Cameron Bryant
CS 458
Assignment 4

1)

a)

```
import hashlib

# My name
name = "Cameron Bryant"

# Compute SHA-256 hash
hash_value = hashlib.sha256(name.encode()).hexdigest()

# Print the hash value in hexadecimal
print(f"SHA-256 hash of '{name}': {hash_value}")
```

This implementation for question a uses the hashlib library to print the SHA-256 hash value of my name. During performance, my output was:

SHA-256 hash of 'Cameron Bryant':
271c30d4fd6d5197a3b4bc19ea92777bba0340dfe3b8185c3bde36ae7a4e4bc7

b)

For b, I automated this task further and on a larger scale than before. File “Assignment4b” can be run to display the testing. The code performs by:

1. Generating random inputs and compute their SHA-256 hashes.
2. Counting the number of 1 bits in each hash output.
3. Aggregating these counts into a histogram to analyze the distribution of 1 bits.

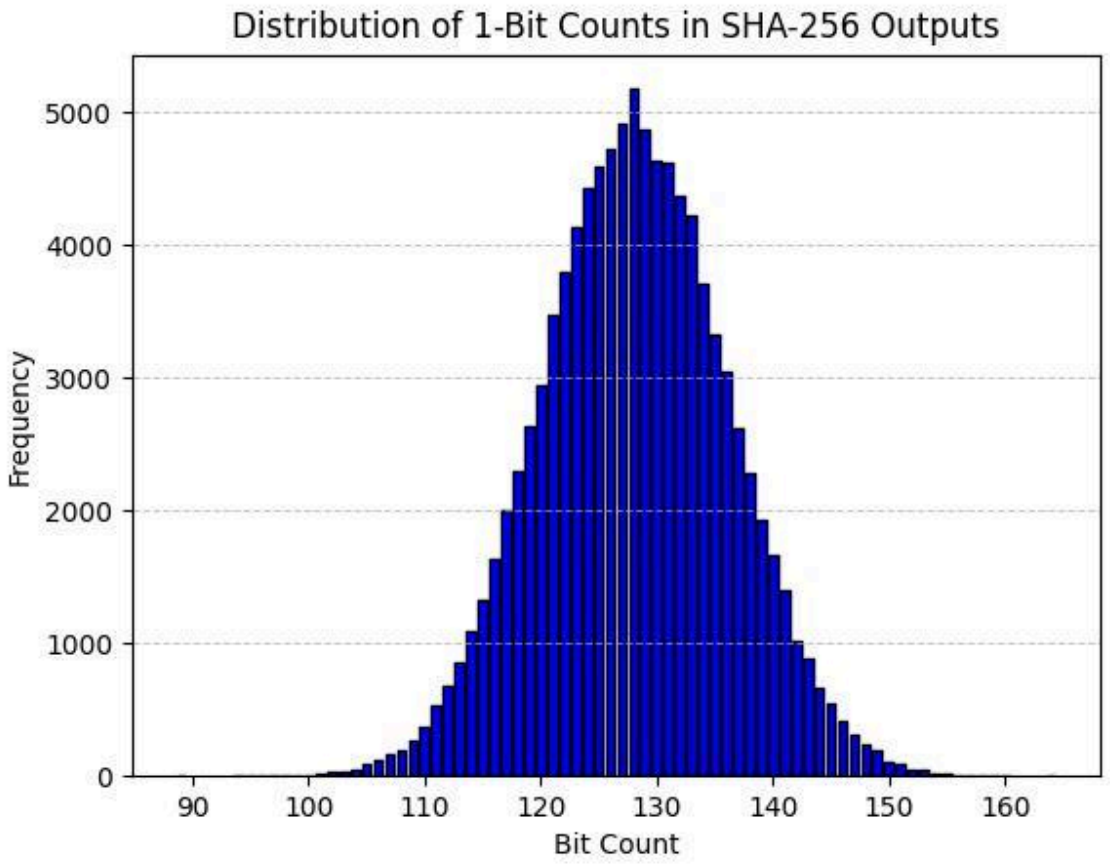
The histogram was visualized using a bar chart, confirming that the number of 1 bits follows a roughly binomial distribution.

Here are the results of when the dataset was set to hashes computed up 100000 times.

Histogram of 1-bit counts:
Bit Count: 89, Frequency: 1
Bit Count: 94, Frequency: 1
Bit Count: 95, Frequency: 1
Bit Count: 96, Frequency: 1
Bit Count: 97, Frequency: 4
Bit Count: 98, Frequency: 4
Bit Count: 99, Frequency: 7
Bit Count: 100, Frequency: 10
Bit Count: 101, Frequency: 17
Bit Count: 102, Frequency: 26
Bit Count: 103, Frequency: 35
Bit Count: 104, Frequency: 51
Bit Count: 105, Frequency: 87
Bit Count: 106, Frequency: 122
Bit Count: 107, Frequency: 162
Bit Count: 108, Frequency: 192
Bit Count: 109, Frequency: 271
Bit Count: 110, Frequency: 377
Bit Count: 111, Frequency: 538
Bit Count: 112, Frequency: 682
Bit Count: 113, Frequency: 854
Bit Count: 114, Frequency: 1097
Bit Count: 115, Frequency: 1321
Bit Count: 116, Frequency: 1632
Bit Count: 117, Frequency: 2003
Bit Count: 118, Frequency: 2291
Bit Count: 119, Frequency: 2641
Bit Count: 120, Frequency: 2944
Bit Count: 121, Frequency: 3482
Bit Count: 122, Frequency: 3804
Bit Count: 123, Frequency: 4139
Bit Count: 124, Frequency: 4436
Bit Count: 125, Frequency: 4600
Bit Count: 126, Frequency: 4723
Bit Count: 127, Frequency: 4923
Bit Count: 128, Frequency: 5179
Bit Count: 129, Frequency: 4871
Bit Count: 130, Frequency: 4632
Bit Count: 131, Frequency: 4628

Bit Count: 132, Frequency: 4367
Bit Count: 133, Frequency: 4226
Bit Count: 134, Frequency: 3717
Bit Count: 135, Frequency: 3323
Bit Count: 136, Frequency: 3051
Bit Count: 137, Frequency: 2616
Bit Count: 138, Frequency: 2281
Bit Count: 139, Frequency: 1936
Bit Count: 140, Frequency: 1667
Bit Count: 141, Frequency: 1399
Bit Count: 142, Frequency: 1016
Bit Count: 143, Frequency: 887
Bit Count: 144, Frequency: 660
Bit Count: 145, Frequency: 548
Bit Count: 146, Frequency: 417
Bit Count: 147, Frequency: 312
Bit Count: 148, Frequency: 233
Bit Count: 149, Frequency: 196
Bit Count: 150, Frequency: 109
Bit Count: 151, Frequency: 86
Bit Count: 152, Frequency: 54
Bit Count: 153, Frequency: 41
Bit Count: 154, Frequency: 23
Bit Count: 155, Frequency: 15
Bit Count: 156, Frequency: 9
Bit Count: 157, Frequency: 7
Bit Count: 158, Frequency: 7
Bit Count: 159, Frequency: 6
Bit Count: 160, Frequency: 1
Bit Count: 164, Frequency: 1

The following histogram displays this data.



By changing the value for “num_inputs”, new data is generated and altered, as well as a new histogram is presented reflecting the new data.

c)

This code has the implementation of question b and c:

```
# Number of hashes to compute
    num_inputs = 100000  # Adjust this for reasonable runtime
    bit_counts = []

    # Start timing
    start_time = time.time()

    # Generate random inputs, compute hashes, and count 1-bits
    for _ in range(num_inputs):
        #Generate a random input
        random_input = str(random.randint(0, 1 << 30)).encode()
```

```
        #Compute SHA-256 hash
        hash_value = hashlib.sha256(random_input).digest()

        # Count the number of 1 bits in the hash
        bit_count = sum(bin(byte).count('1') for byte in hash_value)
        bit_counts.append(bit_count)

    #End timing
    end_time = time.time()
    elapsed_time = end_time - start_time


    # Performance metrics
    hashes_per_second = num_inputs / elapsed_time

    # Estimate time for birthday attack (2^128 hashes)
    birthday_attack_time_years = (2 ** 128) / hashes_per_second / (3600 *
24 * 365)

    # Estimate time for brute-force attack (2^256 hashes)
    brute_force_attack_time_years = (2 ** 256) / hashes_per_second / (3600
* 24 * 365)

    # Print timing and performance results
    print("\nPerformance Metrics:")
    print(f"Elapsed Time: {elapsed_time:.3f} seconds")
    print(f"Number of Hashes Computed: {num_inputs}")
    print(f"Hashes per Second: {hashes_per_second:.3f}")
    print(f"Time for Birthday Attack: {birthday_attack_time_years:.2e}
years")
    print(f"Time for Brute-Force Attack:
{brute_force_attack_time_years:.2e} years\n")
```

During the generation of data, the performance is recorded in time. There are also values that are calculated to show how long it would take to compute hash values to challenge a birthday attack and brute force attack. The output is then printed for the user to understand these values. Using the same data from question b, here is an example:

Performance Metrics:

Elapsed Time: 2.332 seconds
Number of Hashes Computed: 100000
Hashes per Second: 42879.047
Time for Birthday Attack: 2.52e+26 years
Time for Brute-Force Attack: 8.56e+64 years

d)

For question d, instead of using mathematica due to installation issues, I used a library in python called SciPy. SciPy contains “modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering”. This information made it optimal to perform a task that otherwise mathematica would perform. Using 0.5 for the probability of one as given and 256 as the about of output bits, I was able to find the probability of these values in relation to one.

```
# Define the total number of bits and the probability of a bit being 1
total_bits = 256
probability_of_one = 0.5

# Calculate the probability of 128 bits being set to 1
probability_128 = binom.pmf(128, total_bits, probability_of_one)

# Calculate the probability of  100 bits being set to 1
probability_100 = binom.pmf(100, total_bits, probability_of_one)

# Display the results in scientific notation
print(f"\nThe probability of exactly 128 bits being 1:
{probability_128:.2e}\n")
print(f"The probability of exactly 100 bits being 1:
{probability_100:.2e}\n")
```

The results became:

The probability of exactly 128 bits being 1: 4.98e-02

The probability of exactly 100 bits being 1: 1.06e-04

When not outputted in scientific notation we get:

The probability of exactly 128 bits being 1: 0.04981910993614012

The probability of exactly 100 bits being 1: 0.00010625028465164735

2.

Cameron Bryant

2)

$M = 3$

Bob · $N_B = 143$, $e_B = 7$ Alice $N_A = 39$, $e_A = 5$

a)

$S = M^d \bmod N_B$

$N_B = P \cdot q = 11 \cdot 13$

$\phi(N_B) = 10 \cdot 12 = 120$

$7 \cdot d \equiv 1 \bmod \phi(N_B) = 7 \cdot d \equiv 1 \bmod 120$

$d = \frac{1 \bmod 120}{7}$

$120 / 7 = 17 \text{ r } 1$

$120 = (17 \cdot 7) + 1$

$1 = 120 - 17 \cdot 7 \Rightarrow 1 = -17 \cdot 7 + 1 \cdot 120$

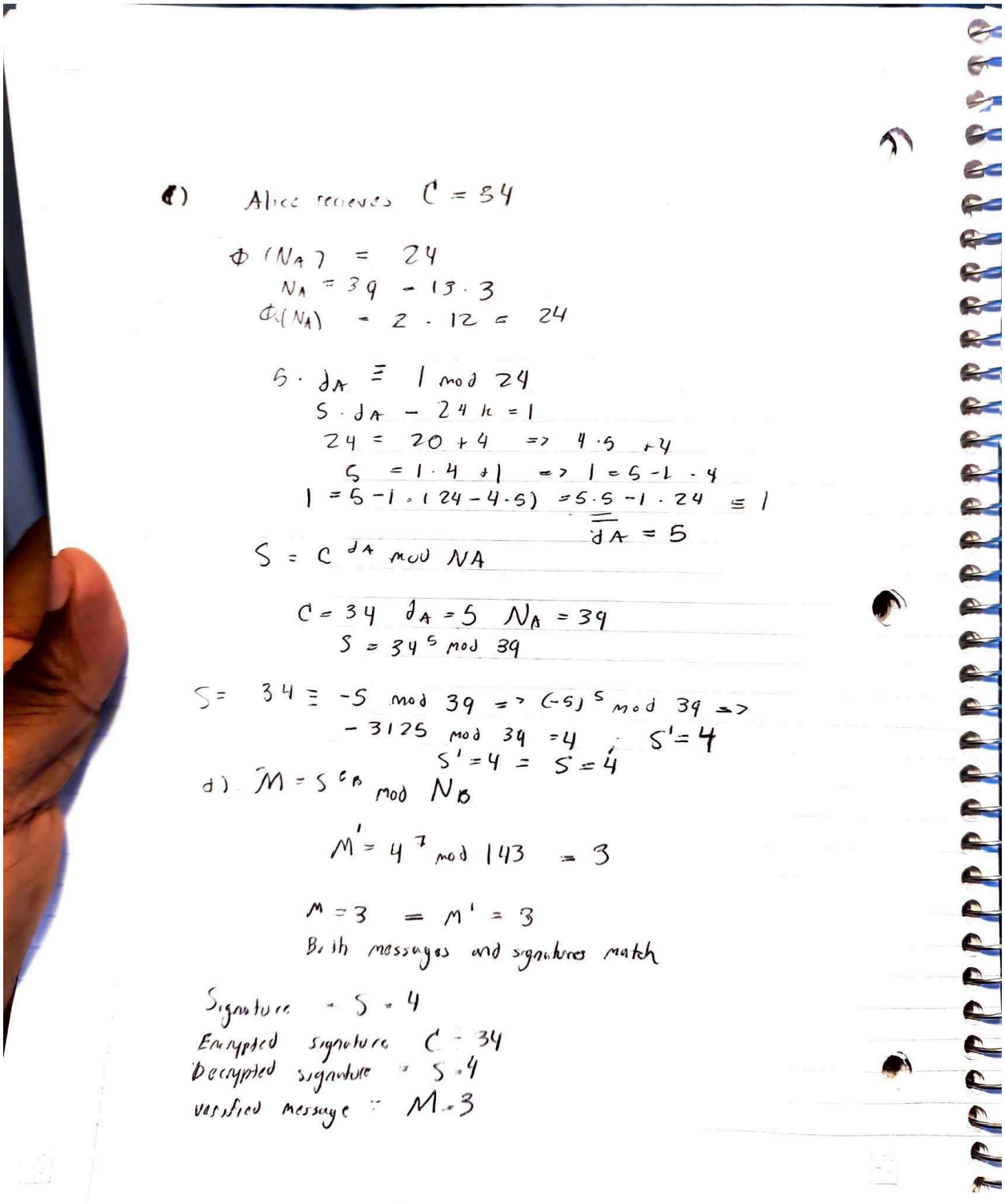
$d_B = -17 \bmod 120 = 103$

$S = 3^{103} \bmod 143 = 4$; $S = 4$

b) $C = S^{e_A} \bmod N_A = 4^5 \bmod 39$

$= 1024 \bmod 39 = 34$

$C = 34$



From these calculations, I found that the value of the signature S = 4, and when encrypted with the signature we got the value of C = 34. Alice was able to decrypt the signature for S = 4 and verify that the message she received was equal to the original message M = 3.