Hangman

given:
- a set of words (dictionary) { apple, banana, orange, kiwi }
- the word to guess is an element of the dictionary
- the length of the word to guess

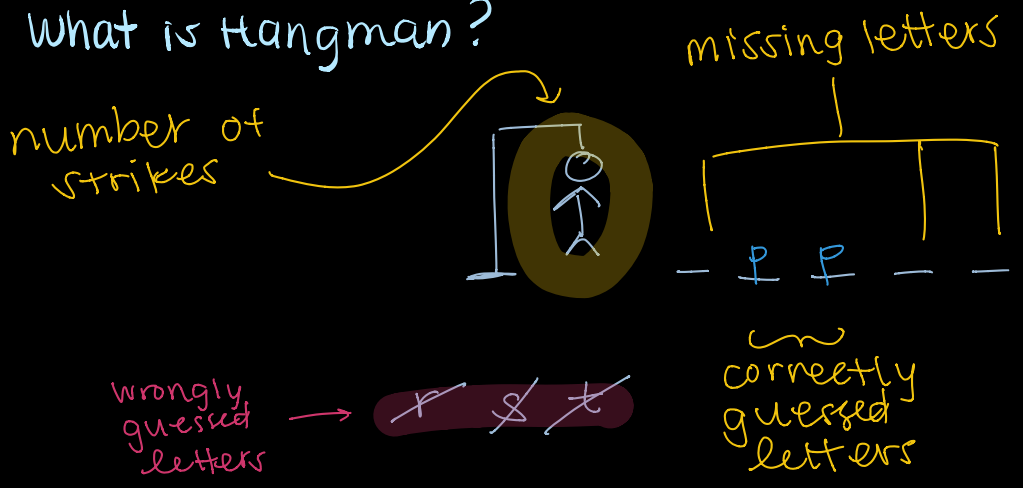
_ _ _ _ _ _ length = 6

- if a letter is guessed correctly, its position(s) are revealed

_ _ _ _ _ e e's position = 6

Determine an optimal algorithm for correctly guessing the answer.

What is Hangman?



Designing the Algorithm (initial thoughts)

- since we are given the length of the answer, we can eliminate words in the dictionary that do not match the length

if length = 6, d = { banana, orange }

- we are given a dictionary of possible answers, therefore it contains a set of letters we should guess as well. for example, given { banana, orange }, we would not guess "z".

- guessing a letter vs a word?
  as we prune off the possibilities of the answer from our dictionary, at what point is it best to start guessing the word?
  - if d's size = 1, then the one remaining word being the answer is 100%.
  - this seems relative to the remaining number of chances. (if chances matter)

Scenarios:

- if the number of words remaining in the dictionary is > the number of missing positions
  ? - should we prune the dictionary based on which letters are most frequent, or least frequent? We want to reduce our dictionary size as soon as possible.

ex) d: { aab, aac, aad, aae } → a a _

ex) d: { aabc, aacd, aade, aaee } → a a _ _

b = 1
c = 2
d = 2
e = 3
(instances)

guess → correctly

remaining
1
2
2
2

ex) d: { aabf, aacg, aadh, aaei, aaf j }

every letter except f has a frequency of 1.
- if f is guessed correctly, d = 2
- if f is incorrect, d = 3
- if g is guessed correctly, d = 1
- if g is incorrect, d = 4

f = higher frequency, less risky / g = lower freq., high risk

Recap:

Assuming strikes do not matter for now, the order of conditions to apply to the dictionary are:

1) matches length of answer
2) guess letters with high frequency in the dictionary
   2.5) use regex to further trim the dictionary based on letter & position
3) when d = 1, we have our answer

Data structures:

1) _ _ _ _ _ e

- an array with each element as a character
- if we make a correct guess, we make a new regex & prune the dictionary
  - A missing character is either represented as
    - (anychar) or [a-z] function formRegex()

2) Dictionary letter frequency

- a map of all letters to the number of instances
- this sucks because of having to iterate over each letter

e.g. { apple, banana, orange, kiwi }

a	5
p	2
l	1
e	2
b	1
n	3
o	1

r	1
g	1
k	1
i	2
w	1

function getMostFrequentLetter

- A recount would need to be done each time a letter is successfully guessed, since we don't know what other letters would be eliminated.
- grab the key with the highest frequency & choose as the next guess function guessLetter()

Implementation

Map

```
function playHangman(dictionary, answerLen){
    // 1. remove all entries that do not match the ans. length
    let prunedDictionary = dictionary.filter(entry =>
            entry.length === answerLen);

    let currentRegex = [];
    for(let i=0; i<answerLen; i++){
        currentRegex[i] = "[a-z]";
    }

    while(prunedDictionary.size !== 1){
        // 2. produce a map of letter frequencies, retrieve
            letter w/ highest frequency
        let letterGuess = this.getMostFrequentLetter(prunedDictionary);

        let positions=this.guessLetter(letterGuess, answerLen);

        // 3. Apply regex to dictionary entries, given positions of
        //    the guessed letter

        if(!positions.isEmpty()){
            currentRegex = this.generateRegex(letterGuess, positions);
            prunedDictionary= dictionary.filter(entry =>
                                entry.match(currentRegex.toString()));
        }
    }
    return dictionary.first() || null;
}
```

```
function getMostFrequentLetter(dictionary) {
    let frequencyMap = new Map();
    let maxFrequency = 0;
    let mostFrequentLetter;
    for(entry in dictionary){
        for (letter in entry) { // this is awful
            const currentFrequency = (frequencyMap.
                                      get(letter) +1 || 1);

            frequencyMap.push(letter, currentFrequeny);
            if (currentFrequeny > maxFrequency) {
               mostFrequentLetter = letter);
            }
        }
    }
    return mostFrequentLetter;
}


//assume the real answer is some class variable.
// this function supplies an array with the index
// positions of the guessed letter.
function guessLetter(letterGuess, answerLen){
    let correctPositions = [];
    for(let i=0; i < answerLen; i++) {
        if (this.realAnswer[i] === letterGuess){
            correctPositions.push(i);
        }
    }
    return correctPositions;
}
```

```
function generateRegex (currentRegex, letterGuess, positions){
    for(position in positions){
        currentRegex[position] = letterGuess;
    }
    return currentRegex;
}
```