Collaborators: Steve Li, Jothi Ramaswamy

Problem 1. 1. To generate a fair coin flip, let us roll the biased die twice. Using symmetry, we see that the probability of rolling any two distinct numbers is the same as the probability of rolling those two numbers in reverse; for example, the probability of rolling 1 followed by 3 is the same as the probability of rolling 3 followed by 1. Of the 30 possible outcomes with distinct rolls, we see that there are 15 pairs of outcomes where the outcomes in each pair have the same probability of occurring. If we assign one outcome in each pair to Heads, and the other outcome to Tails, then the probability of getting Heads is equal to the probability of getting Tails. If the two dice rolls are not distinct, then we roll twice again. Thus e, the probability of obtaining a bit from each round of two dice rolls (or equivalently, the probability of the two rolls being distinct) is

$$e = 1 - p_1^2 - p_2^2 - p_3^2 - p_4^2 - p_5^2 - p_6^2$$

where p_n is the probability of rolling n. Using the general formula derived in class, we have that t, the expected number of total flips, satisfies $t = ef + (1-e)(f+t)$, or simplifying, $t = f/e$. We have e and $f = 2$ (the number of flips per round), so in this case, we have

$$t = \frac{2}{1 - p_1^2 - p_2^2 - p_3^2 - p_4^2 - p_5^2 - p_6^2}$$

.

2. To generate a fair die roll, let us roll the biased die three times. If the three rolls are not all distinct, then we roll three times again. If the three rolls are all distinct, then using symmetry, we see that each of the 6 possible permutations of the three rolls have an equal chance of occurring. In other words, if we consider any three distinct numbers a, b, c between 1 and 6, the probabilities of rolling abc (a followed by b followed by c), acb, bac, bca, cab, and cba are equal. Then we can assign each permutation of a, b, c to a number between 1 and 6 to simulate a fair dice roll, and we can do this for each of the 20 combinations of three distinct numbers from 1 to 6. Thus e, the probability of obtaining a bit from each round of three dice rolls (or equivalently, the probability of the three rolls being distinct) is

$$e = 6 \sum_{i=1}^{6} \sum_{j=i+1}^{6} \sum_{k=j+1}^{6} p_i p_j p_k$$

where p_n is the probability of rolling n. Then we have that the expected number of total flips is

$$t = f/e = \frac{3}{6 \sum_{i=1}^{6} \sum_{j=i+1}^{6} \sum_{k=j+1}^{6} p_i p_j p_k} = \frac{1}{2} \sum_{i=1}^{6} \sum_{j=i+1}^{6} \sum_{k=j+1}^{6} p_i p_j p_k$$

Problem 2. The code for this problem is included at the end. For this problem, the recursive and iterative methods were relatively easy to implement, while the matrix method required a little more consideration with regards to optimization. At first, I implemented a naive approach to matrix powers before changing to the matrix doubling method. In general, the matrix method was slower than the iterative method, which suggests that more optimization could be achieved, probably from

switching from a recursive to iterative approach to matrix powers.

	Recursive	*Iterative*	*Matrix*
1	~ 0	~ 0	~ 0
10	~ 0	~ 0	~ 0
10^2	Too long	0.000119898	0.000102913
10^3	Too long	0.000959091	0.001061417
10^4	Too long	0.005854727	0.000391501
10^5	Too long	0.052516754	0.009049859
10^6	Too long	0.443913589	0.395792052
10^7	Too long	2.980143625	9.868034229
10^8	Too long	29.63914318	Too long
10^9	Too long	Too long	Too long

Times measured in seconds. The largest Fibonacci numbers calculated in under ten seconds for each method were:

- Recursive: $n = 36$
- Iterative: $n = 35000000$
- Matrix: $n = 10000000$

Problem 3.

A	B	O	o	Ω	ω	Θ
$\log n$	$\log(n^2)$	*yes*	*no*	*yes*	*no*	*yes*
$\log(n!)$	$\log(n^n)$	*yes*	*no*	*yes*	*no*	*yes*
$\sqrt[3]{n}$	$(\log n)^6$	*no*	*no*	*yes*	*yes*	*no*
$n^2 2^n$	3^n	*yes*	*yes*	*no*	*no*	*no*
$(n^2)!$	n^n	*no*	*no*	*yes*	*yes*	*no*
$\frac{n^2}{\log n}$	$n \log(n^2)$	*no*	*no*	*yes*	*yes*	*no*
$(\log n)^{\log n}$	$\frac{n}{\log(n)}$	*no*	*no*	*yes*	*yes*	*no*
$100n + \log n$	$(\log n)^3 + n$	*yes*	*no*	*yes*	*no*	*yes*

Problem 4. 1. Find (with proof) a function f_1 such that $f_1(2n)$ is $O(f_1(n))$.

Proof. Let $f_1(n) = n$ and suppose $c = 2$ and $n_0 \in \mathbb{N}$. Then for all $n > n_0$, we have

$$f_1(2n) \leq c f_1(n)$$
$$2n \leq 2n$$

which is true, so by definition, $f_1(2n)$ is $O(f_1(n))$. □

2. Find (with proof) a function f_2 such that $f_2(2n)$ is not $O(f_2(n))$.

Proof. Let $f_2(n) = 2^n$ and suppose $c, n_0 \in \mathbb{N}$. Then for all $n > n_0$, we have

$$f_2(2n) \leq c f_2(n)$$
$$2^{2n} \leq c 2^n$$
$$2^n \leq c$$

This is false, as no constant c can be greater than or equal to 2^n for all n. Thus, $f_2(2n)$ is not $O(f_2(n))$. □

3. Prove that if $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Proof. If $f(n)$ is $O(g(n))$, then by definition there exists $c_1, n_1 \in \mathbb{N}$ such that $f(n) \leq c_1 g(n)$ for all $n > n_1$. Similarly, if $g(n)$ is $O(h(n))$, then by definition there exists $c_2, n_2 \in \mathbb{N}$ such that for all $n > n_2$, $g(n) \leq c_2 h(n)$, and multiplying both sides of the inequality by c_1 gives $c_1 g(n) \leq c_1 c_2 h(n)$. Let $n_3 = \max(n_1, n_2)$ and $c_3 = c_1 c_2$. Then for all $n > n_3$, we have

$$f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$$
$$f(n) \leq c_3 h(n)$$

Thus, $f(n)$ is $O(h(n))$. □

4. Give a proof or a counterexample: if f is not $O(g)$, then g is $O(f)$.

Proof. If f is not $O(g)$, then by definition, for any $c_1, n_1 \in \mathbb{N}$, we have that for all $n > n_1$,

$$f(n) > c_1 g(n)$$
$$c_1 g(n) < f(n)$$
$$g(n) < \frac{1}{c_1} f(n)$$

Let $c_2 = \frac{1}{c_1}$. Then we have

$$g(n) < c_2 f(n) \leq c_2 f(n)$$

for all $n > n_1$. Thus, g is $O(f)$. □

5. Give a proof or a counterexample: if f is $o(g)$, then f is $O(g)$.

Proof. We use proof by contrapositive. Suppose f is not $O(g)$. Then by definition, for any $c_1, n_1 \in \mathbb{N}$, we have that for all $n > n_1$, $f(n) > c_1 g(n)$, which implies that

$$\lim_{n\to\infty} \frac{f(n)}{c_1 g(n)} > 0 \implies \frac{1}{c_1} \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

so by definition, f is not $o(g)$. □

Problem 5. Suppose the general form of $T(n)$ is

$$T(n) = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1 \tag{1}$$

We will prove this result using induction (this result comes roughly from listing the first ten values of $T(n)$ and noticing a certain pattern within them).

First, we are given that $T(1) = 0$ and $T(2) = 1$. (1) holds for $n = 1$, since $T(1) = 1 \lceil \log_2 1 \rceil - 2^{\lceil \log_2 1 \rceil} + 1 = 1(0) - 2^0 + 1 = 0$, and it holds for $n = 2$ since $T(2) = 2 \lceil \log_2 2 \rceil - 2^{\lceil \log_2 2 \rceil} + 1 = 2(1) - 2^1 + 1 = 1$. For $n = 3$, we have by definition that $T(3) = T(1) + T(2) + 3 - 1 = 3$. We also have from (1) that $T(3) = 3 \lceil \log_2 3 \rceil - 2^{\lceil \log_2 3 \rceil} + 1 = 3(2) - 2^2 + 1 = 3$, so (1) holds for $n = 3$.

Next, we assume by induction that for any $k \geq 3$, (1) holds for all $n \leq k$. We want to show that (1) also holds for $n = k + 1$. Now we consider $T(k + 1)$.

If $k+1$ is even, then we have

$$\begin{aligned}
T(k+1) &= T\left(\left\lceil \frac{k+1}{2} \right\rceil\right) + T\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) + (k+1) - 1 \\
&= 2T\left(\frac{k+1}{2}\right) + k \\
&= 2\left[\left(\frac{k+1}{2}\right)\left\lceil \log_2 \frac{k+1}{2} \right\rceil - 2^{\lceil \log_2 \frac{k+1}{2} \rceil} + 1\right] + k \\
&= (k+1)\lceil \log_2(k+1) \rceil - 2^{\lceil \log_2(k+1) \rceil} + 1
\end{aligned}$$

so (1) holds for any even $k+1$.

If $k+1$ is odd, then we have

$$\begin{aligned}
T(k+1) &= T\left(\left\lceil \frac{k+1}{2} \right\rceil\right) + T\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) + (k+1) - 1 \\
&= T\left(\frac{k+2}{2}\right) + T\left(\frac{k}{2}\right) + k \\
&= \left(\frac{k+2}{2}\right)\left\lceil \log_2 \frac{k+2}{2} \right\rceil - 2^{\lceil \log_2 \frac{k+2}{2} \rceil} + 1 + \left(\frac{k}{2}\right)\left\lceil \log_2 \frac{k}{2} \right\rceil - 2^{\lceil \log_2 \frac{k}{2} \rceil} + 1 + k \\
&= \left(\frac{k+2}{2}\right)\lceil \log_2(k+2) - 1 \rceil - 2^{\lceil \log_2(k+2) - 1 \rceil} + 1 + \left(\frac{k}{2}\right)\lceil \log_2 k - 1 \rceil - 2^{\lceil \log_2 k - 1 \rceil} + 1 + k
\end{aligned}$$

In order to combine terms, we consider two cases. The first case is when k is a power of 2. In this case, we have that $\lceil \log_2(k+2) \rceil = \lceil \log_2(k+1) \rceil = \lceil \log_2 k \rceil + 1$, and substituting gives us

$$T(k+1) = \left(\frac{k+2}{2}\right)\lceil \log_2 k \rceil - 2^{\lceil \log_2 k \rceil} + 1 + \left(\frac{k}{2}\right)\lceil \log_2 k \rceil - \frac{k}{2} - \frac{2^{\lceil \log_2 k \rceil}}{2} + 1 + k$$

We also have that $k = 2^{\lceil \log_2 k \rceil}$, so we have

$$\begin{aligned}
T(k+1) &= \left(\frac{k+2}{2}\right)\lceil \log_2 k \rceil - 2^{\lceil \log_2 k \rceil} + \left(\frac{k}{2}\right)\lceil \log_2 k \rceil - \frac{2^{\lceil \log_2 k \rceil}}{2} - \frac{2^{\lceil \log_2 k \rceil}}{2} + 2^{\lceil \log_2 k \rceil} + 2 \\
&= \left(\frac{2k+2}{2}\right)\lceil \log_2 k \rceil - 2^{\lceil \log_2 k \rceil} + 2 \\
&= (k+1)\lceil \log_2(k+1) - 1 \rceil - 2^{\lceil \log_2(k+1) - 1 \rceil} + 2 \\
&= (k+1)\lceil \log_2(k+1) \rceil - k - 1 - 2^{\lceil \log_2(k+1) - 1 \rceil} + 2 \\
&= (k+1)\lceil \log_2(k+1) \rceil - 2^{\lceil \log_2(k+1) - 1 \rceil} - 2^{\lceil \log_2(k+1) - 1 \rceil} + 1 \\
&= (k+1)\lceil \log_2(k+1) \rceil - 2^{\lceil \log_2(k+1) \rceil} + 1
\end{aligned}$$

so (1) holds when $k+1$ is odd k is a power of 2.

The second case is when k is not a power of 2. In this case, we have that $\lceil \log_2(k+2) \rceil = \lceil \log_2(k+1) \rceil = \lceil \log_2 k \rceil$, and substituting (in the last line before we broke into cases) gives us

$$\begin{aligned}
T(k+1) &= \left(\frac{k+2}{2}\right)\lceil \log_2(k+2) - 1 \rceil - 2^{\lceil \log_2(k+2) - 1 \rceil} + 1 + \left(\frac{k}{2}\right)\lceil \log_2 k - 1 \rceil - 2^{\lceil \log_2 k - 1 \rceil} + 1 + k \\
&= \left(\frac{k+2}{2}\right)\lceil \log_2(k+1) \rceil - \frac{k+2}{2} - 2^{\lceil \log_2(k+1) - 1 \rceil} + \left(\frac{k}{2}\right)\lceil \log_2 k - 1 \rceil - 2^{\lceil \log_2 k - 1 \rceil} + k + 2
\end{aligned}$$

Again, we have that $k = 2^{\lceil \log_2 k \rceil}$, so we have

$$\begin{aligned} T(k+1) &= \left(\frac{k+2}{2}\right)\lceil \log_2(k+1)\rceil - \frac{k+2}{2} - 2^{\lceil \log_2(k+1)-1\rceil} + \left(\frac{k}{2}\right)\lceil \log_2(k+1)\rceil - \frac{k}{2} - 2^{\lceil \log_2(k+1)-1\rceil} + k + 2 \\ &= \left(\frac{2k+2}{2}\right)\lceil \log_2(k+1)\rceil - \frac{2^{\lceil \log_2(k+1)\rceil}}{2} - \frac{2^{\lceil \log_2(k+1)\rceil}}{2} - \frac{2k+2}{2} + k + 2 \\ &= (k+1)\lceil \log_2(k+1)\rceil - 2^{\lceil \log_2(k+1)\rceil} - (k+1) + k + 2 \\ &= (k+1)\lceil \log_2(k+1)\rceil - 2^{\lceil \log_2(k+1)\rceil} + 1 \end{aligned}$$

so (1) holds when $k+1$ is odd k is a power of 2.

Therefore, we have proved that (1) holds for $n = k+1$ in all cases, thus closing the induction.

Problem 2 Code

```
# Written in Python 3.7.1
import timeit

def fib_recursive(n):
        if n == 0:
                return 0
        elif n == 1:
                return 1
        else:
                return (fib_recursive(n - 1) + fib_recursive(n - 2)) % 65536

def fib_iterative(n):
        a = [0, 1]
        for i in range(1, n):
                a.append((a[-1] + a[-2]) % 65536)
        return a[n]

def matrix_multiply(m1, m2):
        a, b, c, d = m1
        e, f, g, h = m2
        return [a*e + b*g, a*f + b*h, c*e + d*g, c*f + d*h]

def matrix_power(m, n):
        if n == 0:
                return [1, 0, 0, 1]
        if n == 1:
                return m
        elif n % 2 == 0:
                x = matrix_power(m, n / 2)
                return matrix_multiply(x, x)
        else:
                x = matrix_power(m, n // 2)
                return matrix_multiply(m, matrix_multiply(x, x))
```

```
def fib_matrix(n):
        m = [0, 1, 1, 1]
        return matrix_power(m, n)[1] % 65536

if __name__ == "__main__":
        setup_code = '''
from __main__ import fib_matrix
        '''
        test_code = '''
print(fib_matrix(1000))
        '''
        timer = timeit.timeit(setup=setup_code, stmt=test_code, number=1)
        print(timer)
```