Problem #1

The reason we use logs for computing our posterior, priors, and likelihoods is to avoid floating point errors. In essence, when we multiply all of our likelihoods together (which are all very small numbers), the number will get so small that the computer eventually just processes it as zero, instead of its true value. By calculating things as logs, we avoid hitting those really small values, and can just hold values as logs until the very end when we re-calculate them to the power of e.

Problem #2

When we calculate posteriors, we end up with an integral as our denominator. However, since W and W' come from the same distribution, this integral ends up being the exact same value. In other words, this integral is not dependent on the values of W or W', but rather the distribution itself (which is the same for W and W'). So, when we calculate:

$$\text{Posterior}(W') \,/\, \text{Posterior}(W) = (P(W'|D) \,/\, \text{Integral}) \,/\, (P(W|D) \,/\, \text{Integral})$$

which is simplified to:

$$(P(W'|D) \,/\, \text{Integral}) * (\text{Integral} \,/\, (P(W|D)) = P(W'|D) \,/\, P(W|D)$$

And thus, the final calculation (in red) is equal to our initial value (in purple), saving us from having to calculate an integral.

Problem #3

```
#3
def log_prior(W):
    # computes the log prior with a given W: e^(-W)
    if W < 0:
        return 0
    else:
        return np.log(np.exp(-W))

def log_posterior(prior, likelihood):
    # calculates the log posterior given a prior and a likelihood
    return prior + likelihood
```

Problem #4

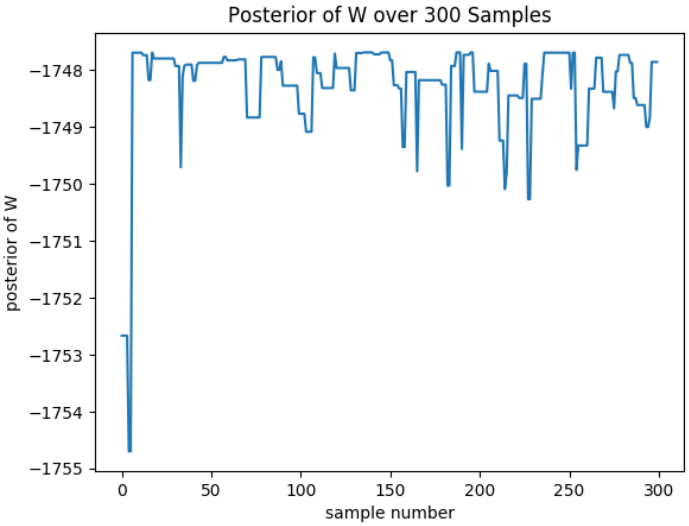
(Code)

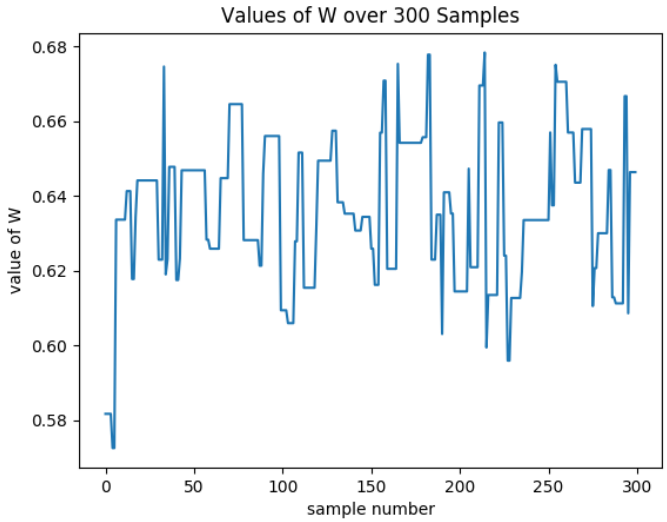
```
#4
def Metropolis(n):
    # implmentation of the Metropolis algorithm that runs
    # over the first n samples

    # generate W and its posterior, and initialize the
    # arrays we will use to store values over samples
    w = np.random.uniform()
    values_array = [w]
    w_pos = log_posterior(log_prior(w), log_likelihood(n1, n2, a, w))
    posterior_array = [w_pos]

    for i in range(0, n - 1):
        # generate W prime and its posterior
        noise = np.random.normal(0, 0.1)
        wP = w + noise
        wP_pos = log_posterior(log_prior(wP), log_likelihood(n1, n2, a, wP))
        w_pos = posterior_array[i]

        # perform the division between the two posteriors
        # comparison = P(W'|D)/P(W|D)
        comparison = np.exp(wP_pos - w_pos)

        # if W prime is better, accept it as new W
        if comparison > 1:
            w = wP
        # if W prime not better, accept it with probability
        else:
            w = np.random.choice([w, wP], 1, p = [1 - comparison, comparison])[0]

        # append this run's values to our values and posteriors arrays
        values_array.append(w)
        if w == wP:
            posterior_array.append(wP_pos)
        else:
            posterior_array.append(w_pos)

    return (values_array, posterior_array)
```

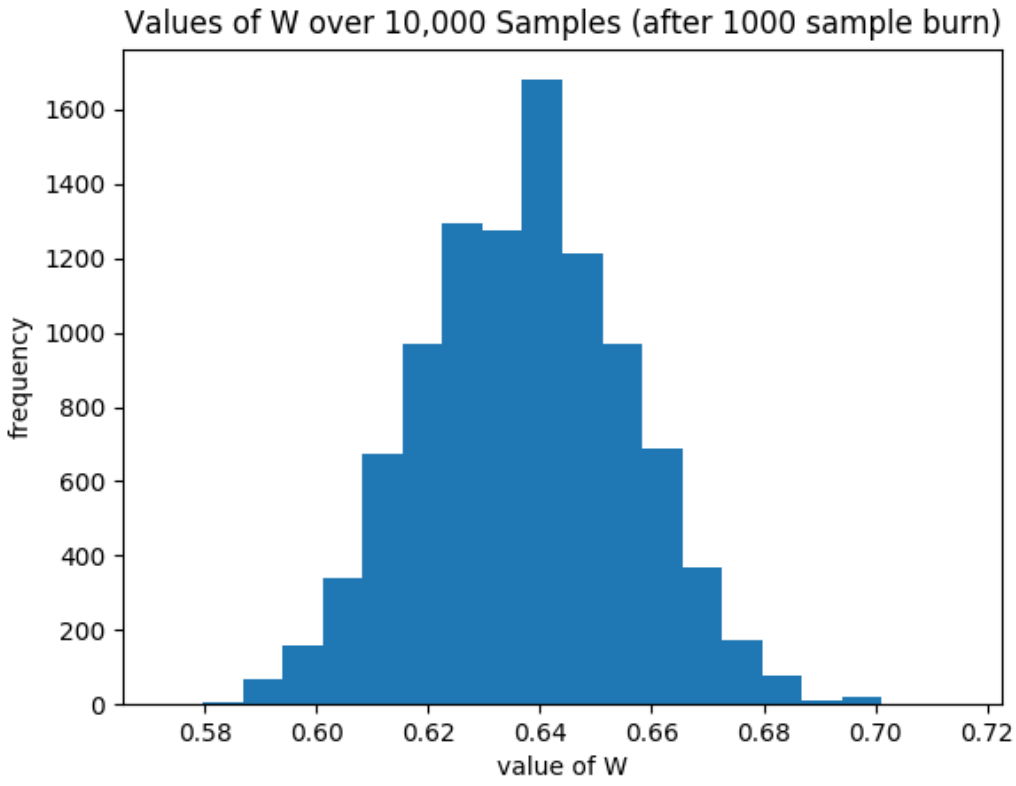
```
#4(a)
results = Metropolis(300)
samples = np.arange(300)
posteriors = results[1]
plt.xlabel('sample number')
plt.ylabel('posterior of W')
plt.title('Posterior of W over 300 Samples')
plt.plot(samples, posteriors)
plt.show()
#4(b)
values = results[0]
plt.xlabel('sample number')
plt.ylabel('value of W')
plt.title('Values of W over 300 Samples')
plt.plot(samples, values)
plt.show()
#4(c)
results = Metropolis(11000)
samples = np.arange(10000)
values = results[0][1000:]
plt.xlabel('value of W')
plt.ylabel('frequency')
plt.title('Values of W over 10,000 Samples (after 1000 sample burn)')
plt.hist(values, 20)
plt.show()
```

Problem #4

(Graphs)







Problem #5

```
#5
count = 0
for value in values:
    if (value >= 0.6) and (value <= 0.65):
        count += 1
count = count / 10000
probability = "The probability that W is in the interval 0.6 and 0.65 is " + str(count)
print(probability)
```

```
The probability that W is in the interval 0.6 and 0.65 is 0.7499
```

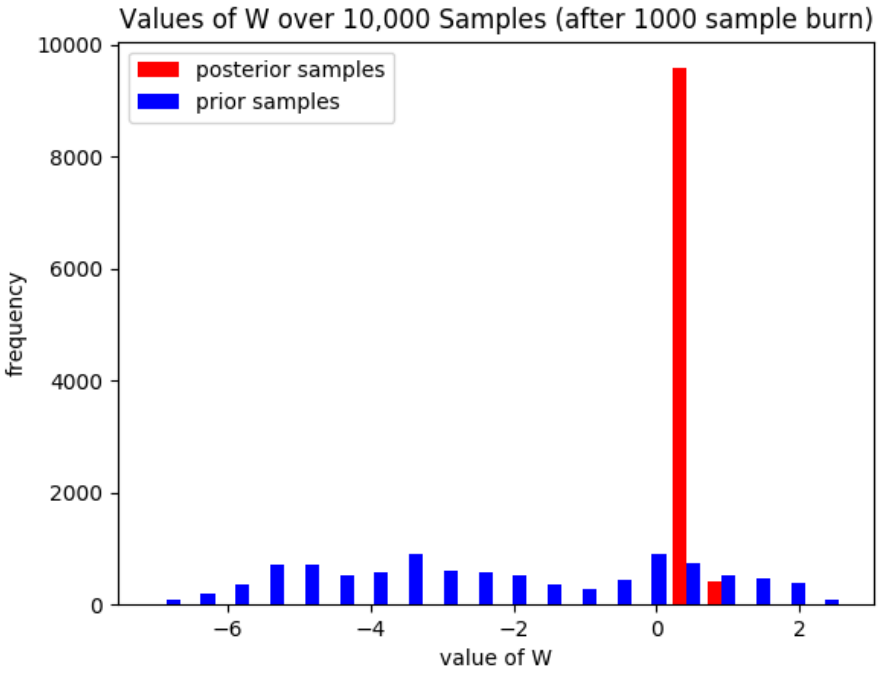
Problem #6

(Code)

```
def Metropolis_Prior(n):
    # implmentation of the Metropolis algorithm that runs
    # over the first n samples

    # generate W and its prior, and initialize the
    # arrays we will use to store values over samples
    w = np.random.uniform()
    values_array = [w]
    w_pos = log_prior(w)
    posterior_array = [w_pos]

    for i in range(0, n - 1):
        # generate W prime and its prior
        noise = np.random.normal(0, 0.1)
        wP = w + noise
        wP_pos = log_prior(wP)
        w_pos = posterior_array[i]

        # perform the division between the two posteriors
        # comparison = P(W'|D)/P(W|D)
        comparison = np.exp(wP_pos - w_pos)

        # if W prime is better, accept it as new W
        if comparison > 1:
            w = wP
        # if W prime not better, accept it with probability
        else:
            w = np.random.choice([w, wP], 1, p = [1 - comparison, comparison])[0]

        # append this run's values to our values and posteriors arrays
        values_array.append(w)
        if w == wP:
            posterior_array.append(wP_pos)
        else:
            posterior_array.append(w_pos)

    return (values_array, posterior_array)

results1 = Metropolis(11000)
results2 = Metropolis_Prior(11000)
values1 = results1[0][1000:]
values2 = results2[0][1000:]
plt.xlabel('value of W')
plt.ylabel('frequency')
plt.title('Values of W over 10,000 Samples (after 1000 sample burn)')
plt.hist([values1, values2], 20, color = ['red', 'blue'], label = ['posterior samples', 'prior samples'])
plt.legend()
plt.show()
```

Problem #6

(Graph)



From the graph above we can see that there is virtually no relationship between the histograms for the samples of the posterior and the sample of the priors. This is because the samples for the prior do not take into account the likelihood of the hypothesis, like we do when calculating the posterior. Because of this, the samples of the posterior are much more accurate.