```
import tensorflow as tf

# You'll generate plots of attention in order to see which parts of an image
# our model focuses on during captioning
import matplotlib.pyplot as plt

# Scikit-learn includes many helpful utilities
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

import re
import numpy as np
import os
import time
import json
from glob import glob
from PIL import Image
import pickle
```

```
# Download caption annotation files
annotation_folder = '/annotations/'
if not os.path.exists(os.path.abspath('.') + annotation_folder):
  annotation_zip = tf.keras.utils.get_file('captions.zip',
                                           cache_subdir=os.path.abspath('.'),
                                           origin = 'http://images.cocodataset.or
                                           extract = True)
  annotation_file = os.path.dirname(annotation_zip)+'/annotations/captions_train
  os.remove(annotation_zip)

# Download image files
image_folder = '/train2014/'
if not os.path.exists(os.path.abspath('.') + image_folder):
  image_zip = tf.keras.utils.get_file('train2014.zip',
                                      cache_subdir=os.path.abspath('.'),
                                      origin = 'http://images.cocodataset.org/zi
                                      extract = True)
  PATH = os.path.dirname(image_zip) + image_folder
  os.remove(image_zip)
else:
  PATH = os.path.abspath('.') + image_folder
```

```
Downloading data from http://images.cocodataset.org/annotations/annotations_tra
252878848/252872794 [==============================] - 5s 0us/step
Downloading data from http://images.cocodataset.org/zips/train2014.zip
13510574080/13510573713 [==============================] - 326s 0us/step
```

```
# Read the json file
with open(annotation_file, 'r') as f:
    annotations = json.load(f)

# Store captions and image names in vectors
all_captions = []
all_img_name_vector = []
```

```
for annot in annotations['annotations']:
    caption = '<start> ' + annot['caption'] + ' <end>'
    image_id = annot['image_id']
    full_coco_image_path = PATH + 'COCO_train2014_' + '%012d.jpg' % (image_id)

    all_img_name_vector.append(full_coco_image_path)
    all_captions.append(caption)

# Shuffle captions and image_names together
# Set a random state
train_captions, img_name_vector = shuffle(all_captions,
                                          all_img_name_vector,
                                          random_state=1)

# Select the first 30000 captions from the shuffled set
num_examples = 30000
train_captions = train_captions[:num_examples]
img_name_vector = img_name_vector[:num_examples]
```

```
len(train_captions), len(all_captions)
```

```
(30000, 414113)
```

```
def load_image(image_path):
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, (299, 299))
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    return img, image_path
```

```
image_model = tf.keras.applications.InceptionV3(include_top=False,
                                                 weights='imagenet')
new_input = image_model.input
hidden_layer = image_model.layers[-1].output

image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicat
87916544/87910968 [==============================] - 1s 0us/step
```

```
!pip install tqdm
```

```
Requirement already satisfied: tqdm in /usr/local/lib/python3.6/dist-packages
```

```
from tqdm import tqdm


```

```
# Get unique images
encode_train = sorted(set(img_name_vector))

# Feel free to change batch_size according to your system configuration
```

```
image_dataset = tf.data.Dataset.from_tensor_slices(encode_train)
image_dataset = image_dataset.map(
  load_image, num_parallel_calls=tf.data.experimental.AUTOTUNE).batch(16)

for img, path in tqdm(image_dataset):
  batch_features = image_features_extract_model(img)
  batch_features = tf.reshape(batch_features,
                              (batch_features.shape[0], -1, batch_features.shape

  for bf, p in zip(batch_features, path):
    path_of_feature = p.numpy().decode("utf-8")
    np.save(path_of_feature, bf.numpy())
```

```
1622it [05:34,  4.85it/s]
```

```
# Find the maximum length of any caption in our dataset
def calc_max_length(tensor):
    return max(len(t) for t in tensor)
```

```
# Choose the top 5000 words from the vocabulary
top_k = 5000
tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=top_k,
                                                  oov_token="<unk>",
                                                  filters='!"#$%&()*+.,-/:;=?@[\
tokenizer.fit_on_texts(train_captions)
train_seqs = tokenizer.texts_to_sequences(train_captions)
```

```
tokenizer.word_index['<pad>'] = 0
tokenizer.index_word[0] = '<pad>'
```

```
# Create the tokenized vectors
train_seqs = tokenizer.texts_to_sequences(train_captions)
```

```
# Pad each vector to the max_length of the captions
# If you do not provide a max_length value, pad_sequences calculates it automati
cap_vector = tf.keras.preprocessing.sequence.pad_sequences(train_seqs, padding='
```

```
# Calculates the max_length, which is used to store the attention weights
max_length = calc_max_length(train_seqs)
```

```
# Create training and validation sets using an 80-20 split
img_name_train, img_name_val, cap_train, cap_val = train_test_split(img_name_vec
                                                                    cap_vector,
                                                                    test_size=0.
                                                                    random_state
```

```
len(img_name_train), len(cap_train), len(img_name_val), len(cap_val)
```

```
(24000, 24000, 6000, 6000)
```

```
# Feel free to change these parameters according to your system's configuration
```

```
BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
vocab_size = top_k + 1
num_steps = len(img_name_train) // BATCH_SIZE
# Shape of the vector extracted from InceptionV3 is (64, 2048)
# These two variables represent that vector shape
features_shape = 2048
attention_features_shape = 64
```

```
# Load the numpy files
def map_func(img_name, cap):
  img_tensor = np.load(img_name.decode('utf-8')+'.npy')
  return img_tensor, cap
```

```
dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

# Use map to load the numpy files in parallel
dataset = dataset.map(lambda item1, item2: tf.numpy_function(
          map_func, [item1, item2], [tf.float32, tf.int32]),
          num_parallel_calls=tf.data.experimental.AUTOTUNE)

# Shuffle and batch
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

```
class BahdanauAttention(tf.keras.Model):
  def __init__(self, units):
    super(BahdanauAttention, self).__init__()
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, features, hidden):
    # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)

    # hidden shape == (batch_size, hidden_size)
    # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
    hidden_with_time_axis = tf.expand_dims(hidden, 1)

    # score shape == (batch_size, 64, hidden_size)
    score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))

    # attention_weights shape == (batch_size, 64, 1)
    # you get 1 at the last axis because you are applying score to self.V
    attention_weights = tf.nn.softmax(self.V(score), axis=1)

    # context_vector shape after sum == (batch_size, hidden_size)
    context_vector = attention_weights * features
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights
```

```
class CNN_Encoder(tf.keras.Model):
    # Since you have already extracted the features and dumped it using pickle
    # This encoder passes those features through a Fully connected layer
    def __init__(self, embedding_dim):
        super(CNN_Encoder, self).__init__()
        # shape after fc == (batch_size, 64, embedding_dim)
        self.fc = tf.keras.layers.Dense(embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x
```

```
class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units, vocab_size):
    super(RNN_Decoder, self).__init__()
    self.units = units

    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')
    self.fc1 = tf.keras.layers.Dense(self.units)
    self.fc2 = tf.keras.layers.Dense(vocab_size)

    self.attention = BahdanauAttention(self.units)

  def call(self, x, features, hidden):
    # defining attention as a separate model
    context_vector, attention_weights = self.attention(features, hidden)

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x)

    # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # passing the concatenated vector to the GRU
    output, state = self.gru(x)

    # shape == (batch_size, max_length, hidden_size)
    x = self.fc1(output)

    # x shape == (batch_size * max_length, hidden_size)
    x = tf.reshape(x, (-1, x.shape[2]))

    # output shape == (batch_size * max_length, vocab)
    x = self.fc2(x)

    return x, state, attention_weights

  def reset_state(self, batch_size):
```

```
    return tf.zeros((batch_size, self.units))
```

```
encoder = CNN_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim, units, vocab_size)
```

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  loss_ = loss_object(real, pred)

  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask

  return tf.reduce_mean(loss_)
```

```
checkpoint_path = "./checkpoints/train"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
                           optimizer = optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)
```

```
start_epoch = 0
if ckpt_manager.latest_checkpoint:
  start_epoch = int(ckpt_manager.latest_checkpoint.split('-')[-1])
  # restoring the latest checkpoint in checkpoint_path
  ckpt.restore(ckpt_manager.latest_checkpoint)
```

## Training

```
# adding this in a separate cell because if you run the training cell
# many times, the loss_plot array will be reset
loss_plot = []
```

```
@tf.function
def train_step(img_tensor, target):
  loss = 0

  # initializing the hidden state for each batch
  # because the captions are not related from image to image
  hidden = decoder.reset_state(batch_size=target.shape[0])

  dec_input = tf.expand_dims([tokenizer.word_index['<start>']] * target.shape[0]

  with tf.GradientTape() as tape:
      features = encoder(img_tensor)

      for i in range(1, target.shape[1]):
          # passing the features through the decoder
          predictions, hidden, _ = decoder(dec_input, features, hidden)
```

```
            loss += loss_function(target[:, i], predictions)

            # using teacher forcing
            dec_input = tf.expand_dims(target[:, i], 1)

    total_loss = (loss / int(target.shape[1]))

    trainable_variables = encoder.trainable_variables + decoder.trainable_variable

    gradients = tape.gradient(loss, trainable_variables)

    optimizer.apply_gradients(zip(gradients, trainable_variables))

    return loss, total_loss
```

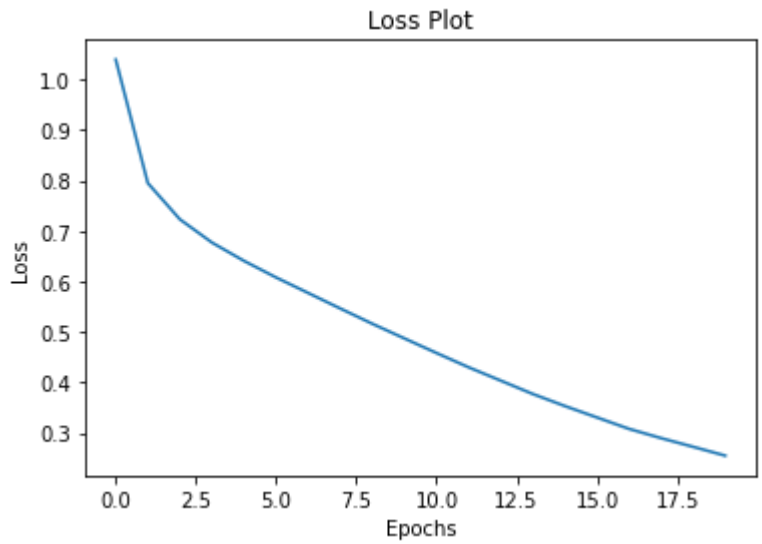
```
EPOCHS = 20

for epoch in range(start_epoch, EPOCHS):
    start = time.time()
    total_loss = 0

    for (batch, (img_tensor, target)) in enumerate(dataset):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss

        if batch % 100 == 0:
            print ('Epoch {} Batch {} Loss {:.4f}'.format(
              epoch + 1, batch, batch_loss.numpy() / int(target.shape[1])))
    # storing the epoch end loss value to plot later
    loss_plot.append(total_loss / num_steps)

    if epoch % 5 == 0:
      ckpt_manager.save()

    print ('Epoch {} Loss {:.6f}'.format(epoch + 1,
                                        total_loss/num_steps))
    print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))
```

```
Epoch 1 Batch 0 Loss 1.9629
Epoch 1 Batch 100 Loss 1.0641
Epoch 1 Batch 200 Loss 1.0097
Epoch 1 Batch 300 Loss 0.9003
Epoch 1 Loss 1.039736
Time taken for 1 epoch 371.2840042114258 sec

Epoch 2 Batch 0 Loss 0.7678
Epoch 2 Batch 100 Loss 0.8646
Epoch 2 Batch 200 Loss 0.9211
Epoch 2 Batch 300 Loss 0.7804
Epoch 2 Loss 0.795022
Time taken for 1 epoch 352.27037048339844 sec

Epoch 3 Batch 0 Loss 0.7298
Epoch 3 Batch 100 Loss 0.7947
Epoch 3 Batch 200 Loss 0.6727
Epoch 3 Batch 300 Loss 0.7272
Epoch 3 Loss 0.723231
Time taken for 1 epoch 349.2909016609192 sec

Epoch 4 Batch 0 Loss 0.6679
Epoch 4 Batch 100 Loss 0.6928
Epoch 4 Batch 200 Loss 0.6878
Epoch 4 Batch 300 Loss 0.6621
Epoch 4 Loss 0.677160
Time taken for 1 epoch 348.0951991081238 sec

Epoch 5 Batch 0 Loss 0.6100
Epoch 5 Batch 100 Loss 0.6671
Epoch 5 Batch 200 Loss 0.5985
Epoch 5 Batch 300 Loss 0.5920
Epoch 5 Loss 0.640795
Time taken for 1 epoch 347.40416526794434 sec

Epoch 6 Batch 0 Loss 0.6440
Epoch 6 Batch 100 Loss 0.5976
Epoch 6 Batch 200 Loss 0.6401
Epoch 6 Batch 300 Loss 0.6115
Epoch 6 Loss 0.607960
Time taken for 1 epoch 346.4037923812866 sec

Epoch 7 Batch 0 Loss 0.5286
Epoch 7 Batch 100 Loss 0.6015
Epoch 7 Batch 200 Loss 0.6223
Epoch 7 Batch 300 Loss 0.5785
Epoch 7 Loss 0.577220
Time taken for 1 epoch 347.7673637866974 sec

Epoch 8 Batch 0 Loss 0.5193
Epoch 8 Batch 100 Loss 0.5708
Epoch 8 Batch 200 Loss 0.5374
Epoch 8 Batch 300 Loss 0.5351
Epoch 8 Loss 0.546481
Time taken for 1 epoch 346.2396740913391 sec

Epoch 9 Batch 0 Loss 0.5695
Epoch 9 Batch 100 Loss 0.5326
Epoch 9 Batch 200 Loss 0.5300
Epoch 9 Batch 300 Loss 0.5603
Epoch 9 Loss 0.516191
```

```
Time taken for 1 epoch 346.0829641819 sec

Epoch 10 Batch 0 Loss 0.4980
Epoch 10 Batch 100 Loss 0.4978
Epoch 10 Batch 200 Loss 0.4818
Epoch 10 Batch 300 Loss 0.4839
Epoch 10 Loss 0.487158
Time taken for 1 epoch 346.4885756969452 sec

Epoch 11 Batch 0 Loss 0.4478
Epoch 11 Batch 100 Loss 0.5002
Epoch 11 Batch 200 Loss 0.4418
Epoch 11 Batch 300 Loss 0.4151
Epoch 11 Loss 0.458268
Time taken for 1 epoch 346.2432270050049 sec

Epoch 12 Batch 0 Loss 0.4603
Epoch 12 Batch 100 Loss 0.4438
Epoch 12 Batch 200 Loss 0.3858
Epoch 12 Batch 300 Loss 0.4008
Epoch 12 Loss 0.429968
Time taken for 1 epoch 345.3726074695587 sec

Epoch 13 Batch 0 Loss 0.4132
Epoch 13 Batch 100 Loss 0.4539
Epoch 13 Batch 200 Loss 0.3743
Epoch 13 Batch 300 Loss 0.3916
Epoch 13 Loss 0.403569
Time taken for 1 epoch 345.8834364414215 sec

Epoch 14 Batch 0 Loss 0.4062
Epoch 14 Batch 100 Loss 0.3725
Epoch 14 Batch 200 Loss 0.3627
Epoch 14 Batch 300 Loss 0.3957
Epoch 14 Loss 0.377096
Time taken for 1 epoch 345.52081847190857 sec

Epoch 15 Batch 0 Loss 0.3490
Epoch 15 Batch 100 Loss 0.3553
Epoch 15 Batch 200 Loss 0.3811
Epoch 15 Batch 300 Loss 0.3235
Epoch 15 Loss 0.353224
Time taken for 1 epoch 345.56896018981934 sec

Epoch 16 Batch 0 Loss 0.4001
Epoch 16 Batch 100 Loss 0.3422
Epoch 16 Batch 200 Loss 0.3360
Epoch 16 Batch 300 Loss 0.3295
Epoch 16 Loss 0.330572
Time taken for 1 epoch 345.6349630355835 sec

Epoch 17 Batch 0 Loss 0.3651
Epoch 17 Batch 100 Loss 0.3014
Epoch 17 Batch 200 Loss 0.3094
Epoch 17 Batch 300 Loss 0.3254
Epoch 17 Loss 0.307942
Time taken for 1 epoch 346.03713154792786 sec

Epoch 18 Batch 0 Loss 0.3328
Epoch 18 Batch 100 Loss 0.2998
Epoch 18 Batch 200 Loss 0.2927
Epoch 18 Batch 300 Loss 0.2962
```

```
Epoch 18 Loss 0.289562
Time taken for 1 epoch 345.2148702144623 sec

Epoch 19 Batch 0 Loss 0.2967
Epoch 19 Batch 100 Loss 0.2594
Epoch 19 Batch 200 Loss 0.2840
Epoch 19 Batch 300 Loss 0.2857
Epoch 19 Loss 0.272336
Time taken for 1 epoch 345.64116311073303 sec

Epoch 20 Batch 0 Loss 0.3181
Epoch 20 Batch 100 Loss 0.2437
Epoch 20 Batch 200 Loss 0.2525
Epoch 20 Batch 300 Loss 0.2457
Epoch 20 Loss 0.254878
Time taken for 1 epoch 344.66556310653687 sec
```

```
plt.plot(loss_plot)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Plot')
plt.show()
```




## Caption

```
def evaluate(image):
    attention_plot = np.zeros((max_length, attention_features_shape))

    hidden = decoder.reset_state(batch_size=1)

    temp_input = tf.expand_dims(load_image(image)[0], 0)
    img_tensor_val = image_features_extract_model(temp_input)
    img_tensor_val = tf.reshape(img_tensor_val, (img_tensor_val.shape[0], -1, im

    features = encoder(img_tensor_val)

    dec_input = tf.expand_dims([tokenizer.word_index['<start>']], 0)
    result = []
```

```
    attention_plot = [] 

    for i in range(max_length):
        predictions, hidden, attention_weights = decoder(dec_input, features, hi

        attention_plot[i] = tf.reshape(attention_weights, (-1, )).numpy()

        predicted_id = tf.random.categorical(predictions, 1)[0][0].numpy()
        result.append(tokenizer.index_word[predicted_id])

        if tokenizer.index_word[predicted_id] == '<end>':
            return result, attention_plot

        dec_input = tf.expand_dims([predicted_id], 0)

    attention_plot = attention_plot[:len(result), :]
    return result, attention_plot
```

```
def plot_attention(image, result, attention_plot):
    temp_image = np.array(Image.open(image))

    fig = plt.figure(figsize=(10, 10))

    len_result = len(result)
    for l in range(len_result):
        temp_att = np.resize(attention_plot[l], (8, 8))
        ax = fig.add_subplot(len_result//2, len_result//2, l+1)
        ax.set_title(result[l])
        img = ax.imshow(temp_image)
        ax.imshow(temp_att, cmap='gray', alpha=0.6, extent=img.get_extent())

    plt.tight_layout()
    plt.show()
```

```
# captions on the validation set
rid = np.random.randint(0, len(img_name_val))
image = img_name_val[rid]
real_caption = ' '.join([tokenizer.index_word[i] for i in cap_val[rid] if i not
result, attention_plot = evaluate(image)

print ('Real Caption:', real_caption)
print ('Prediction Caption:', ' '.join(result))
#plot_attention(image, result, attention_plot)

```

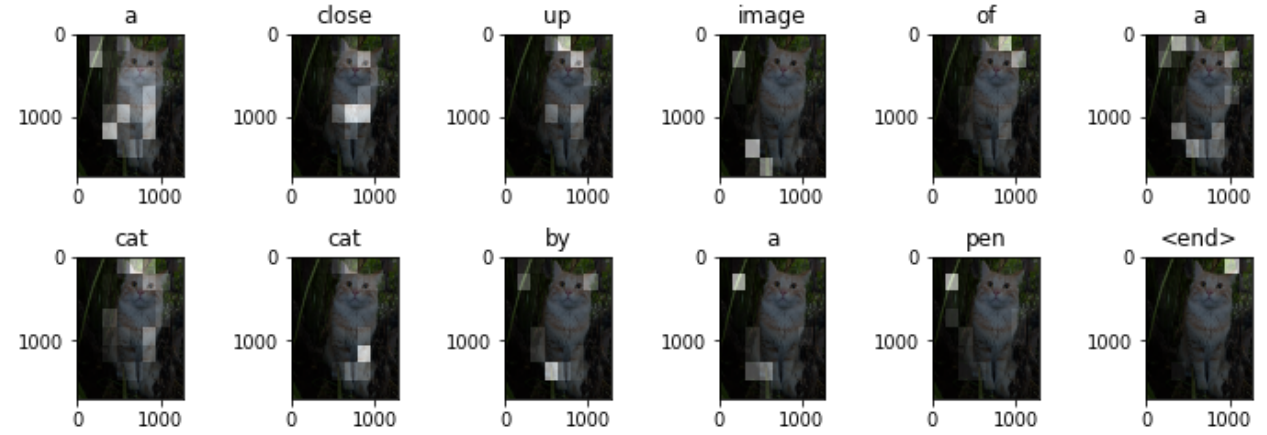
```
Real Caption: <start> a tennis player waiting for the ball to be served <end>
Prediction Caption: a player dressed in red and <unk> a tennis <end>
```

## Try

```
image_url = 'https://upload.wikimedia.org/wikipedia/commons/thumb/4/4b/Domestic_
image_extension = image_url[-4:]
image_path = tf.keras.utils.get_file('image15'+image_extension,
                                     origin=image_url)
```

```
5
6 result, attention_plot = evaluate(image_path)
7 print ('Prediction Caption:', ' '.join(result))
8 plot_attention(image_path, result, attention_plot)
9 # opening the image
10 Image.open(image_path)
```

```
Downloading data from https://upload.wikimedia.org/wikipedia/commons/thumb/4/4
475136/474072 [==============================] - 0s 1us/step
Prediction Caption: a close up image of a cat cat by a pen <end>
```




1