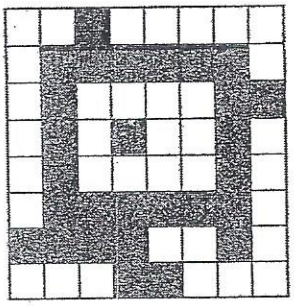
Example



On the right is an image represented as a square grid of black and white cells. Two cells in an image are part of the same "blob" if each is black and there is a sequence of moves from one cell to the other, where each move is either horizontal or vertical to an adjacent black cell. For example, the diagram represents an image that contains two blobs, one of them consisting of a single cell.

Assuming the following `Image` class declaration, you are to write the body of the `eraseBlob` method, using a recursive algorithm.

```
public class Image
{
    private final int BLACK = 1;
    private final int WHITE = 0;
    private int[][] image;    //square grid
    private int size;         //number of rows and columns

    public Image()   //constructor
    { /* implementation not shown */ }

    public void display()   //displays Image
    { /* implementation not shown */ }

    /* Precondition:  Image is defined with either BLACK or WHITE
                      cells.
     * Postcondition: If 0 <= row < size, 0 <= col < size,
     *                and image[row][col] is BLACK, set all cells
     *                in the same blob to WHITE. Otherwise image
     *                is unchanged. */
    public void eraseBlob(int row, int col)
    /* your code goes here */
}
```

Solution:

```
public void eraseBlob(int row, int col)
{
    if (row >= 0 && row < size && col >= 0 && col < size)
        if (image[row][col] == BLACK)
        {
            image[row][col] = WHITE;
            eraseBlob(row - 1, col);
            eraseBlob(row + 1, col);
            eraseBlob(row, col - 1);
            eraseBlob(row, col + 1);
        }
}
```

NOTE

1. The ordering of the four recursive calls is irrelevant.

2. The test

```
if (image[row][col] == BLACK)
```

can be included as the last piece of the test in the first line:

```
if (row >= 0 && ...
```

If `row` or `col` is out of range, the test will short-circuit, avoiding the dreaded `ArrayIndexOutOfBoundsException`.

3. If you put the statement

```
image[row][col] = WHITE;
```

after the four recursive calls, you get infinite recursion if your blob has more than one cell. This is because, when you visit an adjacent cell, one of its recursive calls visits the original cell. If this cell is still `BLACK`, yet more recursive calls are generated, *ad infinitum*.

A final thought: Recursive algorithms can be tricky. Try to state the solution recursively *in words* before you launch into code. Oh, and don't forget the base case!

```
Barron Recursion Free Response Question # 2 pg 295


blic class ImageMain

  public static void main(String[] args)
  {

      Image b = new Image();
      b.display();
      b.eraseBlob(3,3);
      b.display();
  }



ass Image


  private final int BLACK = 1;
  private final int WHITE = 0;
  private int[][] image;
  private int size;

  public Image()
  {
      size = 8;

      image =  new int[][] {{WHITE,WHITE,BLACK,WHITE,WHITE,WHITE,WHITE,WHITE},
               {WHITE,BLACK,BLACK,BLACK,BLACK,BLACK,BLACK,WHITE},
               {WHITE,BLACK,WHITE,WHITE,WHITE,WHITE,BLACK,BLACK},
               {WHITE,BLACK,WHITE,BLACK,WHITE,WHITE,BLACK,WHITE},
               {WHITE,BLACK,WHITE,WHITE,WHITE,WHITE,BLACK,WHITE},
               {WHITE,BLACK,BLACK,BLACK,BLACK,BLACK,BLACK,WHITE},
               {BLACK,BLACK,BLACK,BLACK,WHITE,WHITE,BLACK,WHITE},
               {WHITE,WHITE,WHITE,BLACK,BLACK,WHITE,WHITE,WHITE}};
  }


  public void display() {
      for (int i = 0; i < size; i++)
      {
          for (int j = 0; j < size; j++)
          {
              System.out.print(image[i][j]);
          }
          System.out.println();
      }
      System.out.println();
      System.out.println();
  }



  public void eraseBlob(int row, int col)
  {
      if (row >= 0 && row < size && col >= 0 && col < size)
      {
          if (image[row][col] == BLACK)
          {
              image[row][col] = WHITE;
              eraseBlob(row - 1, col);
              eraseBlob(row + 1, col);
              eraseBlob(row, col + 1);
              eraseBlob(row, col - 1);
          }
      }
  }
```

Sample Free-Response Question 1

Here is a sample free-response question that uses recursion in a two-dimensional array. See if you can answer it before looking at the solution.

A *color grid* is defined as a two-dimensional array whose elements are character strings having values `"b"` (blue), `"r"` (red), `"g"` (green), or `"y"` (yellow). The elements are called pixels because they represent pixel locations on a computer screen. For example,

```
b b g r        r r r r r        y g r
g r g r                         b y g
                                g r b
                                b b g
```

A *connected region* for any pixel is the set of all pixels of the same color that can be reached through a direct path along horizontal or vertical moves starting at that pixel. A connected region can consist of just a single pixel or the entire color grid. For example, if the two-dimensional array is called `pixels`, the connected region for `pixels[1][0]` is as shown here for three different arrays.

```
b b g r        y g r b        b b
g r g r        g g y g        b b
               b g r g
```

The class `ColorGrid`, whose declaration is shown below, is used for storing, displaying, and changing the colors in a color grid.

```
public class ColorGrid
{
    private String[][] myPixels;
    private int myRows;
    private int myCols;

    /**
     * Creates numRows × numCols ColorGrid from String s.
     * @param s the string containing colors of the ColorGrid
     * @param numRows the number of rows in the ColorGrid
     * @param numCols the number of columns in the ColorGrid
     */
    public ColorGrid(String s, int numRows, int numCols)
    { /* to be implemented in part (a) */ }

    /**
     * Precondition:  myPixels[row][col] is oldColor, one of "r",
     *                "b","g", or "y".
     *                newColor is one of "r","b","g", or "y".
     * Postcondition: if 0 <= row < myRows and 0 <= col < myCols,
     *                paints the connected region of
     *                myPixels[row][col] the newColor.
     *                Does nothing if oldColor is the same as
     *                newColor.
     * @param row the given row
     * @param col the given column
     * @param newColor the new color for painting
     * @param oldColor the current color of myPixels[row][col]
     */
    public void paintRegion(int row, int col, String newColor,
        String oldColor)
    { /* to be implemented in part (b) */ }

    //other methods not shown
        ...
}
```

(a) Write the implementation code for the `ColorGrid` constructor. The constructor should initialize the `myPixels` matrix of the `ColorGrid` as follows: The dimensions of `myPixels` are `numRows` × `numCols`. `String s` contains `numRows` × `numCols` characters, where each character is one of the colors of the grid—`"r"`, `"g"`, `"b"`, or `"y"`. The characters are contained in `s` row by row from top to bottom and left to right. For example, given that `numRows` is 3, and `numCols` is 4, if `s` is `"brrygrggyyyr"`, `myPixels` should be initialized to be

```
b r r y
g r g g
y y y r
```

Complete the constructor below:

```
/**
 * Creates numRows x numCols ColorGrid from String s.
 * @param s the string containing colors of the ColorGrid
 * @param numRows the number of rows in the ColorGrid
 * @param numCols the number of columns in the ColorGrid
 */
public ColorGrid(String s, int numRows, int numCols)
```

(b) Write the implementation of the `paintRegion` method as started below. Note: You must write a recursive solution. The `paintRegion` paints the connected region of the given pixel, specified by `row` and `col`, a different color specified by the `newColor` parameter. If `newColor` is the same as `oldColor`, the color of the given pixel, `paintRegion` does nothing. To visualize what `paintRegion` does, imagine that the different colors surrounding the connected region of a given pixel form a boundary. When paint is poured onto the given pixel, the new color will fill the connected region up to the boundary.

For example, the effect of the method call `c.paintRegion(2, 3, "b", "r")` on the `ColorGrid c` is shown here. (The starting pixel is shown in a frame, and its connected region is shaded.)

before	after
r r b g y y	r r b g y y
b r b y r r	b r b y b b
g g r [r] r b	g g b b b b
y r r y r b	y b b y b b

Complete the method `paintRegion` below. Note: Only a recursive solution will be accepted.

```
/**
 * Precondition:  myPixels[row][col] is oldColor, one of "r",
 *                "b","g", or "y".
 *                newColor is one of "r","b","g", or "y".
 * Postcondition: if 0 <= row < myRows and 0 <= col < myCols,
 *                paints the connected region of
 *                myPixels[row][col] the newColor.
 *                Does nothing if oldColor is the same as
 *                newColor.
 * @param row the given row
 * @param col the given column
 * @param newColor the new color for painting
 * @param oldColor the current color of myPixels[row][col]
 */
public void paintRegion(int row, int col, String newColor,
    String oldColor)
```