Guess My Word

In Guess My Word [http://hryanjones.com/guess-my-word/](http://hryanjones.com/guess-my-word/), you make guesses at a secret word and th Being bad at it, Will wants to write a program to solve it. In this lab, you’ll explore a couple ideas he ha:

```python
import matplotlib.pyplot as plt
import numpy as np
```

Modeling the Game

Suppose we have obtained a list of the words used in the game. Each word has a frequency represent English language. Suppose the secret word is chosen proportionally to this frequency. The code in the things to note:

- The frequency is not a probability i.e. it is unnormalized.
- In this version of the game, we’re assuming guesses come from the list of possible words used.
- We’re also assuming that you aren’t told the secret word if you guess it. You have to deduce it yo

```python
from words import WORD_FREQ
type(WORD_FREQ)

dict

total_freq = sum(WORD_FREQ.values())
sorted_words = sorted(list(WORD_FREQ))
sorted_word_freqs = np.array([WORD_FREQ[w] for w in sorted_words])
cum_freqs = np.cumsum(sorted_word_freqs)

print(f'Total frequency: {total_freq}
print(f'First 10 words sorted: {sorted_words[:10]}
print(f'Frequencies associated with these words: {sorted_word_freqs[:10]}
print(f'Cumulative freqs associated with these words: {cum_freqs[:10]}

Total frequency: 37009
First 10 words sorted: ['abstain', 'accept', 'across', 'action', 'active', 'adher
Frequencies associated with these words: [  2  24 111 122  23   1 116 221  16   1
Cumulative freqs associated with these words: [  2  26 137 259 282 283 399 620 63

The following function will use binary search to find the word associated with an idx, which should be

```python
def binary_search_word_generation(idx):
```
i, j = 0, len(sorted_words) - 1
while i < j:
    mid = (i + j) // 2
    if cum_freqs[mid] > idx:
        j = mid
    else:
        i = mid + 1
return i

In the following cell, implement `generate_secret_word` using `binary_search_word_generation`.

```python
def generate_secret_word():
    # BEGIN YOUR SOLUTION
    rand_idx = np.random.randint(total_freq)
    return sorted_words[binary_search_word_generation(rand_idx)]
    # END YOUR SOLUTION
```

This cell creates a game by generating a secret word and returning `guess_function`. `guess_function` lexicographically "less than" the secret word.

```python
def create_game(secret_word):
    def guess_function(guess):
        assert guess in WORD_FREQ
        return guess < secret_word
    return guess_function
```

### Binary Search Idea

The first idea is to ignore the frequencies and binary search on the correct word. In the cell below, write a method and returns the number of guesses.

*Hint: A good way to binary search would be to model it off of `binary_search_word_generation`.*

```python
def binary_search_game(guess_function):
    guesses = 0
deduced_word = None
    # BEGIN YOUR SOLUTION
    i, j = 0, len(sorted_words) - 1
    while i < j:
        mid = (i + j) // 2
        word_to_guess = sorted_words[mid]
        if guess_function(word_to_guess):
            i = mid + 1
        else:
            j = mid
        guesses += 1
deduced_word = sorted_words[i]
    # END YOUR SOLUTION
```
Disclaimer: Your implementation may not get the exact same values.

```python
def test_binary_search_game():
    '''Sanity checks for binary_search_game.

    While your implementation may give different numbers, the number of guesses
    should be 9 or 10.
    '''
    words = ['find', 'time', 'great', 'anyone', 'lying']
    print([(binary_search_game(create_game(w)) for w in words)]
    
    # Staff solution gets [('find', 10), ('time', 10), ('great', 10), ('anyone', 10), ('lying', 9)]
```

Huffman Coding Idea

The second idea uses the frequencies and is similar to **Huffman Coding**.

It works as follows:

1. Sort all the words in alphabetical order in a list.
2. Go through all consecutive pairs of words. For each pair, consider the sum of the two frequencies. Find the pair with the smallest sum.
3. Combine those two words into a node, and make the frequency their sum. The node's left child will be the first word in the pair, and the right child will be the second node in the pair.
4. Replace the two words with the node in the list.
5. Go back to step 1, and repeat until there's only one node left.
6. This node is the root of the Huffman Tree.

When creating new nodes, we also need to set `rightmost_word`. It'll be useful when we try to guess the rightmost word of the current node's left child.

```python
class HuffmanNode:
    def __init__(self, freq, word=None):
        # The total frequency of the words in our own subtree.
        self.freq = freq
        # Only non-null in leaf nodes.
        self.word = word
        # The rightmost word in our own subtree.
```
self.rightmost_word = word
    # Our children nodes
self.left_subtree = None
self.right_subtree = None

In the following cell, fill code to generate the HuffmanTree.

def build_huffman_tree():
    sorted_words = sorted(list(WORD_FREQ))
    nodes = [HuffmanNode(WORD_FREQ[w], w) for w in sorted_words]
    # Will need to combine two nodes len(WORD_FREQ) - 1 times.
    for i in range(len(WORD_FREQ) - 1):
        min_node_idx = None
        min_freq = None
        # Choose the consecutive pair of nodes with smallest frequency sum to
        # combine.
        for j in range(0, len(nodes) - 1):
            freq = nodes[j].freq + nodes[j+1].freq
            if min_freq is None or freq < min_freq:
                min_node_idx = j
                min_freq = freq
        new_node = HuffmanNode(min_freq)
        # Setup the new node and then replace the current two nodes in the list
        # with the new node.
        # BEGIN YOUR SOLUTION
        new_node.left_subtree = nodes[min_node_idx]
        new_node.right_subtree = nodes[min_node_idx + 1]
        new_node.rightmost_word = new_node.right_subtree.rightmost_word
        nodes.pop(min_node_idx)
        nodes.pop(min_node_idx)
        nodes.insert(min_node_idx, new_node)
        # END YOUR SOLUTION
    return nodes[0]

In the cell below, write code that plays a game with the HuffmanTree and returns the number of guesses.

root = build_huffman_tree()

def huffman_game(guess_function):
    guesses = 0
    deduced_word = None
    # BEGIN YOUR SOLUTION
    curr_node = root
    while curr_node.left_subtree:
        if guess_function(curr_node.left_subtree.rightmost_word):
            curr_node = curr_node.right_subtree
        else:
            curr_node = curr_node.left_subtree
        guesses += 1
    # END YOUR SOLUTION
    return guesses
deduced_word = curr_node.word
# END YOUR SOLUTION
return deduced_word, guesses

Disclaimer: Your implementation may not get the exact same values.

def test_huffman_game():
    '''Sanity checks for huffman_game.

    The number of guesses may not be less than binary search for particular
    examples.
    '''
    words = ['find', 'time', 'great', 'anyone', 'lying']
    print([huffman_game(create_game(w)) for w in words])

    # Staff solution gets [('find', 8), ('time', 6), ('great', 8), ('anyone', 8), ('lying'
    test_huffman_game()

Analysis

In this section, we'll try to compare the average number of guesses required by each of the two metho

In the following cell, compute the entropy of the distribution for the secret word.

Note: By default, np.log uses the natural log.

entropy = None
## BEGIN YOUR SOLUTION
probs = sorted_word_freqs / total_freq
entropy = np.sum(-probs * np.log2(probs))
print(entropy)
## END YOUR SOLUTION

7.93574630307442

We'll use sampling to estimate the mean number of guesses for each method. In particular, if $X_i$ is th
we can approximate

$$E[X] \approx \frac{1}{n} \sum_{i=1}^{n} X_i$$

SAMPLES = 100000
binary_search = np.zeros(SAMPLES)
huffman = np.zeros(SAMPLES)
for i in range(SAMPLES):
secret_word = generate_secret_word()
_, binary_search[i] = binary_search_game(create_game(secret_word))
_, huffman[i] = huffman_game(create_game(secret_word))

bins = np.arange(4, 15)
plt.hist(binary_search, bins, alpha=0.5, label='Binary Search')
plt.hist(huffman, bins, alpha=0.5, label='Huffman')
plt.xlabel('Number of Guesses')
plt.legend()
plt.title('Guess My Word')

Let's create confidence intervals for $E[X]$. We know that $\frac{1}{n} \sum_{i=1}^{n} X_i \sim \mathcal{N}(E[X], \frac{\text{var}(X)}{n})$ by the CLT and $\mu$. We can use the sample mean to estimate $E[X]$ and sample variance to estimate $\text{var}(X)$.

def confidence_interval_95(samples):
    # BEGIN YOUR SOLUTION
    sigma_n = np.std(samples) / np.sqrt(len(samples))
    mu = np.mean(samples)
    # END YOUR SOLUTION
    return f'{mu:.3f} \pm 1.96 \times {sigma_n:.3f}'

print(f'Binary Search: {confidence_interval_95(binary_search)}')
print(f'Huffman: {confidence_interval_95(huffman)}')

Binary Search: 9.682 +- 0.003
Huffman: 8.266 +- 0.013