Implementation of Shaffer et al

Please indicate at the top of your assignment whether or not you used any AI tools, such as MS Copilot. If you did use one of these tools, please provide a very brief explanation alongside each answer for how you confirmed the correctness of your solution.

- Used MS Copilot in some cases

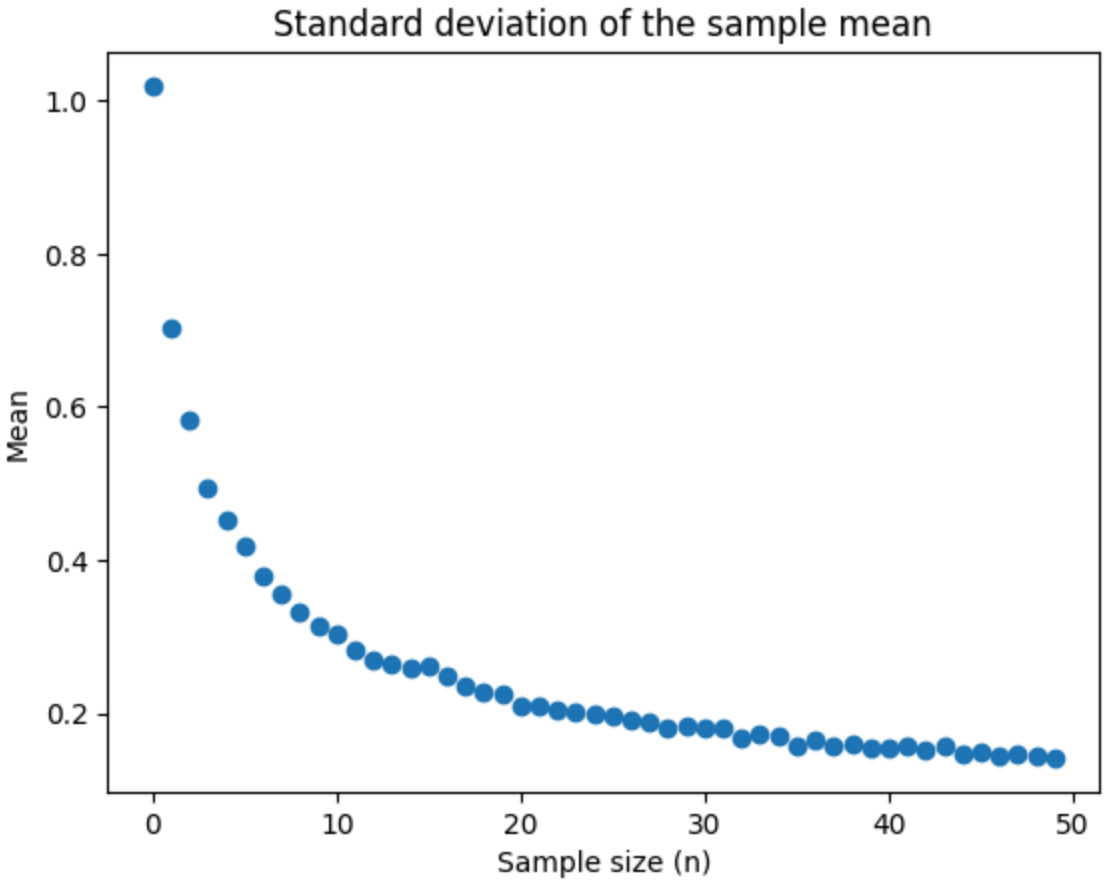
In [8]:
```
import numpy as np # We'll need numpy later
from scipy.stats import kstest, ttest_ind, ks_2samp, zscore, iqr
import matplotlib.pyplot as plt # This lets us access the pyplot functions
import pandas as pd
```

(1) Estimation of a sample mean from a normally distributed variable.

Let us assume that a true distribution of a process is described by the normal distribution with $\mu = 5$ and $\sigma = 1$. You have a measurement technique that allows you to sample n points from this distribution. In Python we can use a random number generator whose numbers will be chosen from the desired normal distribution by using the function `np.random.normal`. Sample from this normal distribution from n=1 to 50 (i.e. n=1:50). Create a plot for the standard deviation of the calculated mean from each n when you repeat the sampling 2000 times each. (i.e. You will repeat your n observations 2000 times and will calculate the sample mean for each of the 2000 trials).

In [4]:
```
# Hint: You can get 50 normally-distributed random numbers all at once:
# x = np.random.normal(mean, standard_deviation, size=50)
mu = 5
std = 1

stdev_sample = np.zeros(50)

# Explore how this works. It uses array processing along an axis to avoid an additional
# for the repeated sampling.
for n in range(50):
    sample_data = np.random.normal(mu, std, size=(n + 1, 2000))

    # axis=0 means only take the mean across that axis
    sample_mean = np.mean(sample_data, axis=0)
    assert sample_mean.shape == (2000,)

    # Now we can take the standard deviation across the remaining axis
    stdev_sample[n] = np.std(sample_mean)


plt.scatter(range(50), stdev_sample)
plt.xlabel("Sample size (n)")
plt.ylabel("Mean")
plt.title("Standard deviation of the sample mean")
```

Out[4]:
```
Text(0.5, 1.0, 'Standard deviation of the sample mean')
```



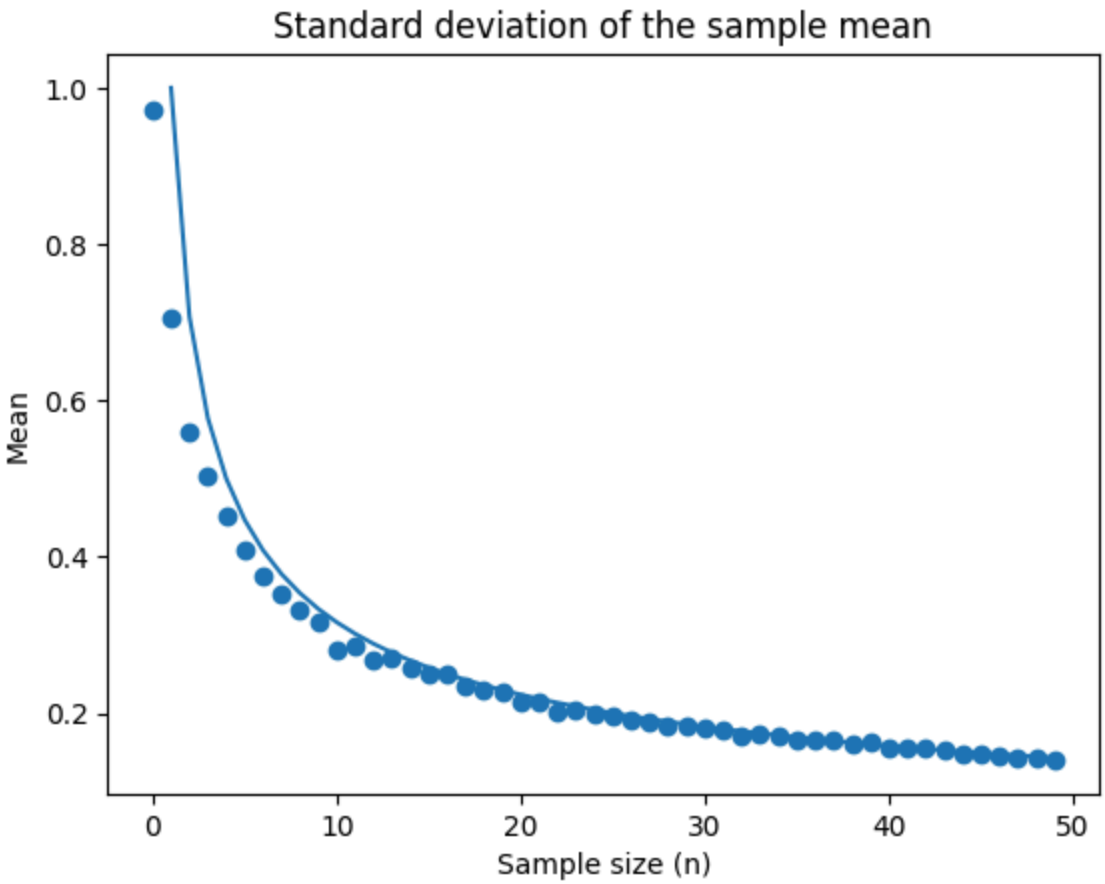
1a. Plot the standard deviation of the sample mean versus n. Add a second line which is 1/sqrt(n). Describe what this tells you about the relationship between n and your power to estimate the underlying mean.

The function 1/sqrt(n) tells us that the standard deviation of the sample mean decreases as the sample size n increases. It informs us that with increasing sample size the mean tends to approach 0 which is what the central limit theorem asserts.

In [14]:
```
# Answer to 1a here
# plt.scatter(x, y) and plt.plot(x, y) will give you scatter and line plots

plt.scatter(range(50), stdev_sample)
plt.plot(range(50), 1/np.sqrt(range(50)))
plt.xlabel("Sample size (n)")
plt.ylabel("Mean")
plt.title("Standard deviation of the sample mean")
```

```
/tmp/ipykernel_9454/391474209.py:5: RuntimeWarning: divide by zero encountered in divide
  plt.plot(range(50), 1/np.sqrt(range(50)))
```

Out[14]:
```
Text(0.5, 1.0, 'Standard deviation of the sample mean')
```



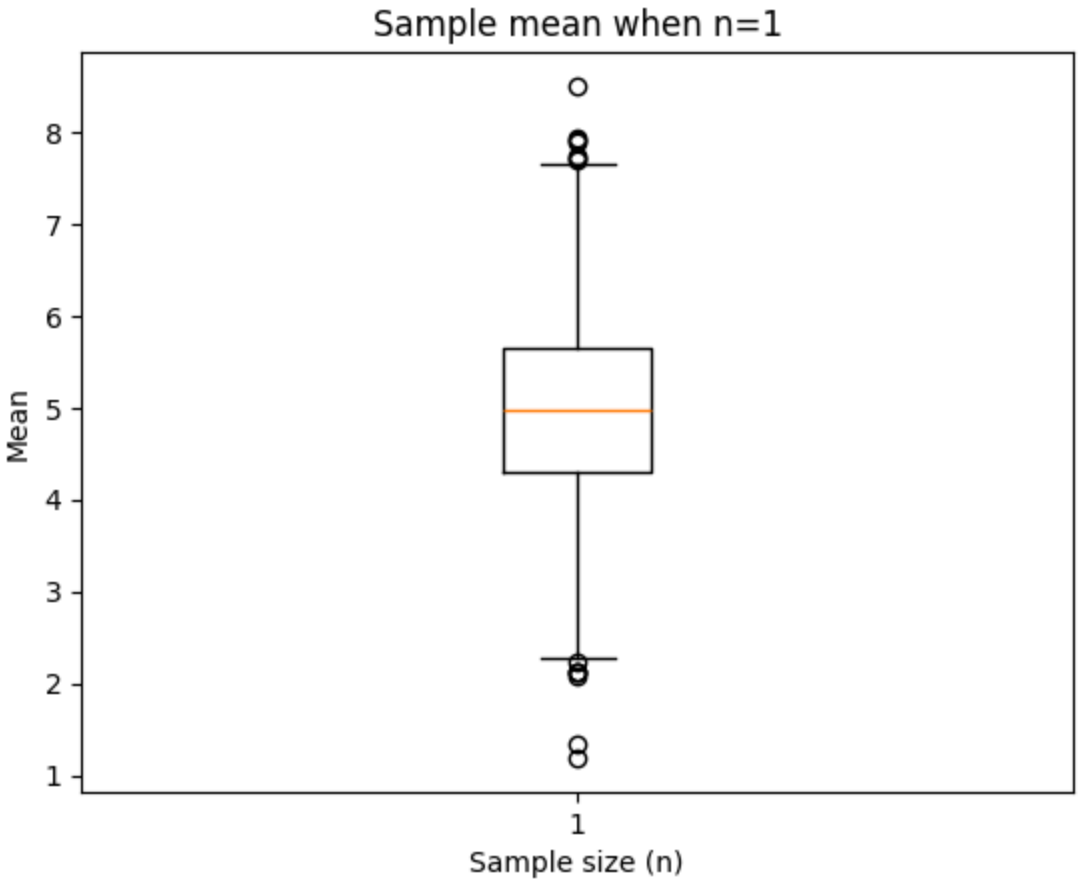
1b. Plot the boxplot for the sample means for all values n. Using words, interpret what the boxplot view of the 2000 trials for n=1 means and what the trends in the boxplot demonstrate compared to the plot in 1a. What information do you gain or lose in the two different plotting schemes?

We loose information on the tailedness of the dataset. Compared to 1a, we see outliers and there is no general trend or relationship that is observed like in the previous plot. The trends here demonstrate that there center of mass is around 5 with outliers at >5.4 and <4.6. This reflects the uncertainty in estimating the population mean with fewer samples. Larger sample sizes lead to more consistent estimates of the true population mean. The boxplot also reveals that the standard deviation of the sample means increases as n decreases.

In [86]:
```
# Answer to 1b here
# Hint: You'll want to look at plt.boxplot(sample_mean), where sample_mean is a 2D array

# Parameters
mu = 5
std = 1
n_trials = 2000
n = 1

#Test
sample_data = np.random.normal(mu, std, size=(n, 2000))
sample_mean = np.mean(sample_data, axis=0)
assert sample_mean.shape == (2000,)


plt.boxplot(sample_mean)
plt.xlabel("Sample size (n)")
```

```
plt.ylabel("Mean")
plt.title("Sample mean when n=1")
```

Out[86]: `Text(0.5, 1.0, 'Sample mean when n=1')`



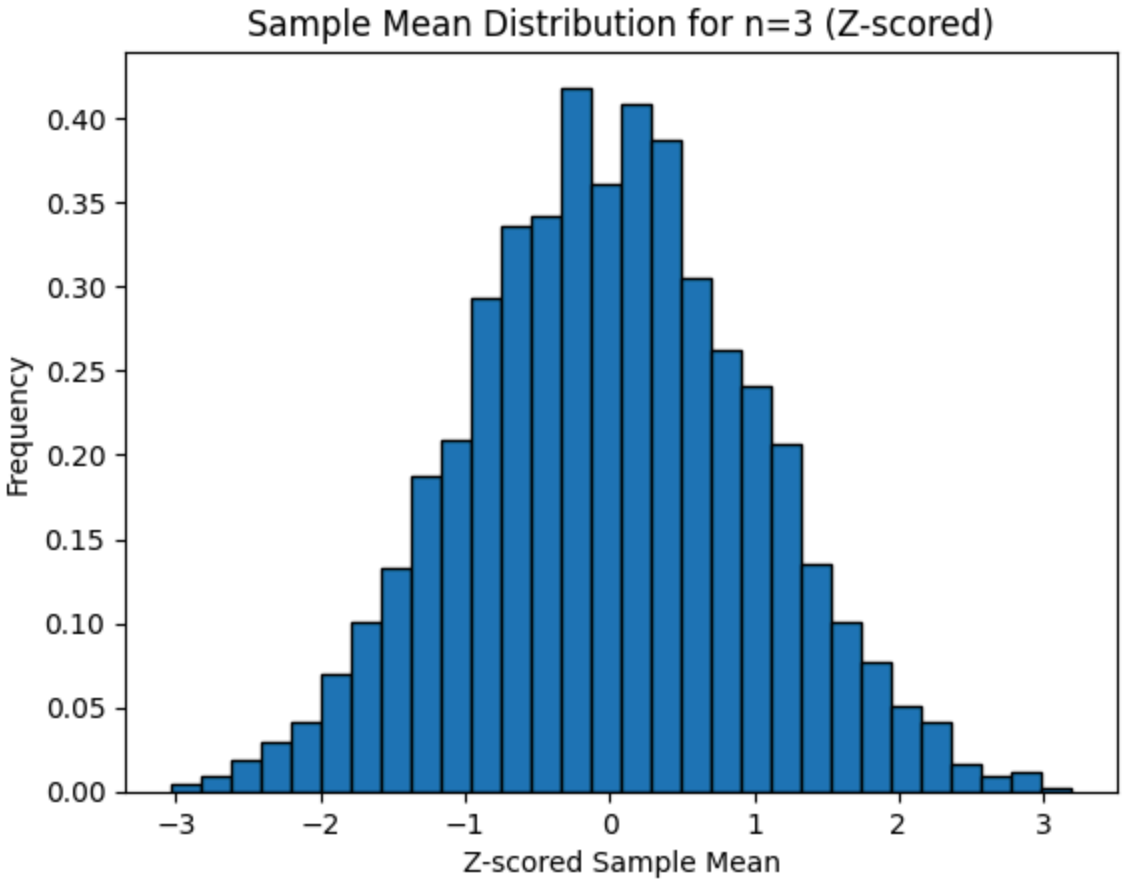
1c. For n=3, plot the histogram of the mean for the 2000 trials. Use the Kolmogorov-Smirnov test to see if this sample distribution is normal. (Hint: You may need to translate this to the *standard* normal distribution.) Report the sample mean and sample standard deviation, the p-value from the test, and whether you would reject the null hypothesis.

Given the p-value (greater than 0.05), we fail to reject the null hypothesis. This suggests that the distribution of sample means for n = 3 does not significantly deviate from normality. Hence, the sample distribution appears to follow a normal distribution.

In [34]:
```
# Answer to 1c here
# stat, pvalue = kstest(zscore_sample_means, 'norm') will return the ks statistic and p-
# for the KS test against the standard normal distribution.

# Parameters
mu = 5
std = 1
n_trials = 2000
n = 3

# Test
sample_data = np.random.normal(mu, std, size=(n, n_trials))
sample_mean = np.mean(sample_data, axis=0)
stat, pvalue = kstest(zscore(sample_mean), "norm")

# Result
```

```
print(f"KS test statistic: {stat}")
print(f"p-value: {pvalue}")
print(f"Sample standard deviation: {np.std(sample_mean)}")
print(f"Mean of z-scored sample means: {np.mean(zscore(sample_mean))}")

# Plot
plt.hist(zscore(sample_mean), bins=30, edgecolor='black', density=True)
plt.xlabel("Z-score")
plt.ylabel("Frequency")
plt.title("Sample mean distribution for n=3")
plt.show()
```

```
KS test statistic: 0.00930605561229525
p-value: 0.9945530122806998
Sample standard deviation: 0.58081721851109
Mean of z-scored sample means: -8.917311333789257e-16
```



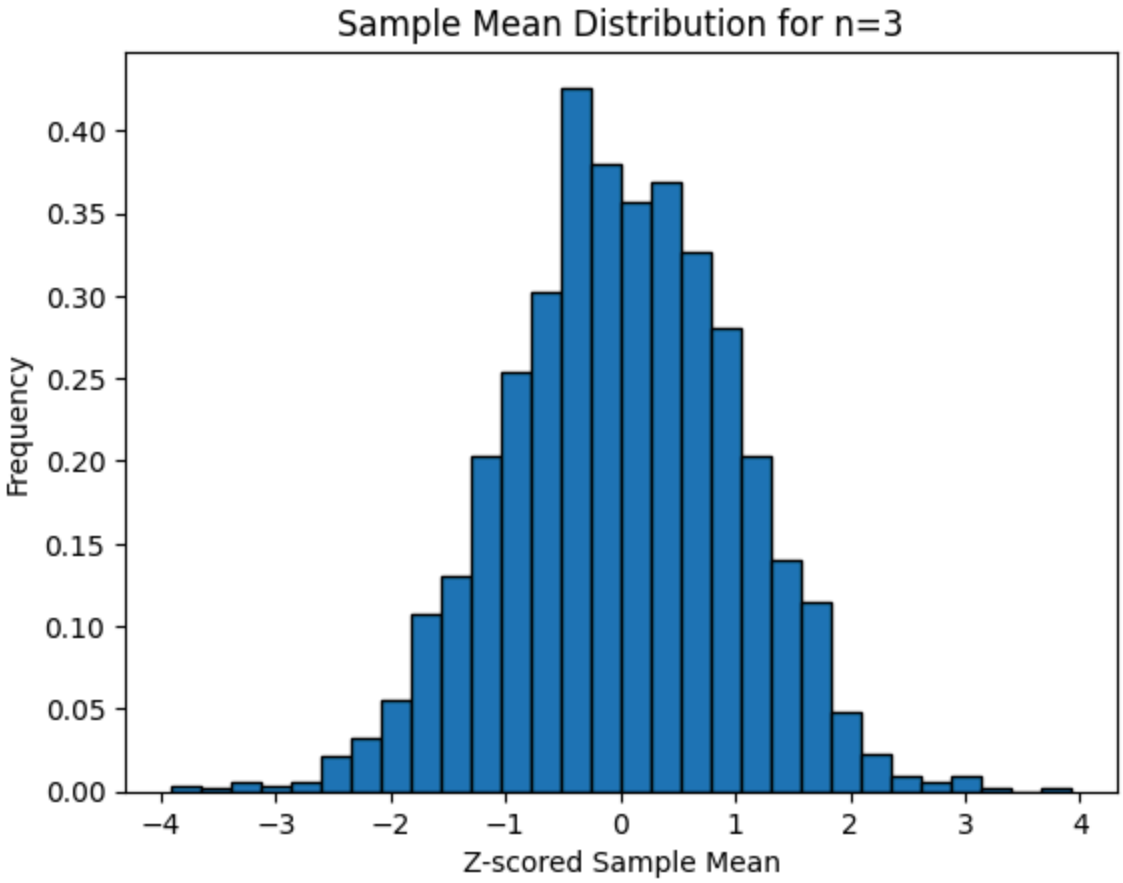
1d. Repeat 1c but for n=20. What changes when the number of samples increases?

The standard deviation decreases, the p-value slightly increases showing that with more samples, the sample mean gets closer to that of a normal distribution.

In [36]:
```
# Answer to 1d here

# Parameters
mu = 5
std = 1
n_trials = 2000
n = 20

# Test
sample_data = np.random.normal(mu, std, size=(n, n_trials))
```

```
sample_mean = np.mean(sample_data, axis=0)
stat, pvalue = kstest(zscore(sample_mean), "norm")

# Result
print(f"KS test statistic: {stat}")
print(f"p-value: {pvalue}")
print(f"Sample standard deviation: {np.std(sample_mean)}")
print(f"Mean of z-scored sample means: {np.mean(zscore(sample_mean))}")

# Plot
plt.hist(zscore(sample_mean), bins=30, edgecolor='black', density=True)
plt.xlabel("Z-scored Sample Mean")
plt.ylabel("Frequency")
plt.title("Sample Mean Distribution for n=3")
plt.show()
```

```
KS test statistic: 0.010975258392602438
p-value: 0.9673610245353964
Sample standard deviation: 0.22582859960431806
Mean of z-scored sample means: 7.931433287922118e-16
```



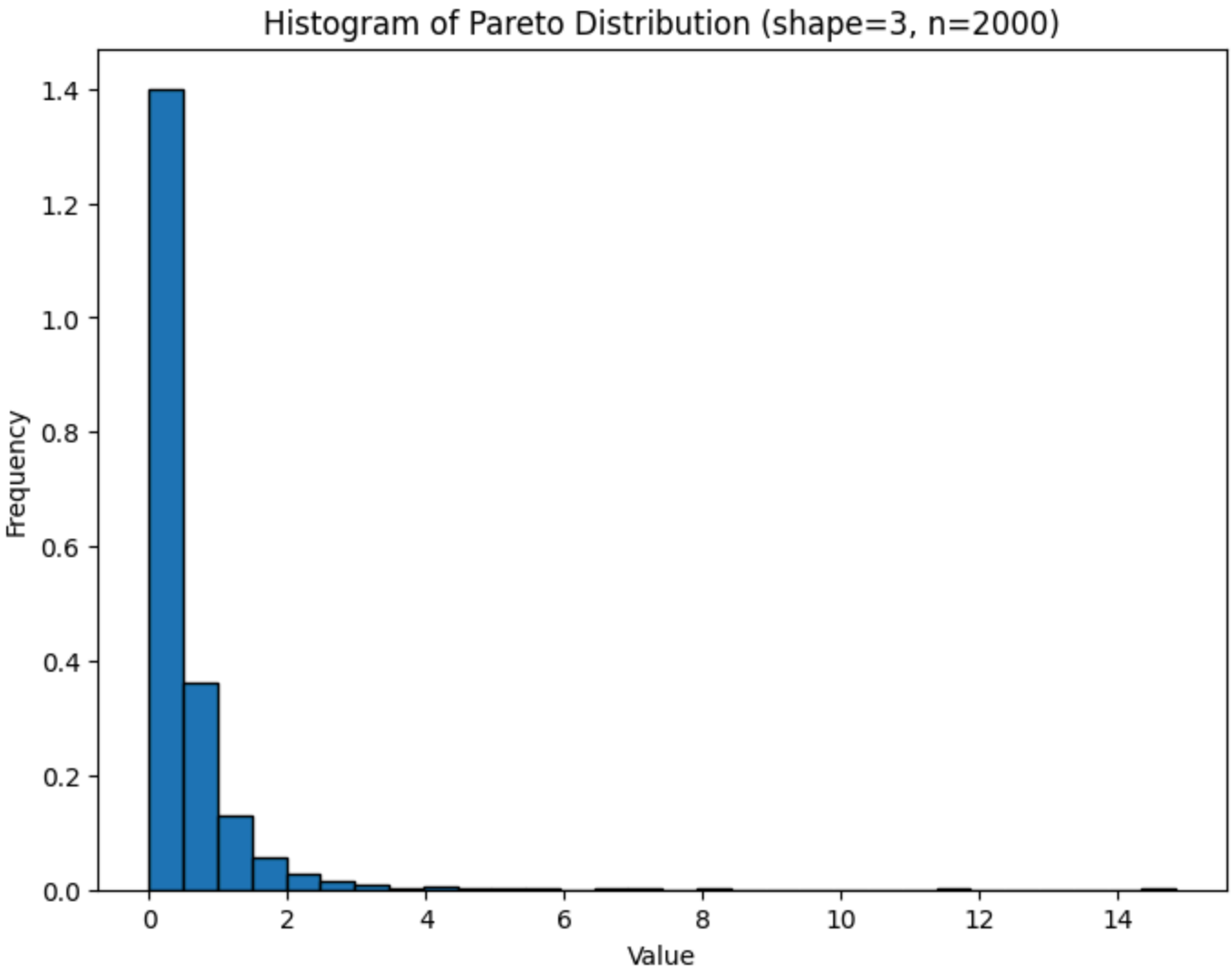
(2) Now we will explore sampling from an alternate distribution type.

2a. Sample the Pareto distribution (`np.random.pareto`) with parameter shape = 3, 2000 times. Plot the histogram of these values. Describe the shape of this histogram in words. Is it anything like the normal distribution?

The Pareto distribution looks looks like a positively skewed distribution with high sparcity. The center of mass lies towards the left which again contributes to most of the kurtosis. It is very different compared

to the normal distribution with different moments.

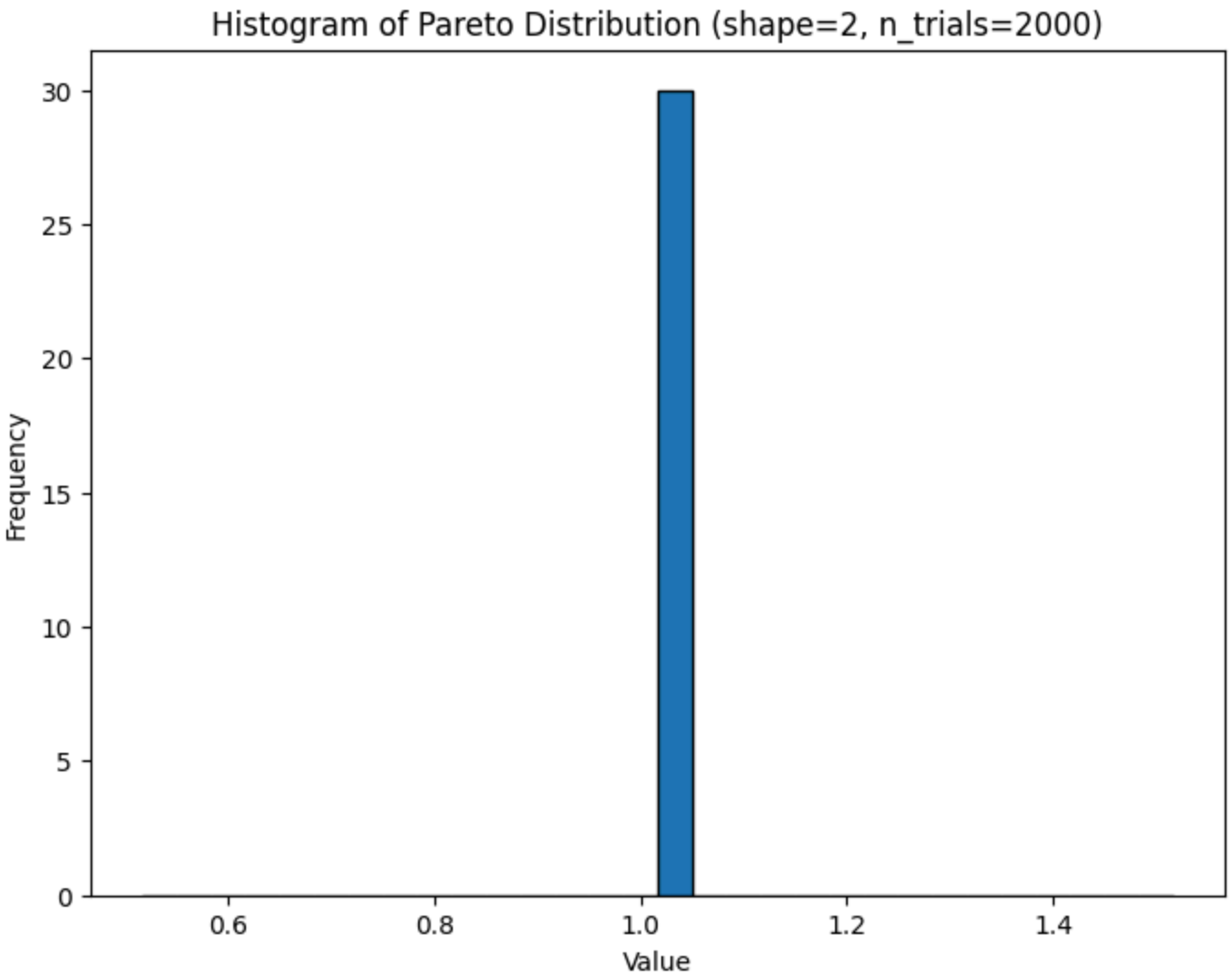
In [42]:
```python
# Answer 2a here
# plt.hist(x, bins=30) will make a histogram with 30 bins

# Parameters
shape = 3
n_trials = 2000

# Generate samples from Pareto distribution
sample_data = np.random.pareto(shape, n_trials)

# Calculate sample means for each trial
sample_mean = np.mean(sample_data, axis=0)

# Plot histogram of Pareto samples
plt.figure(figsize=(8, 6))
plt.hist(sample_data, bins=30, edgecolor='black', density=True)
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram of Pareto Distribution (shape=3, n_trials=2000)")
plt.show()
```



2b. What do you expect the sample distribution of the mean to look like for n=2. Just like a Pareto distribution, normal distribution, or something else? You can make a plot if you would like, but don't have to.

The sample distribution supposedly has to look like a Pareto distribution with the small sample size of n=2. As we increase the sample size and take the mean of the samples, the distribution of those means will tend to approach a normal distribution (mean = 0), regardless of the original Pareto distribution.

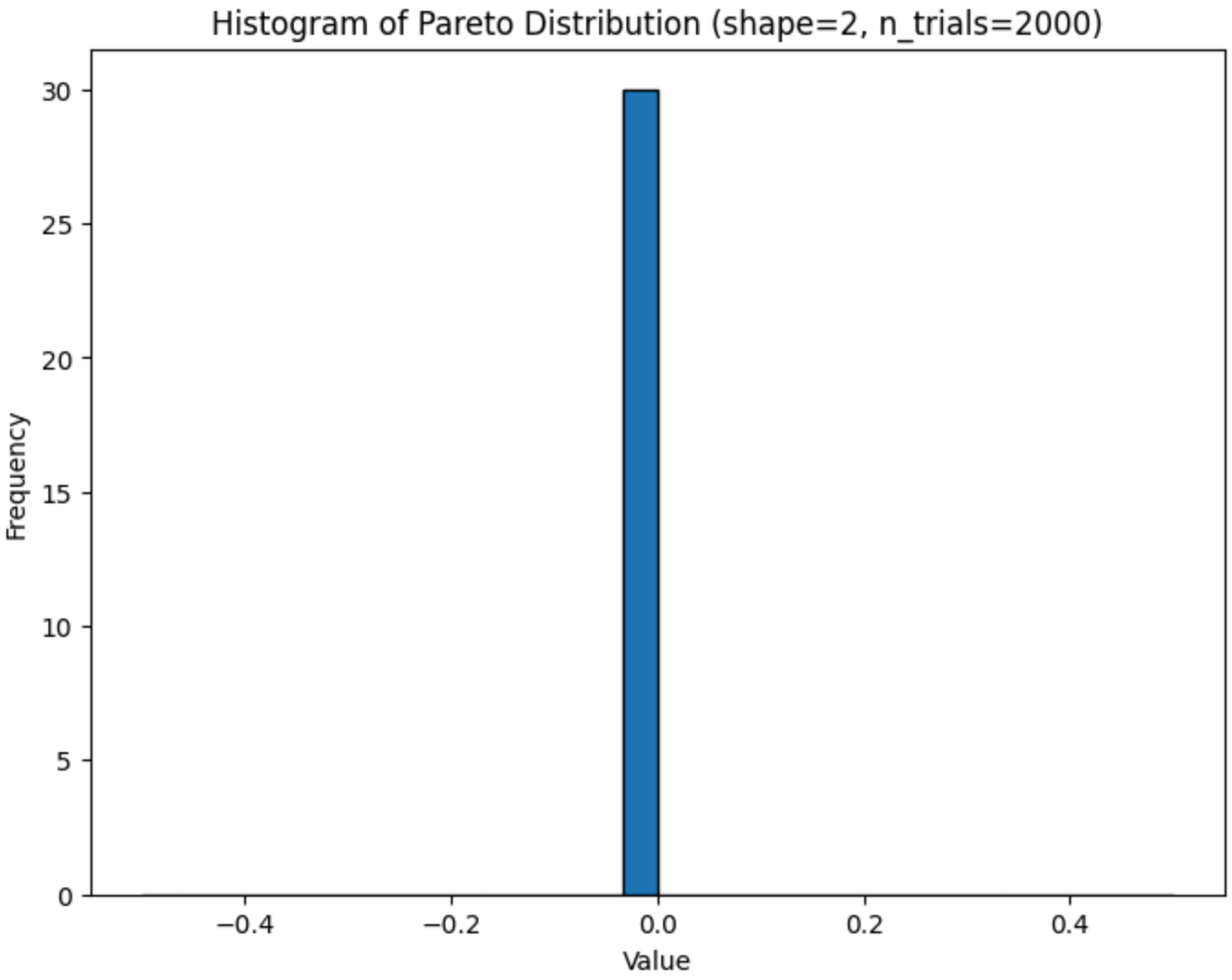
In [6]:
```
# Parameters
shape = 2
n_trials = 2000

# Generate samples from Pareto distribution
sample_data = np.random.pareto(shape, n_trials)

# Calculate sample means for each trial
sample_mean = np.mean(sample_data, axis=0)

# Plot histogram of Pareto samples
plt.figure(figsize=(8, 6))
plt.hist(sample_mean, bins=30, edgecolor='black', density=True)
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram of Pareto Distribution (shape=2, n_trials=2000)")
plt.show()
```



2c. How about for n=1,000,000?

For a large n such as 1,000,000, the distribution of the sample means will be nearly normal (mean = 0). Sample means will be centered around the mean of the standard normal distribution smaller variance.

In [4]:
```
# Parameters
shape = 1000000
n_trials = 2000

# Generate samples from Pareto distribution
sample_data = np.random.pareto(shape, n_trials)

# Calculate sample means for each trial
sample_mean = np.mean(sample_data, axis=0)

# Plot histogram of Pareto samples
plt.figure(figsize=(8, 6))
plt.hist(sample_mean, bins=30, edgecolor='black', density=True)
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram of Pareto Distribution (shape=2, n_trials=2000)")
plt.show()
```



(3) Differential expression. In this problem you will use the two-sample t-test to explore what differential hypothesis testing

looks like in known standards and how multiple hypothesis correction effects the number of false positives and negatives from these tests.

- Distribution 1, normal with mu=1, sigma=1
- Distribution 2, normal with mu=3, sigma=1

In [20]:
```
# Here is a function where you can input the mean and stdev for each distribution
# and then get back the number of positive outcomes from the two-sample t-test
# (all other tests were negative outcomes)

# n: the sample size
# n_tests: number of tests
# alpha: the significance cutoff

def two_sample_ttest(n: int, n_tests: int, mu1: float, mu2: float, sigma: float, alpha:
    # This loads all the trials in one big array at once, hence no loop
    dist1_sample = np.random.normal(mu1, sigma, size=(n, n_tests))
    dist2_sample = np.random.normal(mu2, sigma, size=(n, n_tests))
    _, p_values = ttest_ind(dist1_sample, dist2_sample)

    return np.sum(p_values <= alpha)

# Example:

fps = two_sample_ttest(n=1000, n_tests=100, mu1=1.0, mu2=3.0, sigma=1, alpha=0.05)
print(fps)
```

```
10
```

3a. False Negative: Using n=3, perform 100 comparisons of distribution 1 versus distribution 2 with an alpha=0.05. Anytime you fail to reject the hypothesis it is a false negative. Why is this a false negative? Report the number of false negatives from your 100 tests.

They are false negatives because we fail to reject the null that the sample means of the the two distributions are the same (mu1=mu2). We know that the the means of the distribution are different (mu1=1.0, mu2=3.0), however, the variance for the two distributions remains the same. This may allow for some overlap between the distributions resulting in false negatives. Also, the small sample size (n=3) makes it difficult to detect the true difference.

In [58]:
```
fns = 100 - two_sample_ttest(n=3, n_tests=100, mu1=1.0, mu2=3.0, sigma=1, alpha=0.05)
print(f"False negatives in 100 tests: {fps}")
```

```
False negatives in 100 tests: 3
```

3b. False Positives: Using n=3, perform 100 comparisons of distribution 1 versus distribution 1 with an alpha=0.05. Anytime you reject the hypothesis this is a false positive. Why is this a false positive? Report the number of false positives from your 100 tests.

In [59]:
```
# Answer to 3b here.

fps = two_sample_ttest(n=3, n_tests=100, mu1=1.0, mu2=1.0, sigma=1, alpha=0.05)
print(f"False positives in 100 tests: {fps}")
```

```
False positives in 100 tests: 3
```

In this case we care comparing the exact same distribution to itself (mu1=mu1). Here we fail to reject our null in some cases even though they are the same distributions which results in a few false positives. Any rejection of the null hypothesis is incorrect and results in a false positive.

3c. Repeat 3b but 2000 times. What is the number of false positives? Predict the number of false positives you would get if you compared samples from the same distribution 10,000 times and explain why.

In [55]:
```
# Answer to 3c
fps = two_sample_ttest(n=3, n_tests=2000, mu1=1.0, mu2=1.0, sigma=1, alpha=0.05)
print(f"False positives in 2000 tests: {fps}")
```

```
False positives in 2000 tests: 89
```

The alpha value represents the probability of type 1 (false positives) in the data. With alpha set at 0.05, we are expecting to see about 100 false positives in 2000 tests (0.05 * 2000 = 100). With 10000 tests, we would seen an increase in false positives by a magnitude of 5 (0.05 * 10000 = 500).

3d. Now sweep n from 3 to 30 and report the number of false positives and false negatives for each n when you run 100 comparisons. (Provide this in a table format). Please explain the trend you see and interpret its meaning.

In [62]:
```
# Answer to 3d
data = []
for n in range(3, 31):
    fps = two_sample_ttest(n=n, n_tests=100, mu1=1.0, mu2=1.0, sigma=1, alpha=0.05)
    fns = 100 - two_sample_ttest(n=n, n_tests=100, mu1=1.0, mu2=3.0, sigma=1, alpha=0.05
    data.append({"n": n, "False positives": fps, "False negatives": fns})

df = pd.DataFrame(data)

print(df)
```

	n	False positives	False negatives
0	3	1	62
1	4	6	27
2	5	4	13
3	6	4	10
4	7	2	9
5	8	4	2
6	9	6	1
7	10	7	1
8	11	4	0
9	12	4	0
10	13	5	0
11	14	7	0
12	15	7	0
13	16	8	0
14	17	3	0
15	18	8	0
16	19	7	0
17	20	8	0
18	21	7	0
19	22	1	0
20	23	7	0
21	24	5	0
22	25	6	0
23	26	6	0
24	27	4	0
25	28	4	0
26	29	5	0
27	30	7	0

The number of false negatives decrease as n increases, and the number of false positives seems to be stable. In terms of type I error, our alpha threshold set at 5% (alpha = 0.05) seems to control for false positive rate with increasing number of samples. There is a dramatic decrease in false negative rate as sample size increases. This shows that with small sample size we fail to detect true differences and with increasing sample size, we reduce the amount of type II errors with the ability to detect true difference with higher sample size and statistical power.

3e. For n=3, suggest how the number of false negatives changes according to sigma for the two distributions and test this. Report your new values and sigma and the number of false negatives in 100 tests.

In [72]:
```
# Answer to 3e
data = []
for sigma in [0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 5.0]:
    fns = 100 - two_sample_ttest(n=3, n_tests=100, mu1=1.0, mu2=3.0, sigma=sigma, alpha=
    data.append({"sigma": sigma, "False negatives in 100 tests": fns})

df = pd.DataFrame(data)

print(df)
```

	sigma	False negatives in 100 tests
0	0.1	0
1	0.5	3
2	1.0	57
3	1.5	73
4	2.0	84
5	2.5	87
6	3.0	86
7	5.0	93

The number of false negatives increase with increasing sigma. For a small sigma like 0.5, we encounter less false negatives because the distributions do not overlap as much. With increasing sigma values, there is more overlap between distributions due the increase in the magnitude of deviation form the mean.

3f. Lastly, perform 3d for p < 0.01 instead of p < 0.05. How does this influence the rate of false positives and negatives? How might you use this when performing many tests?

In [73]:
```
# Answer to 3f
data = []
for n in range(3, 31):
    fps = two_sample_ttest(n=n, n_tests=100, mu1=1.0, mu2=1.0, sigma=1, alpha=0.01)
    fns = 100 - two_sample_ttest(n=n, n_tests=100, mu1=1.0, mu2=3.0, sigma=1, alpha=0.01
    data.append({"n": n, "False positives": fps, "False negatives": fns})

df = pd.DataFrame(data)

print(df)
```

	n	False positives	False negatives
0	3	3	93
1	4	2	67
2	5	3	50
3	6	0	38
4	7	0	27
5	8	0	18
6	9	2	9
7	10	2	10
8	11	0	6
9	12	0	2
10	13	0	1
11	14	1	2
12	15	2	1
13	16	1	1
14	17	0	1
15	18	3	0
16	19	2	0
17	20	0	0
18	21	0	0
19	22	3	0
20	23	2	0
21	24	1	0
22	25	0	0
23	26	1	0
24	27	1	0
25	28	0	0
26	29	0	0
27	30	2	0

By setting alpha to 0.01, we are only allowing roughly 1% of false positive for each test we run. This makes the analysis more stringent allowing for very few false positives. But on the contrary, we observe more false negatives with sample sizes less than 18. This is because a stricter significance threshold makes it harder to detect true effects. The tradeoff between sample size the false negative rate is important to consider to be able to detect true differences in the data.

(4) Power analysis

Now that we've observed the trends above, we have all the tools to talk about power analysis. Power analysis is an essential step when designing an experiment, and asks *assuming we should be rejecting the null hypothesis, what is the probability we will do so*. The power of an experiment is equal to 1 minus the false negative rate. A common choice for the power of an experiment (like a p-value cutoff of 0.05) is 0.8.

4a. Power analysis is often used to determine the necessary N of an experiment. Why is power used as opposed to the false positive rate?

power = 1 - false negative rate (fnr)

Power is used to determine the necessary N for an experiment opposed to false positive rate because we know that we can control for the number of false positives using the alpha parameter which refers to the probability of incorrectly rejecting the null hypothesis when it is actually true. However, disregarding the false negative rate can deter us from **detecting true effects** and therefore it is crucial that we are detecting true differences and power can help measure this. A smaller N leads to more variability, making it harder to detect true effects, thus reducing power. A larger N leads to less variability, making it easier to detect true effects with increasing power.

4b. Determine the necessary N to acheive a power of 0.9 from the situation in (3).

In [84]:
```
# Answer.
# power = 1 - fnr
power = 0.9
fnr = 0.1

data = []
for n in range(0, 15):
    fns = 100 - two_sample_ttest(n=n, n_tests=100, mu1=1.0, mu2=3.0, sigma=1, alpha=0.01
    fnr = fns/100
    data.append({"n": n,  "False negatives": fns, "False negative rate": fnr, "Power": 1

df = pd.DataFrame(data)

print(df)
print(f"Necessary sample size for 90% power: {df[df['Power'] >= 0.9].iloc[0]['n']}")
```

```
     n  False negatives  False negative rate  Power
0    0              100                 1.00   0.00
1    1              100                 1.00   0.00
2    2               96                 0.96   0.04
3    3               83                 0.83   0.17
4    4               68                 0.68   0.32
5    5               55                 0.55   0.45
6    6               32                 0.32   0.68
7    7               22                 0.22   0.78
8    8               18                 0.18   0.82
9    9                9                 0.09   0.91
10  10                3                 0.03   0.97
11  11                2                 0.02   0.98
12  12                4                 0.04   0.96
13  13                0                 0.00   1.00
14  14                0                 0.00   1.00
Necessary sample size for 90% power: 9.0
```

```
/tmp/ipykernel_2356/1256792429.py:13: SmallSampleWarning: All axis-slices of one or more
sample arguments are too small; all elements of returned arrays will be NaN. See document
ation for sample size requirements.
  _, p_values = ttest_ind(dist1_sample, dist2_sample)
/home/codespace/.local/lib/python3.12/site-packages/scipy/stats/_stats_py.py:6558: Runtim
eWarning: invalid value encountered in divide
  svar = ((n1 - 1) * v1 + (n2 - 1) * v2) / df
```

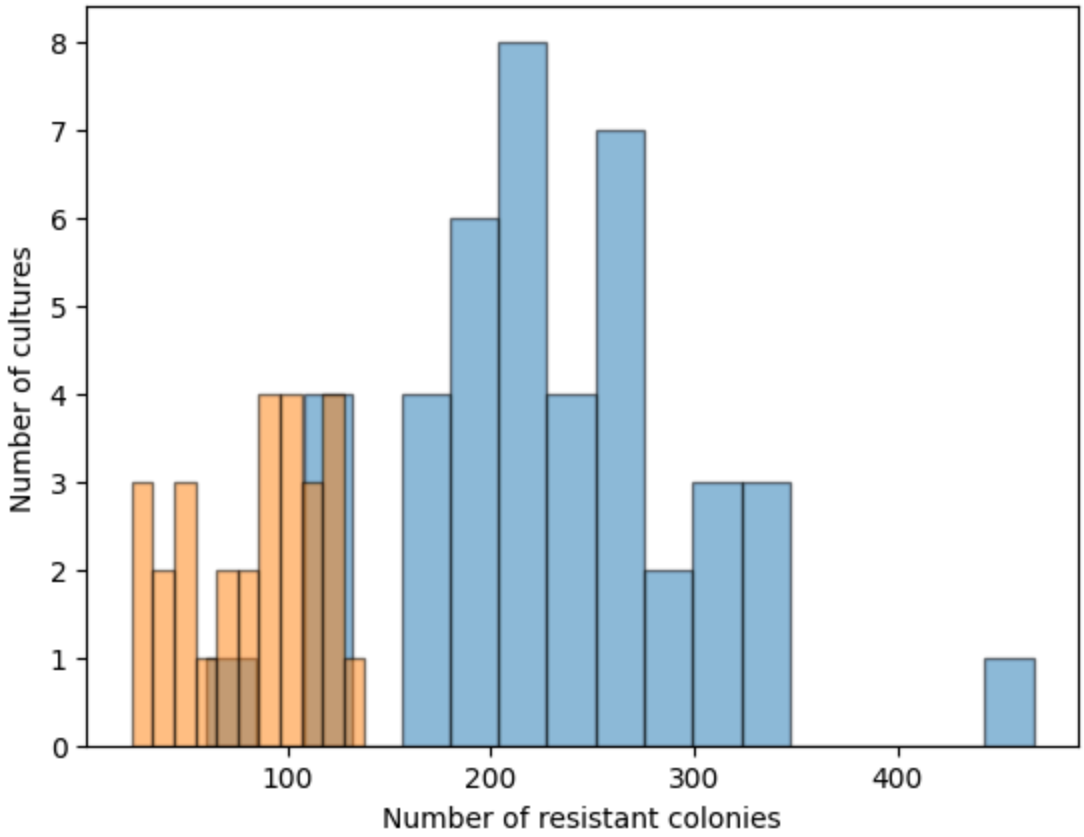
(5) Shaffer et al

In this excercise we're going to explore some basic concepts of statistics, and use them to build up to some more advanced ideas. To examine these ideas we're going to consider a classic of molecular biology—the Luria-Delbrück experiment. The data we'll use is from Shaffer et al.

In [9]:
```
repOne = np.array([259, 213, 182, 167, 254, 221, 236, 168, 190, 262, 285, 158, 240, 187,
repTwo = np.array([28, 72, 53, 103, 46, 90, 105, 78, 86, 127, 30, 52, 105, 111, 88, 35,
```

In [10]:
```
plt.hist(repOne, bins=17, edgecolor='black', alpha=0.5, label='Rep One')
plt.hist(repTwo, bins=11, edgecolor='black', alpha=0.5, label='Rep Two')
plt.xlabel("Number of resistant colonies")
plt.ylabel("Number of cultures")
```

Out[10]:
```
Text(0, 0.5, 'Number of cultures')
```



5a. First, we need to build up a distribution of outcomes for what an experiment would look like if it followed the Luria-Delbruck process.

Assumptions:

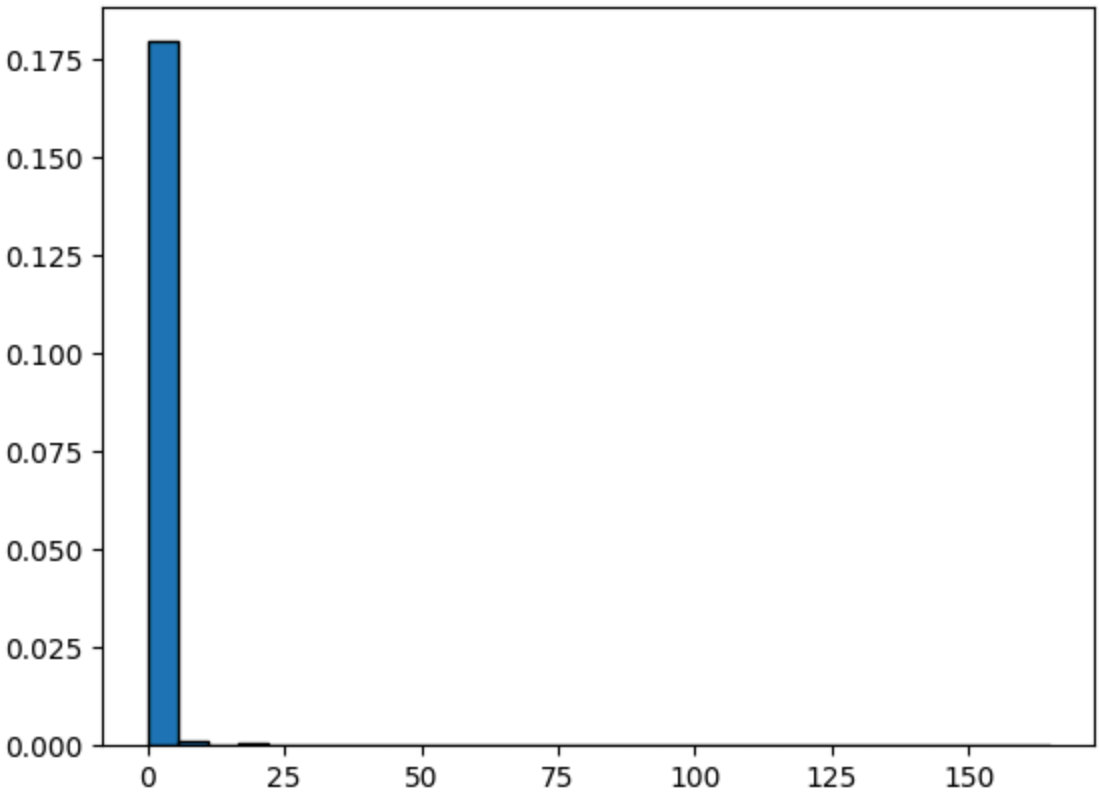
- Mutations occur at a constant rate per cell per generation.
- Mutations happen at random times during the cell division process.
- Once a cell mutates, it retains the ability to divide and grow like non-mutant cells.
- The population of non-mutant cells doubles in each generation.
- There is no selective pressure (e.g., antibiotics) until after the mutations have occurred.
- Once a cell mutates, it passes the mutation to all of its progeny.
- No resource limitation for growth.

Fill in the function below keeping track of normal and mutant cells. Then, make a second function, `CVofNRuns`, that runs the experiment 3000 times. You can assume a culture size of 120000 cells, and mutation rate of 0.0001 per cell per generation. What does the distribution of outcomes look like?

In [11]:
```
# Runs the simulation a bunch of times, and looks for how often the fano (cv/mean) comes

def simLuriaDelbruck(cultureSize, mutationRate):
    nCells, nMuts = 1, 0  # Start with 1 non-resistant cell

    for _ in range(int(np.floor(np.log2(cultureSize)))):  # num of gens
        nCells = nCells*2      # Double the number of cells, simulating division
        newMuts = np.random.poisson(nCells * mutationRate)  # de novo
        nMuts = 2*nMuts + newMuts # Previous mutants divide and add
        nCells =  nCells - newMuts  # Non-resistant pop goes down by newMuts

    return nMuts
```

```
def CVoNRuns(nRuns, cultureSize, mutationRate, print=False):
    numCellsPositivePerCulture = np.zeros(nRuns)
    cv = np.zeros(nRuns)
    fano = np.zeros(nRuns)

    for i in range(nRuns):
        numCellsPositivePerCulture[i] = simLuriaDelbruck(cultureSize, mutationRate)
        cv[i] = np.std(numCellsPositivePerCulture) / np.mean(numCellsPositivePerCulture)
        fano[i] = np.var(numCellsPositivePerCulture) / np.mean(numCellsPositivePerCultur

    return numCellsPositivePerCulture, cv, fano


numCellsPositivePerCulture, cv, fano = CVoNRuns(nRuns=3000, cultureSize=120000, mutation

plt.hist(numCellsPositivePerCulture/np.mean(numCellsPositivePerCulture), bins=30, edgeco
```

Out[11]:
```
(array([1.79477337e-01, 1.39647454e-03, 1.82148853e-04, 4.85730275e-04,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 2.42865138e-04,
        0.00000000e+00, 6.07162844e-05, 0.00000000e+00, 0.00000000e+00,
        0.00000000e+00, 0.00000000e+00, 1.82148853e-04, 0.00000000e+00,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
        0.00000000e+00, 1.21432569e-04]),
 array([2.92091145e-02, 5.51922447e+00, 1.10092398e+01, 1.64992552e+01,
        2.19892705e+01, 2.74792859e+01, 3.29693012e+01, 3.84593166e+01,
        4.39493319e+01, 4.94393473e+01, 5.49293626e+01, 6.04193780e+01,
        6.59093933e+01, 7.13994087e+01, 7.68894240e+01, 8.23794394e+01,
        8.78694547e+01, 9.33594701e+01, 9.88494854e+01, 1.04339501e+02,
        1.09829516e+02, 1.15319531e+02, 1.20809547e+02, 1.26299562e+02,
        1.31789578e+02, 1.37279593e+02, 1.42769608e+02, 1.48259624e+02,
        1.53749639e+02, 1.59239654e+02, 1.64729670e+02]),
 <BarContainer object of 30 artists>)
```



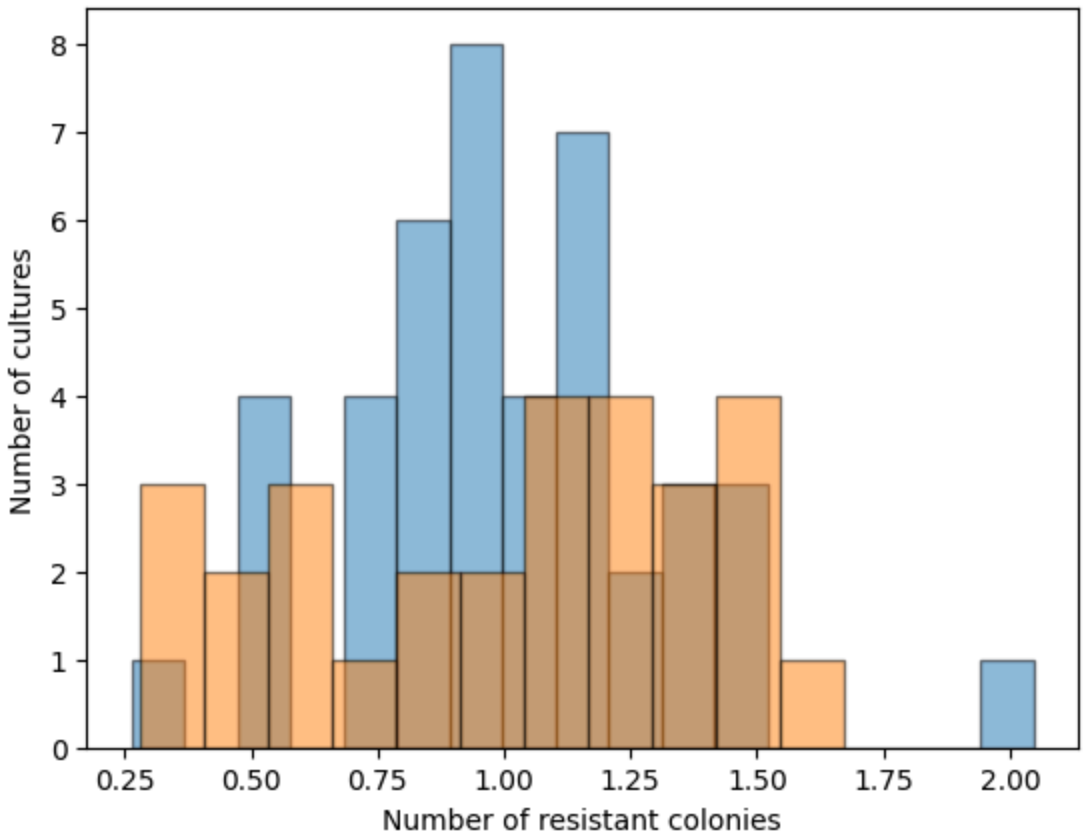
The outcome distribution looks sparse in nature, showing that mutaitons occur randomly. The distribution resembles a poisson distribution.

5b. Compare the distribution of outcomes between the two replicates of the experiment using the 2-sample KS test. Are they consistent with one another?

Hint: Each experiment varies slightly in the amount of time it was run. The absolute values of the numbers doesn't matter, so much as the variation of them. You'll need to correct for this by dividing by the mean of the results.

In [12]:
```
# Answer to 5b
# e.g. repOneCorrected = repOne / np.mean(repOne)

repOneCorrected = repOne / np.mean(repOne)
repTwoCorrected = repTwo / np.mean(repTwo)

plt.hist(repOneCorrected, bins=17, edgecolor='black', alpha=0.5, label='Rep One')
plt.hist(repTwoCorrected, bins=11, edgecolor='black', alpha=0.5, label='Rep Two')
plt.xlabel("Number of resistant colonies")
plt.ylabel("Number of cultures")

stat, pvalue = ks_2samp(repOneCorrected, repTwoCorrected)

# Result
print(f"KS test statistic: {stat}")
print(f"p-value: {pvalue}")
```

```
KS test statistic: 0.25100240577385724
p-value: 0.1784655548732054
```

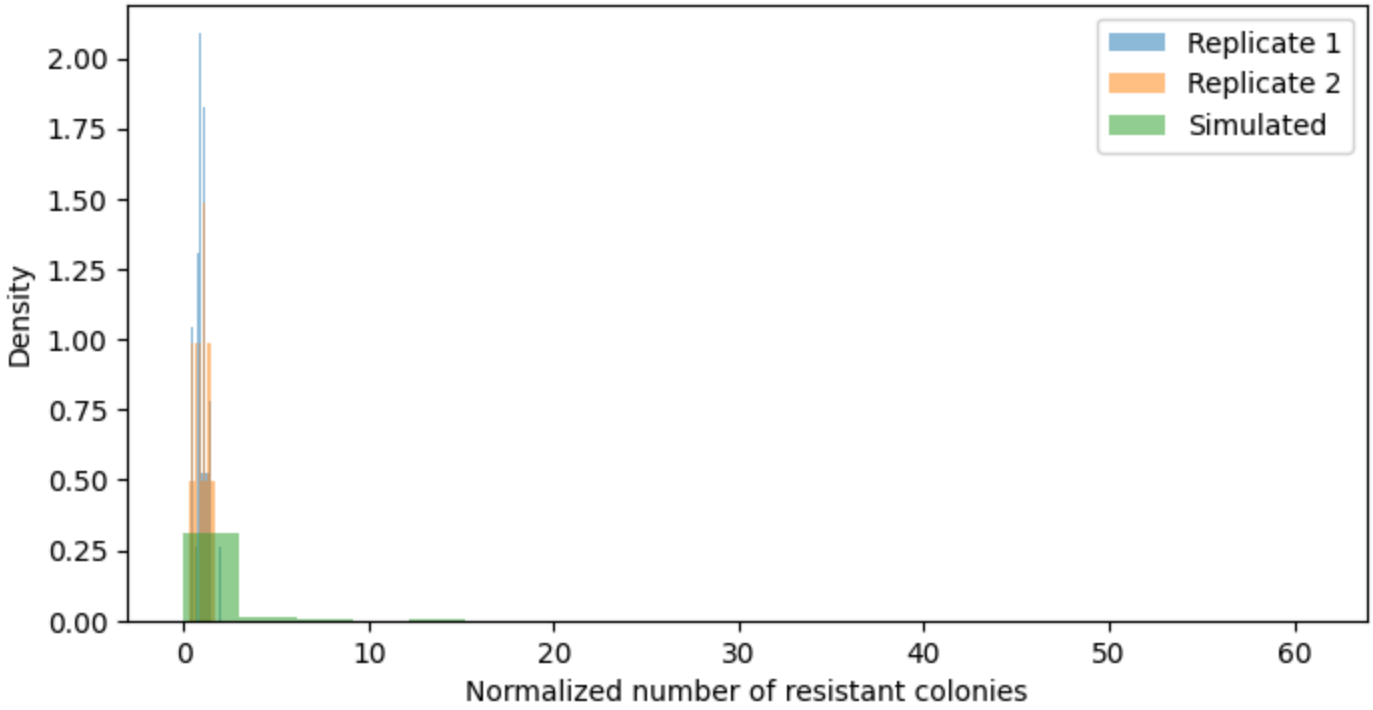


The p-value is > 0.05 therefore we fail to reject the null hypothesis, suggesting the distributions are not significantly different form each other.

5c. Compare the distribution of outcomes between the experiment and model. Are our results consistent with resistance arising through a Luria-Delbruck related process?

In [21]:
```
# Answer to 5c

# Replicate1 = 43 clones
# Replicate2 = 29 clones

# Simulation parameters
cultureSize = 1000 # Adjust based on your experimental setup
mutationRate =  0.001 # Initial guess, may need adjustment
nRuns = 10000

# Run simulations
simulated_data, sim_cv, sim_fano= CVoNRuns(nRuns, cultureSize, mutationRate)

# Calculate CV and Fano factor for experimental data
exp_cv1 = np.std(repOne) / np.mean(repOne)
exp_cv2 = np.std(repTwo) / np.mean(repTwo)

exp_fano1 = np.var(repOne) / np.mean(repOne)
exp_fano2 = np.var(repTwo) / np.mean(repTwo)

# Normalize data
exp_normalized1 = repOne / np.mean(repOne)
exp_normalized2 = repTwo / np.mean(repTwo)

sim_normalized = simulated_data / np.mean(simulated_data)

# KS test
ks_stat1, p_value1 = ks_2samp(exp_normalized1, sim_normalized)
ks_stat2, p_value2 = ks_2samp(exp_normalized2, sim_normalized)


# Plotting
plt.figure(figsize=(8, 4))
plt.hist(exp_normalized1, bins=20, alpha=0.5, label='Replicate 1', density=True)
plt.hist(exp_normalized2, bins=20, alpha=0.5, label='Replicate 2', density=True)
plt.hist(sim_normalized, bins=20, alpha=0.5, label='Simulated', density=True)
plt.xlabel('Normalized number of resistant colonies')
plt.ylabel('Density')
plt.legend()
plt.show()

# Print results
print(f"KS test statistic replicate 1: {ks_stat1:.4f}")
print(f"p-value replicate 1: {p_value1:.4f}")

print(f"KS test statistic replicate 2: {ks_stat2:.4f}")
print(f"p-value replicate 2: {p_value2:.4f}")

print(f"CV (Replicate 1): {exp_cv1:.4f}")
print(f"CV (Replicate 2): {exp_cv2:.4f}")
print(f"CV (Simulated): {np.mean(sim_cv):.4f}")
```

```
print(f"Fano factor (Replicate 1): {exp_fano1:.4f}")
print(f"Fano factor (Replicate 2): {exp_fano2:.4f}")
print(f"Fano factor (Simulated): {np.mean(sim_fano):.4f}")
```




```
KS test statistic replicate 1: 0.6504
p-value replicate 1: 0.0000
KS test statistic replicate 2: 0.5358
p-value replicate 2: 0.0000
CV (Replicate 1): 0.3184
CV (Replicate 2): 0.3968
CV (Simulated): 6.1405
Fano factor (Replicate 1): 23.1398
Fano factor (Replicate 2): 12.9808
Fano factor (Simulated): 47.2583
```

Our results for the simulated data represent a Poisson process. The coefficient of variation and the Fano factor differ significantly between the experimental data and the simulated data. Therefore, the experimental data are not consistent with the Luria and Delbrück process (Poisson process). The experimental data do not fit the heritable mutation model, but fits the transient mutation model proposed in the paper.

What does this tell us, biologically, about the process of developing drug resistance in the melanoma cells within this study? Describe the results using the concepts of the Luria-Delbruck experiment.

The Luria-Delbruck experiment provided strong evidence that mutations in bacteria occur randomly, rather than as a direct result of exposure to selective pressures such as viruses or antibiotics.

Biologically, this tells us whether the process of developing drug resistance in melanoma cells can be due to random and drug induced mutations which are transient in nature. This has important implications for understanding tumor evolution and designing more effective treatment strategies for melanoma.

The modeled distribution here does not match the experimental distributions. This may suggest that the mutations that lead to resistance may be caused due to the drug induced transient mutations and not due to randomly occuring mutations. In terms of the Luria-Delbruck experiment we do not see a lot of fluctuation described in terms of the coefficient of variance and the fano factors in the experimental data suggesting that they might be drug induced transient mutations and not random mutations.