```
#%%

'''
I spent some time working through a more optimal way to compute the Julia set.
The posted solutions do this with
a for loop that iterates over each point in the grid and computes the number of
iterations it takes to diverge,
whereas this code vectorizes the computation using numpy. This is much more
computationally efficient and allows
higher resolution images to reasonably be computed. This method also allows for a
more artistic representation of the
Julia set as it shows the number of iterations it takes for each point to
diverge, rather than just whether or not it
diverges, giving a more detailed heatmap of the set.

With this code, I computed a 2000x2000 resolution image in 1.62s allowing 30
iterations per point. The posted solution
took 22.33s to compute the same resolution image with 30 iterations per point
(while giving less information), and
also fails without switching the abs(z) to np.abs(z) as the base abs function
cant handle how large the modulus of some
of the numbers get.

This code also computed a 10000x10000 resolution image in about a minute and a
half at 50 iterations per point.
'''


import numpy as np
import time
import plotly.graph_objects as go
import matplotlib.pyplot as plt

def juliaSet(c, xlims, ylims, res, maxIter=30, threshold=2):
    '''
    Function that computes and returns the Julia set for a given fixed complex
number. This
    function returns a heatmap of how many iterations it takes a point to diverge
that can
    be used to plot the julia set.
    '''

    # Creating a matrix of all complex numbers to check convergence in the range
    x = np.linspace(xlims[0], xlims[1], res)
    y = np.linspace(ylims[0], ylims[1], res)
```

```python
    X, Y = np.meshgrid(x, y)
    Z = X + 1j*Y

    # Setting up a convergence matrix that tracks how many iterations a point
takes to diverge
    convergence = np.full(Z.shape, maxIter)

    # Computing each starting point to check convergence
    for i in range(maxIter):

        # Checking if the point has already diverged and making a boolean matrix
to track all points that have not
        mask = np.abs(Z) <= threshold

        # Applying the Julia set formula to points that have not yet diverged
        Z[mask] = Z[mask]**2 + c

        # Updating the convergence matrix with the number of iterations it took
each point to diverge on this cycle
        convergence[mask & (np.abs(Z) > threshold)] = i


    return convergence

#%%
# Setting base conditions and generating the convergence heatmap matrix

c = complex(-0.8, 0.2)
xlims = [-1.5, 1.5]
ylims = [-1, 1]
res = 2000 #Points over the intervals xlim and ylim

startTime = time.time()
juliaMat = juliaSet(c, xlims, ylims, res, maxIter=30)
myRunTime = time.time() - startTime
print(f'My runtime: {myRunTime} seconds')



#%%
# Graphing my data using matplotlib

plt.figure(figsize=(12, 8))
plt.imshow(juliaMat,
           cmap='hot',
           extent=[xlims[0], xlims[1], ylims[0], ylims[1]],
)
plt.title('Julias Set Heatmap')
plt.xlabel('Re')
plt.ylabel('Im')
plt.colorbar(label='Iterations')

plt.show()

#%%
# Graphing my data using plotly -- my preference for a plotting library in python

fig = go.Figure(data=go.Heatmap(
    z=juliaMat,
    x=np.linspace(xlims[0], xlims[1], res),
    y=np.linspace(ylims[0], ylims[1], res),
    colorscale='hot',
    colorbar=dict(title=dict(text='Iterations', side='right')) # Setting colorbar
formatting
))

fig.update_layout(
    title='Julia Set Heatmap',
    xaxis_title='Re',
    yaxis_title='Im',
    xaxis=dict(range=[xlims[0], xlims[1]]),
    yaxis=dict(range=[ylims[0], ylims[1]]),
    width=900,
    height=600,
)

fig.show()

# %%
'''
Here is the posted solution for efficiency comparison. It is significantly slower
than my solution and produces a
much less pretty plot in my opinion.
'''
# set up arrays to hold the values
x_values = []
y_values = []
# set the c value
c = complex(-0.8, 0.2)
```

```python
startTime = time.time()
# loop over evenly spaced values from -1 to 1 along both axes
for x in np.linspace(-1, 1, res):
    for y in np.linspace(-1, 1, res):
        # start z at x,y
        z = complex(x, y)
        # loop over 30 times
        for i in range(0,30):
            z = z * z + c
            # if the modulus is less than 100, put it in the set
        if(np.abs(z) < 2):
            x_values.append(x)
            y_values.append(y)

postRunTime = time.time() - startTime
print(f'Posted runtime: {postRunTime} seconds')

# %%
plt.scatter(x_values, y_values, s=0.2)
plt.show()
# %%
```