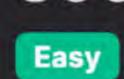
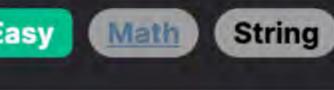
Enumeration





Problem Description

In this problem, we are dealing with hexadecimal color codes. A hexadecimal color code is a 7 character string, starting with a '#' followed by six hexadecimal digits. Each pair of digits represents a color in RGB (Red, Green, and Blue) format with values ranging from 00 to FF. For example, in #AABBCC, AA is for Red, BB for Green, and CC for Blue.

Sometimes, these color codes can be abbreviated if each pair of digits is the same. For example, #AABBCC can be abbreviated to #ABC because AA = A, BB = B, and CC = C.

The problem asks us to find the color in shorthand hexadecimal notation (#XYZ) that is most similar to the provided color (#ABCDEF) and return it as a string. The similarity between two colors #ABCDEF and #UVWXYZ is defined as -(AB - UV)^2 - (CD - WX)^2 - (EF -

YZ)^2. To clarify, we need to minimize the differences between the corresponding color components (AB and UV for Red, CD and WX for

Green, EF and YZ for Blue) of the full hexadecimal color code and its shorthand, corrosponding the given color code (#ABCDEF), to find the most similar shorthand color (#XYZ).

Intuition

shorthand approximation which can be only one of the 00, 11, 22, ..., EE, FF values in hexadecimal. Since 11 in hexadecimal is 17 in decimal, it essentially means each color component can be divided by 17 to find the nearest shorthand value. In the given solution, the f function takes a two-digit hexadecimal string and returns the shorthand notation for that specific

component. The first step is to convert the two-digit hexadecimal number to decimal and then to perform an integer division by 17

The key here is to understand that for each color component (Red, Green, Blue) in the full color code, we are looking for the closest

which gives us the multiplier for the shorthand notation. We also use the modulo operation to find the remainder, and if the remainder is greater than 8 (z > 8) we need to round up, otherwise we keep the quotient as it is. Next, we use format to convert the decimal value back to a two-digit hexadecimal notation after multiplying by 17. This operation

essentially snaps the original color component to its closest shorthand approximation. Finally, we apply this f function to each of the red (color[1:3]), green (color[3:5]), and blue (color[5:7]) components of the given

color, concatenate the results, and prefix with a '#' to return the most similar shorthand color. Solution Approach

The solution follows a step-by-step approach to find the most similar shorthand RGB notation to a given 6-digit RGB color. Let's walk

through the algorithm and patterns used: 1. Define a Helper Function f: This function takes a two-digit hexadecimal component of the original color and finds the nearest

int(x, 16), where x is the hexadecimal string and 16 is the base for conversion.

2. Conversion to Decimal: Inside the f function, the two-digit hexadecimal value is converted to its decimal equivalent using

shorthand value. The input is a string representing a two-digit hexadecimal value (e.g. AA, BC, etc.).

code (color), which are color[1:3] for red, color[3:5] for green, and color[5:7] for blue.

values. These are then concatenated in order to form the resulting shorthand color starting with #.

- 3. Find Nearest Shorthand Multiplier: The decimal number is then divided by 17 using divmod(int(x, 16), 17), which returns a quotient and a remainder. This division is based on the fact that the shorthand values in hexadecimal (00, 11, 22, ..., EE, FF) correspond to multiples of 17 in decimal.
- 4. Rounding the Multiplier: Depending on the remainder (z), if it is greater than 8, the multiplier (y) is increased by 1 for rounding to the nearest shorthand value.
- hexadecimal value using Python's format function: '{:02x}'.format(17 * y). 6. Applying the Helper Function: The main function (similarRGB) extracts the individual color components from the given color

5. Conversion Back to Hexadecimal: The closest shorthand multiplier is then multiplied by 17 and converted back into a two-digit

- 7. Construct the Result String: The helper function f is applied to each extracted component to get the shorthand hexadecimal
- It is important to note that this is more of a mathematical problem than an algorithmic one since we're performing direct calculations to find the nearest values and don't need to iterate over a set of possibilities or maintain any data structures. The solution optimizes

Example Walkthrough

the calculation by utilizing the unique property of hexadecimal color codes and the way they are presented in shorthand notations.

Here are the steps we'd follow according to the solution approach:

needed (z < 8).

closest shorthand hexadecimal color notation.

1. Define a Helper Function f: This function finds the nearest shorthand value for any two-digit hexadecimal string input it receives.

Let's go through an example to illustrate the solution approach. Assume we are given the color #09ABCD. Our task is to find the

3. Find Nearest Shorthand Multiplier: Next, we divide 9 by 17 which gives us y = 0 and a remainder z = 9 (since 9 < 17).

hexadecimal is C and no rounding is needed.

def get_similar_value(comp_hex):

comp_int = int(comp_hex, 16)

major, remainder = divmod(comp_int, 17)

4. Rounding the Multiplier: Since z > 8, we increase y by 1, leading to y = 1.

5. Conversion Back to Hexadecimal: We now convert the shorthand multiplier y = 1 into hexadecimal. 1 multiplied by 17 is 17,

○ Blue: CD in decimal is 205. 205 / 17 gives a quotient of 12 and a remainder of 1 (205 = 12 * 17 + 1), so y = 12. 12 in

2. Conversion to Decimal: For the first component (red) 09, we convert it to decimal which gives us int('09', 16) = 9.

- which is 11 in hexadecimal (' $\{:02x\}$ '.format(17 * y) = '11').
- 6. Applying the Helper Function: Repeat steps 2 to 5 for the green and blue components: • Green: AB in decimal is 171. 171 / 17 is 10 with a remainder of 1, hence y = 10. In hexadecimal, 10 is A and no rounding is
- 7. Construct the Result String: The concatenation of the shorthand hex values we have computed with # reads #1AC. So #1AC is the shorthand hexadecimal color notation that is the most similar to #09ABCD.

Extract the red, green, and blue components from the hexadecimal color string

getClosestColorComponent(blueComponent);

// Finding the nearest multiple of 17 (0x11), since all similar colors

// Helper method to find the closest component value

int value = Integer.parseInt(component, 16);

// have components that are multiples of 17

int value = stoi(component, nullptr, 16);

int roundedValue = 17 * value;

char formattedComponent[3];

value = value / 17 + (value % 17 > 8 ? 1 : 0);

// Nearest multiple of 17 (0x11), since all similar colors

// Preparing the nearest value by multiplying it by 17

sprintf(formattedComponent, "%02x", roundedValue);

// have components that are multiples of 17 (0x11 is hex for 17)

// Formatting the component as a two-digit hexadecimal string

private String getClosestColorComponent(String component) {

// Converting the hexadecimal string to an integer

Convert the two hexadecimal digits to an integer

class Solution: def similarRGB(self, color: str) -> str: # Helper function to find the closest similar value in terms of RGB

Divide the integer by 17 to find the closest factor of 17 (0x11), since we are working with values in the form of 0x11

if remainder > 8: # If the remainder is greater than half of 17, increase the major by 1 to find the closer factor of 17 10 11 major += 1# Return the string representation of the new similar component, formatted as two hexadecimal digits 12 13 return '{:02x}'.format(17 * major)

14

15

12

13

14

15

16

17

18

19

20

21

20

22

23

24

25

26

27

28

29

30

31

32

Python Solution

```
red_component = color[1:3]
16
           green_component = color[3:5]
17
           blue_component = color[5:7]
18
19
           # Get the closest similar color by applying the helper function to each RGB component
20
           # Then, concatenate the '#' symbol with the new similar components to form the hexadecimal color string
21
22
           return f'#{get_similar_value(red_component)}{get_similar_value(green_component)}{get_similar_value(blue_component)}'
23
Java Solution
  class Solution {
       // Method to find the most similar color in hexadecimal RGB format
       public String similarRGB(String color) {
           // Extracting the red, green, and blue components from the input color string
           String redComponent = color.substring(1, 3);
           String greenComponent = color.substring(3, 5);
 6
           String blueComponent = color.substring(5, 7);
9
           // Constructing the similar color by finding the closest component values
           return "#" + getClosestColorComponent(redComponent) +
10
11
                        getClosestColorComponent(greenComponent) +
```

22 value = value / 17 + (value % 17 > 8 ? 1 : 0); 23 24 // Returning the component as a 2-digit hexadecimal string 25 return String.format("%02x", 17 * value); 26 27 } 28 C++ Solution 1 class Solution { 2 public: // Method to find the closest similar color in hexadecimal RGB format string similarRGB(string color) { // Extracting the red, green, and blue components from the input color string string redComponent = color.substr(1, 2); string greenComponent = color.substr(3, 2); string blueComponent = color.substr(5, 2); // Constructing the similar color by finding the closest component values return "#" + getClosestColorComponent(redComponent) + 12 getClosestColorComponent(greenComponent) + 13 getClosestColorComponent(blueComponent); 14 15 private: // Helper method to find the closest component value 17 string getClosestColorComponent(string component) -18 // Converting the hexadecimal component to an integer 19

```
33
           // Return the formatted string
34
           return formattedComponent;
35
36 };
37
Typescript Solution
  // Function to find the most similar color in hexadecimal RGB format
 2 function similarRGB(color: string): string {
     // Extracting the red, green, and blue components from the input color string
     const redComponent = color.substring(1, 3);
     const greenComponent = color.substring(3, 5);
     const blueComponent = color.substring(5, 7);
     // Constructing the similar color by finding the closest component values
     return "#" + getClosestColorComponent(redComponent) +
                  getClosestColorComponent(greenComponent) +
10
                  getClosestColorComponent(blueComponent);
11
13
   // Helper function to find the closest component value
   function getClosestColorComponent(component: string): string {
     // Converting the hexadecimal string to an integer
16
     let value = parseInt(component, 16);
17
18
     // Finding the nearest multiple of 17 (0x11), since all similar colors
     // have components that are multiples of 17
     value = Math.floor(value / 17) + (value % 17 > 8 ? 1 : 0);
21
22
23
     // Returning the component as a 2-digit hexadecimal string
24
     return toTwoDigitHex(17 * value);
25
26
   // Helper function to convert a number to a 2-digit hexadecimal string
   function toTwoDigitHex(num: number): string {
     // Creating a hexadecimal string with padding to ensure 2 digits are returned
29
     const hex = num.toString(16);
30
```

Time and Space Complexity

return hex.length === 1 ? '0' + hex : hex;

representing the color code. Therefore, the space complexity is also 0(1).

31

33

32

Time Complexity The time complexity of the function similar RGB primarily depends on the operations involving string slicing and calculations for each of the three components of the color code (red, green, and blue). Each call to f(x) involves parsing the component as hexadecimal, performing a division and conditional operation, and then formatting the result back into a string. Since these operations are

constant in time for a given two-character string, and there are only three such strings in the input, the overall time complexity is 0(1). Space Complexity

As for the space complexity, the function uses a fixed amount of extra space for the variables a, b, c, and the result of the formatting

operation inside f(x). The space required does not change with the size of the input, as the input is always a fixed-sized string