BIO 2017 Question 1 Solution

Part A

Just... make the program!

```python
def get_input_as_list():
    input_string = input()
    return [x for x in input_string]

def find_next_layer(current_layer):
    next_layer = []
    for i in range(len(current_layer) - 1):
        if current_layer[i] == current_layer[i+1]:
            next_layer.append(current_layer[i])
        else:
            # A nice and concise way
            for c in ["R", "G", "B"]:
                if current_layer[i] != c and current_layer[i+1] != c:
                    next_layer.append(c)
                    break
    return next_layer

def main():
    current_layer = get_input_as_list()
    while len(current_layer) > 1:
        current_layer = find_next_layer(current_layer)

    print(current_layer[0])

main()
```

Part B

We have the row $RRGBRGBB$. If we chose that the row must begin with R, then we know that the next square must be R as well. This is always the case, meaning that the first square completely determines the rest of the list and therefore there can only be 3 possible rows:

$$RRRBBGGRG$$
$$GBGGRRBBB$$
$$BGBRGBRGR$$

Part C

If we know one square on each row, then we must know the square in the final row and one of the squares in the row above that. For example,

```
? ? ? ? ? ? ? ? ?
 ? ? ? ? ? ? ? ?
  ? ? ? ? ? ? ?
   ? ? ? ? ? ?
    ? ? ? ? ?
     ? ? ? ?
      ? ? ?
       G ?
        R
```

In this example, we can instantly see that only B can be put in square next to G. Now the problem becomes just like the previous one (Part B), except this time, we also know one of the squares in the row above (the row we are trying to find). Notice in our example for Part B that at each position, each solution has a different colour. Since we already have a known value in the row above, there can only be one possible solution to that row. The same applies when filling out the row above that, and so on. Hence there is only one way to fill in the triangle.

Part D

Let's mess around with this size-4 triangle. Take the following example:

```
B R G G
 G B G
  R R
   R
```

We are told that only the leftmost B and the rightmost G have any impact at all on the problem - that means the squares in the middle don't matter at all. Interesting. You know what else is interesting? The fact that the leftmost B and the rightmost G generate an R at the bottom, just like the original rules of the problem.

Let's try a few more examples to see if this always works:

```
R R R R     R B G R     B G G R
 R R R       G R B       R G B
  R R         B G         B R
   R           R           G
  yes         yes         yes
```

This does always work! It's almost as if we can say this is a recursive version of the original pattern, or the same effect on a grander scale... maybe that will be useful...

If the final square only depends can be generated solely by the left and right squares in the layer above, then surely those squares themselves can be generated in a similar fashion. Expanding the problem:

```
1 ? ? 2 ? ? 3
 ? ? ? ? ? ?
  ? ? ? ? ?
   4 ? ? 5
    ? ? ?
     ? ?
      6
```

Notice that the squares 1 and 2 completely determine square 4, and squares 2 and 3 completely determine square 5. We already know that squares 4 and 5 completely determine square 6. The issue is that we still have this annoying square 2 which isn't the leftmost or rightmost square, and it still plays a role in determining the colour in square 6.

But wait... if we use the idea that this is the same effect on a grander scale then the squares 1, 2, 3, 4, 5 and 6 can actually just be written like this, since the other squares don't matter:

```
1 2 3
 4 5
  6
```

This is valid since we already saw that the same properties are obeyed in the larger version. Can you see where this is going? That's a size-3 triangle. We don't know anything about size-3 triangles, but we do know a lot about size-4 triangles like the following:

```
1 2 3 4
 5 6 7
  8 9
   0
```

We know that squares 1 and 4 determine the colour in square 0. But if this is just a shrunken version of a larger triangle, we can re-expand this to include all the unimportant squares again:

```
1 ? ? 2 ? ? 3 ? ? 4
 ? ? ? ? ? ? ? ? ?
  ? ? ? ? ? ? ? ?
   5 ? ? 6 ? ? 7
    ? ? ? ? ? ?
     ? ? ? ? ?
      8 ? ? 9
       ? ? ?
        ? ?
         0
```

Do you see? Is it obvious now that the squares 1 and 4 must completely determine the value of the square 0? This is all because the bigger version of this works exactly the same as the smaller version. So a size-10 triangle works. How about bigger ones? It doesn't take much logic to realise that if we forget about the bottom layer, we're basically multiplying the number of additional layers by 3. We can obviously just repeat this, to get the next size that works, given by

$$(10-1)\times 3+1=28$$

or we can generalise this to see that valid sizes must of course be of the form

$$3^k + 1$$

where the $+1$ is just about adding the bottom layer back on.