n. All operations handle intermediate values which require a constant amount of space.

exponentiation to compute the result, having a time complexity of O(log n), since it effectively halves the exponent in each step of

Post exponentiation, the subtraction and the modulo operation each take constant time, 0(1).