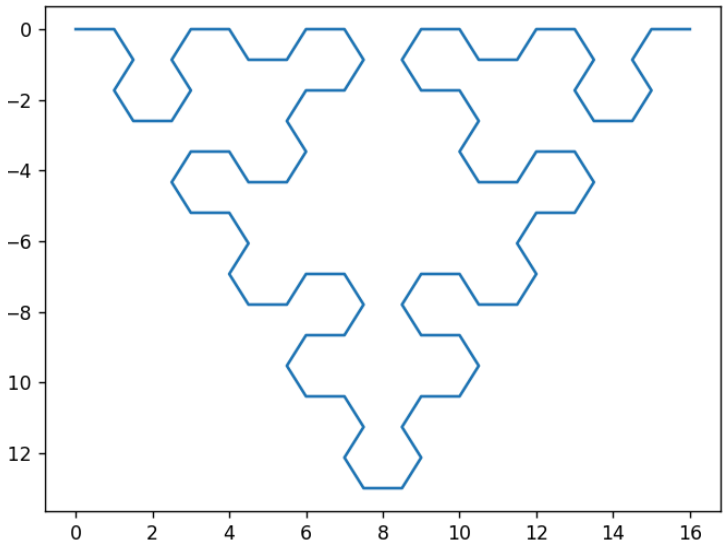
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import numpy as np
import matplotlib.pyplot as plt
import math

# Define vocabulary
vocab = {'F': 0, 'B': 1, 'X': 2, 'Y': 3, 'P': 4, 'M': 5}  # Add 'P' as
a padding token
reverse_vocab = {v: k for k, v in vocab.items()}

# Define the fractal rules
koch_snowflake_rules = {
    'F': ['F', 'F', 'B', 'B'],
    'B': ['F', 'B', 'F', 'B']
}

sierpinski_triangle_rules = {
    'X': ['Y', 'P', 'X', 'P', 'Y'],
    'Y': ['X', 'M', 'Y', 'M', 'X'],
    'P': ['P'],
    'M': ['M']
}

# Fractal Dataset
class FractalDataset(Dataset):
    def __init__(self, rules, max_depth=5, fractal_type='koch'):
        self.rules = rules
        self.max_depth = max_depth
        self.fractal_type = fractal_type
        self.data = self.generate_fractals()

    def generate_fractals(self):
        fractals = []
        for i in range(1, self.max_depth + 1):
            input_seq, output_seq = self.generate_fractal(i)
            fractals.append((input_seq, output_seq))
        return fractals

    def generate_fractal(self, depth):
        # Start with the initial string
        if self.fractal_type == 'koch':
            phrase = ['F']
        elif self.fractal_type == 'sierpinski':
            phrase = ['X']

        # Iterate according to the depth
        for _ in range(depth):
```

```
            new_phrase = []
            for symbol in phrase:
                new_phrase.extend(self.rules.get(symbol, [symbol]))
            phrase = new_phrase
        return phrase, phrase  # We use the same phrase as input and
output

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        input_seq, output_seq = self.data[idx]
        input_seq = [vocab[s] for s in input_seq]
        output_seq = [vocab[s] for s in output_seq]
        return input_seq, output_seq

# Seq2Seq Model
class Seq2Seq(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim,
embedding_dim=128):
        super(Seq2Seq, self).__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)  #
Embedding layer
        self.hidden_dim = hidden_dim
        self.encoder = nn.LSTM(embedding_dim, hidden_dim,
batch_first=True)
        self.decoder = nn.LSTM(hidden_dim, hidden_dim,
batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, input_seq):
        # Embedding layer
        embedded = self.embedding(input_seq)

        # Encoding
        _, (hidden, _) = self.encoder(embedded)

        # Decoding
        output, _ = self.decoder(embedded, (hidden, _))
        output = self.fc(output)
        return output

# Collate function for padding sequences
def collate_fn(batch):
    input_seqs, output_seqs = zip(*batch)
    input_seqs = [torch.tensor(seq, dtype=torch.long) for seq in
input_seqs]
    output_seqs = [torch.tensor(seq, dtype=torch.long) for seq in
output_seqs]
```

```
    # Pad sequences to the same length
    input_seqs_padded = torch.nn.utils.rnn.pad_sequence(input_seqs,
batch_first=True, padding_value=vocab['P'])
    output_seqs_padded = torch.nn.utils.rnn.pad_sequence(output_seqs,
batch_first=True, padding_value=vocab['P'])
    return input_seqs_padded, output_seqs_padded

# Initialize dataset and dataloader
max_depth = 5
fractal_type = 'koch'  # Change to 'sierpinski' for Sierpinski
triangle
dataset = FractalDataset(koch_snowflake_rules if fractal_type ==
'koch' else sierpinski_triangle_rules, max_depth)
dataloader = DataLoader(dataset, batch_size=2, shuffle=True,
collate_fn=collate_fn)

# Initialize the model, loss function, and optimizer
input_dim = len(vocab)  # Number of symbols in the vocabulary
hidden_dim = 128
output_dim = len(vocab)
model = Seq2Seq(input_dim, hidden_dim, output_dim)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = nn.CrossEntropyLoss(ignore_index=vocab['P'])  # Ignore
padding token in loss
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
epochs = 40
epoch_losses = []
for epoch in range(epochs):
    model.train()
    epoch_loss = 0
    for input_seq, output_seq in dataloader:
        input_seq, output_seq = input_seq.to(device),
output_seq.to(device)

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(input_seq)

        # Reshape output and output_seq to calculate loss
        output = output.view(-1, output_dim)
        output_seq = output_seq.view(-1)

        # Compute loss
        loss = criterion(output, output_seq)
```

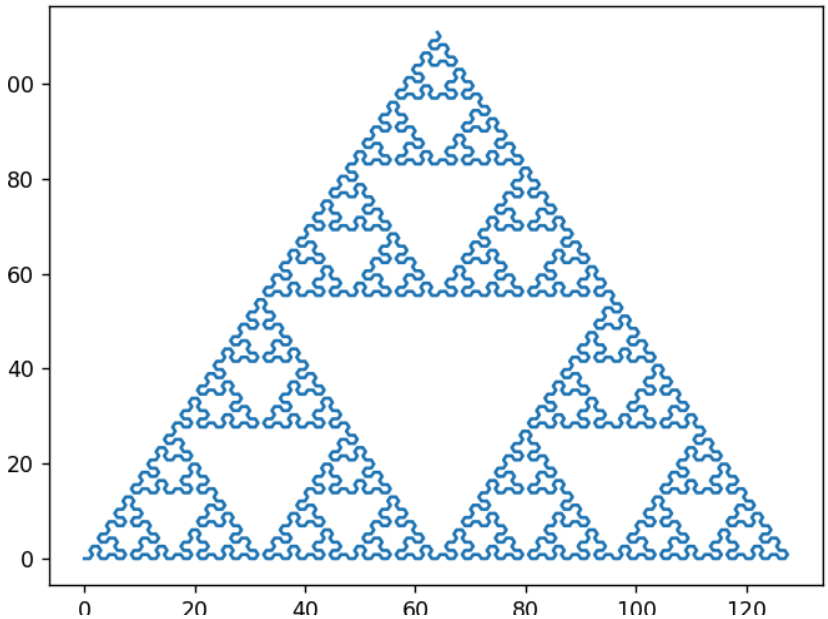
SIERPINSKI TRIANGLE



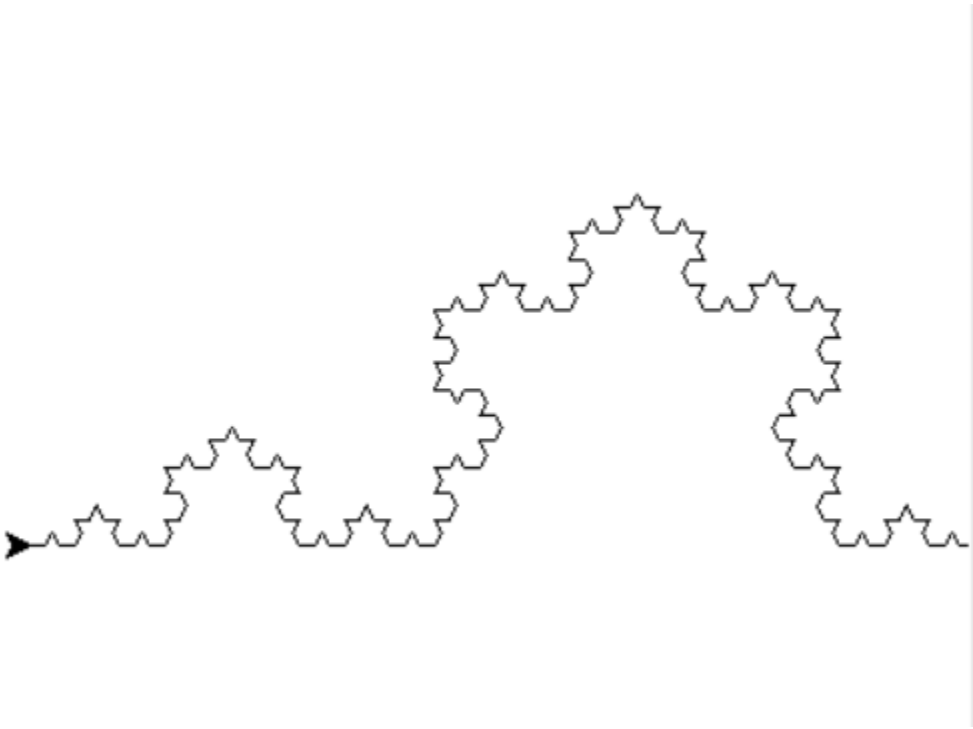
SEQUENCE

enter limit:3

xmymxpypxpypxmymxmypxpymxmymxmypxpymxmymxpypxpypxmymxpy
pxpymxmymxmypxpypxmymxpypxpypxmymxpypxpymxmymxmypxpypxmy
mxpypxpypxmymxmypxpymxmymxmypxpymxmymxpypxpypxmymx



KOCH’S SNOWFLAKE



SEQUENCE

Enter recursion depth (limit): 4

['f', 'f', 'b', 'b', 'b', 'b', 'f', 'f', 'b', 'b', 'f', 'f', 'f', 'f', 'b', 'b', 'b', 'b', 'f', 'f', 'f', 'f', 'b', 'b', 'f', 'f', 'b', 'b', 'b', 'b', 'f', 'f', 'b', 'b', 'f', 'f', 'f', 'f', 'b', 'b', 'f', 'f', 'b', 'b', 'b', 'b', 'f', 'f', 'f', 'f', 'b', 'b', 'b', 'b', 'f', 'f', 'b', 'b', 'f', 'f', 'f', 'f', 'b', 'b']

```
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    epoch_losses.append(epoch_loss / len(dataloader))
    print(f'Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss/len(dataloader)}')
```

```
Epoch 1/40, Loss: 1.6193701028823853
Epoch 2/40, Loss: 1.0368828972180684
Epoch 3/40, Loss: 0.6373928785324097
Epoch 4/40, Loss: 0.398540198802948
Epoch 5/40, Loss: 0.24859429895877838
Epoch 6/40, Loss: 0.1613376041253408
Epoch 7/40, Loss: 0.10702384263277054
Epoch 8/40, Loss: 0.07392699768145879
Epoch 9/40, Loss: 0.0528667705754439
Epoch 10/40, Loss: 0.039289296915133796
Epoch 11/40, Loss: 0.030209428320328396
Epoch 12/40, Loss: 0.024080105125904083
Epoch 13/40, Loss: 0.019722378502289455
Epoch 14/40, Loss: 0.016564464817444485
Epoch 15/40, Loss: 0.014177580984930197
Epoch 16/40, Loss: 0.012455772298077742
Epoch 17/40, Loss: 0.01097479990373055
Epoch 18/40, Loss: 0.009863068349659443
Epoch 19/40, Loss: 0.008941091286639372
Epoch 20/40, Loss: 0.008499624518056711
Epoch 21/40, Loss: 0.007603804115206003
Epoch 22/40, Loss: 0.007271444424986839
Epoch 23/40, Loss: 0.0065979377056161565
Epoch 24/40, Loss: 0.00618502264842391
Epoch 25/40, Loss: 0.005845997482538223
Epoch 26/40, Loss: 0.005529631860554218
Epoch 27/40, Loss: 0.0053785916728278
Epoch 28/40, Loss: 0.004985795356333256
Epoch 29/40, Loss: 0.004736336724211772
Epoch 30/40, Loss: 0.004548306266466777
Epoch 31/40, Loss: 0.0044487725632886095
Epoch 32/40, Loss: 0.004172452259808779
Epoch 33/40, Loss: 0.004007600713521242
Epoch 34/40, Loss: 0.00385422189719975
Epoch 35/40, Loss: 0.0037141641757140556
Epoch 36/40, Loss: 0.003563332293803493
Epoch 37/40, Loss: 0.0034553942581017814
Epoch 38/40, Loss: 0.0033286112205435834
Epoch 39/40, Loss: 0.003210880017528931
Epoch 40/40, Loss: 0.003121413988992572
```

```
# Plot loss function graph
plt.plot(range(epochs), epoch_losses, label='Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.show()
```