## 2647. Color the Triangle Red

observing patterns, we can craft a solution that meets the criteria.

# **Problem Description**

<u>Array</u>

<u>Math</u>

Hard

the large triangle is n, and thus it contains n^2 smaller triangles. The large triangle is subdivided into n rows, which are 1indexed, meaning each row number starts with 1 rather than 0. The rows have an increasing number of triangles, with row i containing 2i - 1 unit triangles. Each small triangle within the larger triangle can be referenced by coordinates, where the first number is the row number, and the

The problem presents us with a large equilateral triangle which is made up of smaller unit equilateral triangles. The side length of

second number is the position in the row, starting at 1. Two small triangles are considered neighbors if they share a side.

Initially, all the smaller triangles are white. We want to color a certain number of them red, denoted by k, and run an algorithm that propagates this color to all other triangles, under the rule that a white triangle can only turn red if it has at least two red neighbors.

The goal is to determine the minimum number k of small triangles that need to be initially colored red, and to identify their coordinates such that when the algorithm runs its course, all triangles end up being red. The question asks us to return a 2D list with the coordinates of the initially red triangles.

Intuition The key to solving this problem lies in understanding the structure of the triangle and how the coloring algorithm propagates. By

### We notice that the algorithm starts at the bottom-most row and works its way up through the rows. The first insight is the

eventually become red. The second insight is understanding that by identifying a suitable pattern which involves coloring a certain configuration of

understanding that the top row, consisting of a single unit triangle, must be colored red initially to ensure all triangles can

triangles at the bottom rows, we can ensure the propagation of the red color throughout the triangle, minimizing the number k in the process. Specifically, we notice that every fourth row from the bottom shares a coloring pattern. Coloring the last row with every triangle

red ensures that every triangle in the row above will have at least two red neighbors. Then, moving up, we color one less as we go up each subsequent row, alternating the position of the uncolored triangle every four rows. This accounts for the rule that each white triangle must have at least two red neighbors to become colored. By applying this

pattern, we make sure that with every step, more triangles turn red, ultimately leading to all triangles being red when the

By carefully coloring according to this pattern before running the algorithm, we can guarantee that we use the smallest possible

value of k to color all triangles red by propagating from the initial red triangles selected. Hence, we provide a solution that offers the coordinates of the triangles to be initially colored red, which ensures the entire triangle becomes red while respecting the rule of unit triangles needing at least two red neighbors to change color themselves. **Solution Approach** 

The solution follows an observation-based pattern-finding approach, which is then translated into a Python function for execution. This solution does not require any advanced data structures or algorithmic techniques beyond simple iteration and an understanding of the problem's geometric constraints. Here's the breakdown of the implementation:

be red initially to ensure the propagation can reach the top of the large triangle.

 $\circ$  k == 1: Here, only the second triangle of the row (i, 2) needs to be colored red initially.

coordinates), we add each triangle's coordinates to the answer list.

o k == 3: Only the first triangle of the row (i, 1) is colored red.

that represent our chosen initial red triangles.

We start by adding the first and top-most triangle's coordinates [1, 1] to our answer list, as this triangle is alone and must

Next, we need to tackle the problem from the bottom-most row upwards to the second row as noted in the reference solution

#### approach. We use a variable k to keep track of a 4-row cycle. We iterate over the rows in reverse order, starting from n down to 2,

algorithm stops.

decrementing by 1 each time. For each row i, we have four different scenarios based on the value of k: ∘ k == 0: In this case, we color every triangle on the current row. Starting from 1 to 2i - 1 stepping every two units (to stick to triangular

The variable k cycles between 0 to 3, with k = (k + 1) % 4 after handling each row. This cycle repeats and dictates the pattern for which triangles to color initially.

The Python list, ans, keeps track of the selected triangles. It's being continuously appended with the 1-indexed coordinates

By using this step-wise approach and relying on the geometry of the larger triangle, the function ensures a minimum set of

starting triangles colored red which can fill the entire larger triangle through the given algorithm. The solution is concise yet

efficient in that it only tracks the specific triangles that kick-start the propagation of color without any need for managing the

 $\circ$  k == 2: Similar to k == 0, but we start from the third position (i, 3) and add every alternate triangle up to (i, 2i - 1).

actual propagation or recalculating states, since the pattern found guarantees complete coverage. **Example Walkthrough** 

Let's go through an example with a smaller triangle to illustrate the solution approach. Suppose we have a large equilateral

triangle with a side length n = 4. The rows are numbered from 1 to 4, and they contain 1, 3, 5, and 7 smaller triangles,

First, we add the coordinates of the top-most triangle [1, 1] to our answer list. This is required to ensure the color can propagate to the top of the large triangle. Now, we tackle the problem starting from the bottom-most row and moving upwards to the second row. Since n = 4, our bottom-most row is row 4.

We initiate a variable k to keep track of a 4-row cycle. As we are dealing with a triangle of height 4, we only need to iterate

• Row 4: k == 0. Since our cycle starts with us working our way from bottom to top, we color every triangle at the bottom-most row. We add

all coordinates from (4, 1) to (4, 7) stepping by 2. Our answer list now contains [(1, 1), (4, 1), (4, 3), (4, 5), (4, 7)].

only includes (2, 3) itself. The answer list is now [(1, 1), (4, 1), (4, 3), (4, 5), (4, 7), (3, 2), (2, 3)].

need to color while ensuring eventual full coverage once the coloring algorithm runs.

# Initialize the answer list with the top-left corner red cell.

# This variable 'direction' will determine the coloring pattern.

# Case when direction is 1, paint the second cell only.

// Function that colors positions red based on the specified pattern

// Adding the first fixed pair as per the required start

# Loop through rows starting from the bottom (n) to the second row (1).

# Case when direction is 0, paint every other cell in the current row.

# Case when direction is 2, paint every other cell starting from the third.

# Update the 'direction' to the next one by taking a modulo operation.

// increasing in steps of 2 (odd numbers), stopping before 2 \* i

// Adding a single coordinate with first value as i and second value as 2

// Adding a single coordinate with first value as i and second value as 1

// Similar to direction 0, but starting at 3 and filling in odd number coordinates

// Fill the sub-array elements when the direction is 0. This fills every other element up to 2 \* row.

// Fill the sub-array elements when the direction is 2. This starts at 3 and fills every other element up to 2 \* row.

for (int j = 1; j < i << 1; j += 2) {

for (int j = 3; j < i << 1; j += 2) {

return result; // Return the final set of coordinates

// Loop to build the array based on the input number 'n'.

answer.push([row, column]);

answer.push([row, column]);

// Initialize the answer array with the first sub-array 'row' element.

for (let column = 1; column < row << 1; column += 2) {</pre>

for (let column = 3; column < row << 1; column += 2) {</pre>

// Add a single sub-array element when the direction is 1.

// Add a single sub-array element when the direction is 3.

for (let row = n, direction = 0; row > 1; --row, direction = (direction + 1) % 4) {

result.push\_back({i, j});

} else if (direction == 1) { // Case for direction 1

} else if (direction == 2) { // Case for direction 2

result.push\_back({i, j});

result.push back({i, 2});

} else { // Case for direction 3

result.push\_back({i, 1});

function colorRed(n: number): number[][] {

const answer: number[][] = [[1, 1]];

} else if (direction === 1) {

answer.push([row, 2]);

} else if (direction === 2) {

answer.push([row, 1]);

if (direction === 0) {

#### ∘ Row 3: k is now 1 (cycle moves to the next step). We only color the second triangle of the row, i.e., (3, 2). The answer list is updated to include [(1, 1), (4, 1), (4, 3), (4, 5), (4, 7), (3, 2)].

**Python** 

class Solution:

respectively.

Following the solution approach:

over rows 4 down to 2.

Here's how we apply our solutions for each row:

def colorRed(self, n: int) -> List[List[int]]:

for j in range(1, i \* 2, 2):

answer.append([i, i])

k would reset to 0 if we continued for larger triangles. The final answer is the list of coordinates: [(1, 1), (4, 1), (4, 3), (4, 5), (4, 7), (3, 2), (2, 3)]. By coloring these

triangles red initially, we ensure that the coloring propagation algorithm will turn all the triangles in the large triangle red.

∘ Row 2: k is now 2. We start from the third position and add every alternate triangle up to (2, 3). In this case, since the row is short, it

As we have reached the second row and there is no row k == 3 to apply in this case, we conclude our iteration. The variable

Solution Implementation

Using this approach, we found the minimum set k of initially red triangles for n = 4, which minimizes the number of triangles we

elif direction == 2: for i in range(3, i \* 2, 2): answer.append([i, i]) # Case when direction is 3, paint the first cell only.

else: answer.append([i, 1])

public int[][] colorRed(int n) {

return answer

Java

class Solution {

answer = [[1, 1]]

for i in range(n, 1, -1):

if direction == 0:

elif direction == 1:

answer.append([i, 2])

direction = (direction + 1) % 4

# Return the list of painted cells.

// List to store the answer pairs

answerList.add(new int[] {1, 1});

List<int[]> answerList = new ArrayList<>();

direction = 0

```
// Start from 'n' and iterate 'i' downwards.
        // Use a counter 'direction' to cycle through the 4-case pattern
        for (int i = n, direction = 0; i > 1; --i, direction = (direction + 1) % 4) {
            // Case 0: Fill columns 1 to 2*i-1 for the i-th row
            if (direction == 0) {
                for (int i = 1; i < i * 2; i += 2) {
                    answerList.add(new int[] {i, j});
            // Case 1: Add a single cell (i, 2)
            else if (direction == 1) {
                answerList.add(new int[] {i, 2});
            // Case 2: Fill columns 3 to 2*i-1 for the i-th row
            else if (direction == 2) {
                for (int i = 3; i < i * 2; i += 2) {
                    answerList.add(new int[] {i, j});
            // Case 3: Add a single cell (i, 1)
            else {
                answerList.add(new int[] {i, 1});
        // Convert the list to an array before returning
        return answerList.toArray(new int[0][]);
C++
#include <vector>
using namespace std;
class Solution {
public:
    // Method to generate color coordinates using a specific pattern
    vector<vector<int>> colorRed(int n) {
        vector<vector<int>> result: // Creating a result vector to store the color coordinates
        result.push_back({1, 1}); // Starting with the first coordinate at {1, 1}
        // Loop to calculate the coordinates based on the given pattern
        for (int i = n, direction = 0; i > 1; —i, direction = (direction + 1) % 4) { // Loop from n decreasingly and cycle direction
            if (direction == 0) { // Case for direction 0
                // Filling in coordinates with the first value fixed and the second value
```

} else {

**}**;

**TypeScript** 

```
// Return the populated answer array.
   return answer;
class Solution:
   def colorRed(self, n: int) -> List[List[int]]:
       # Initialize the answer list with the top-left corner red cell.
       answer = [[1, 1]]
       # This variable 'direction' will determine the coloring pattern.
       direction = 0
       # Loop through rows starting from the bottom (n) to the second row (1).
       for i in range(n. 1. -1):
           # Case when direction is 0, paint every other cell in the current row.
           if direction == 0:
               for j in range(1, i * 2, 2):
                   answer.append([i, i])
           # Case when direction is 1, paint the second cell only.
           elif direction == 1:
               answer.append([i, 2])
           # Case when direction is 2, paint every other cell starting from the third.
           elif direction == 2:
               for i in range(3, i * 2, 2):
                   answer.append([i, j])
           # Case when direction is 3, paint the first cell only.
           else:
               answer.append([i, 1])
           # Update the 'direction' to the next one by taking a modulo operation.
           direction = (direction + 1) % 4
       # Return the list of painted cells.
```

#### **Time Complexity** 1. In every iteration of the for i in range(n, 1, -1) loop, there is an if condition that determines what happens next. 2. When k == 0, the inner for j in range(1, i << 1, 2) loop runs roughly i/2 times.

return answer

complexity:

**Space Complexity** 

Time and Space Complexity

3. When k == 2, the similar for j in range(3, i << 1, 2) loop also runs approximately i/2 times. 4. In the other two cases (k == 1 and k == 3), there are direct appends that run in constant time 0(1). 5. Summing up the runs of the inner loop over all iterations, we get something like (n/2) + (n/2 - 1) + ... + 1, which is the sum of the first n-1

The provided Python code consists of a loop that iterates n-1 times where n is the parameter given to the colorRed function.

Inside the loop, conditional blocks execute different patterns of inner loop iterations. Here's an analysis of the time and space

Hence, each k == 0 and k == 2 case contributes a running time that is a triangular number, and the total time complexity, when combined, is quadratic 0(n^2).

Ignoring the space required to store the answer (as per the reference given), the extra space usage is as follows: 1. A couple of integer variables i, j, and k - this is constant space O(1). 2. No additional data structures or recursive calls that would consume space proportionate to the input size are made.

integers divided by 2. This sum can be expressed as  $0((n^2)/4)$ , which simplifies to  $0(n^2)$ .

Therefore, the space complexity of the code, not including the output list ans, is constant 0(1).