1. Suppose you are given a six-sided die, that might be biased in an unknown way. Explain how to use die rolls to generate unbiased coin flips, and determine the expected number of die rolls until a coin flip is generated.

We will label our die's 6 sides Side 1 - Side 6 with probabilities of being rolled equal to $p_1 - p_6$. To generate a heads or tails, we will roll the die twice, calling our first roll Roll X and our second roll Roll Y. If Roll X and Roll Y yield the same side, then we roll again. If Roll X yields a side that is larger than Roll Y, we will call that a heads. If Roll X yields a side that is smaller than Roll Y, we will call that a tails. Because the probability of two different sides being rolled in one order is the same as the probability of them being rolled in the reverse order ie. $p_a * p_b = p_b * p_a$, we can determine that the coin flip generated is unbiased.

We will calculate the expected number of die rolls to generate a single coin flip. Let us name the expected number to be C. We will call the variable $p_{doubles}$ equal to the sum of the probabilities of rolling all kinds of doubles:

$$p_{doubles} = \sum_{i=1}^{6} p_i^2$$

$$C = (1 - p_{doubles})(2) + (p_{doubles})(C + 2)$$

$$C - C * p_{doubles} = 2$$

$$C = \frac{2}{1 - p_{doubles}} = \frac{2}{1 - \sum_{i=1}^{6} p_i^2}$$

Now suppose instead you want to generate unbiased die rolls (from a six-sided die) given your potentially biased die. Explain how to do this, and again determine the expected number of biased die rolls until an unbiased die roll is generated. For both problems, you need not give the most efficient solution – simple, non-recursive are all that we are looking for – however, your solution should be reasonable, and exceptional solutions will receive exceptional scores.

We will number our die's 6 sides Side 1 - Side 6 with probabilities of being rolled equal to $p_1 - p_6$. We will maintain the same concept as the unbiased coin flip, except we will add a die roll to yield 3 times as many outcomes (which will result in 6 different results). We will roll the biased die three times, calling our first roll Roll X, calling our second roll Roll Y, and calling our third roll Roll Z. If any of those three rolls yield the same side value, we will flip the die again three more times. With all 3 Rolls being different values, we will compare those three values. Of the three rolls with three different numbers, one of the three rolls yields the highest value–we will call that value H. Of the two rolls remaining, one of the rolls yield the lowest side value, we will call that value L. The other roll we will call value M.

We will order our three Rolls Roll X-Roll Y-Roll Z. Below, we have assigned final values to each ordering pattern.

If the order of Roll X-Roll Y-Roll Z fits the pattern of...
L-M-H, then we will assign that ordering pattern the value 1.
L-H-M, then we will assign that ordering pattern value 2.
M-L-H, then we will assign that ordering pattern the value 3.
M-H-L, then we will assign that ordering pattern the value 4.
H-L-M, then we will assign that ordering pattern the value 5.

H-M-L, then we will assign that ordering pattern the value 6.

The probabilities of any of those orderings are equal because of the Law of Commutativity ie. $p_a * p_b * p_c = p_b * p_a * p_c = ...$

(Note** If we were to flip the die again less than three times every time a side was repeated in the sequence, then the results would be skewed: If the side with the lowest value had a very high probability of being rolled, it is likely that the the simulation would yield a 1 or 2 high frequency).

Let us calculate the expected value an unbiased roll; we will call the expected value D. We will call the sum of all triples and all doubles $p_{nondistinct}$.

$$p_{nondistinct} = \sum_{i=1}^{6} p_i^3 + \sum_{i=1}^{6} +p_i^2(1-p_i) = \sum_{i=1}^{6} p_i^3 + p_i^2(1-p_i) = \sum_{i=1}^{6} p_i^3 + p_i^2 - p_i^3 = \sum_{i=1}^{6} p_i^2$$

$$D = (1 - p_{nondistinct})(3) + (p_{nondistinct})(D+3)$$

$$D - D * p_{nondistinct} = 3$$

$$D = \frac{3}{1 - p_{nondistinct}} = \frac{3}{1 - \sum_{i=1}^{6} p_i^2}$$

2. On a platform of your choice, implement the three different methods for computing the Fibonacci numbers (recursive, iterative, and matrix) discussed in lecture. (LAST PAGE)

3. Indicate for each pair of expressions (A, B) in the table below the relationship between A and B. Your answer should be in the form of a table with a "yes" or "no" written in each box. For example, if A is $O(B)$, then you should put a "yes" in the first box.

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*	*yesno*

4. For all of the problems below, when asked to give an example, you should give a function mapping positive integers to positive integers. (No cheating with 0's!)

   - Find (with proof) a function f_1 such that $f_1(2n)$ is $O(f_1(n))$.
   - Find (with proof) a function f_2 such that $f_2(2n)$ is not $O(f_2(n))$.
   - Prove that if $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.
   - Give a proof or a counterexample: if f is not $O(g)$, then g is $O(f)$.
   - Give a proof or a counterexample: if f is $o(g)$, then f is $O(g)$.

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

Proof. By definition of O, a (n) is O(b(n)) if and only if there exists a constant c and a minimum value n_0 such that for all n $\geq n_0$, $a(n) \leq c * b(n)$.

$$\text{Let } f_1(n) = n$$
$$\text{Thus, } f_1(2n) = 2n$$
$$\text{When } n \text{ is any integer and c} = 2,\ 2n \leq 2n$$
$$\text{Thus, } f_1(2n) \leq c * f_1(n)$$

This fits the formatted definition of O with $n > -\infty$ and c = 2. Because this fits the formatted definition of O, we have proved $f_1(2n)$ is $O(f_1(n))$ ■

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

Proof. By definition of O, a(n) is O(b(n)) if and only if

$$\lim_{x\to\infty} \frac{a(n)}{b(n)} < \infty$$

$$\text{Let } f_2(n) = n^{\log_2(n)}$$

We will use proof by contradiction. Let us assume that $f_2(2n)$ is $O(f_2(n))$. Thus,

$$\lim_{x\to\infty} \frac{f_2(2n)}{f_2(n)} < \infty$$

$$f_1(2n) = n^{\log_2(2n)} = n^{\log_2(n)+\log_2(2)} = n^{\log_2(n)+1} = n^{\log_2(2n)} * n$$

$$\lim_{x\to\infty} \frac{f_2(2n)}{f_2(n)} = \frac{n * n^{\log_2(n)}}{n^{\log_2(n)}} = \frac{n}{1} = \infty$$

Because ∞ is not less than ∞, this does not fit the definition of O. Thus we have found a contradiction, and we have proved that $f_2(2n)$ is not $O(f_2(n))$. ■

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

Proof. By definition of O, a(n) is O(b(n)) if and only if there exists a positive constant c and a minimum value n_0 such that for all n $\geq n_0$,

$$a(n) \leq c * b(n)$$

Because we know that f(n) is O(g(n)), then there exists a positive constant c_1 and a minimum n_{01} such that $f(n) \leq c_1 * g(n)$ for all n $\geq n_{01}$. Because we know that g(n) is O(h(n)), then there exists a positive constant c_2 and a minimal n_{02} such that $g(n) \leq c_2 * h(n)$ for all n $\geq n_{02}$.

$$f(n) \leq c_1 * g(n)$$

$$g(n) \leq c_2 * h(n)$$

If we multiply the second equation listed above by the positive constant c_1 we get,

$$c_1 * g(n) \leq c_1 * c_2 * h(n)$$

. Based upon the first equation given, we can add f(n) term to this equation:

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

Taking the middle g(n) term out of the equation,

$$f(n) \leq c_1 * c_2 * h(n)$$

We can say that $c_1 * c_2 = c$. Because the multiplication of two positive constants yields a positive constant, we know that c is a positive constant. We can declare n to be greater than or equal to the larger of n_{01} and n_{02}.
Because this fits the formatted definition of O with $n \geq Max(n_{01}, n \geq n_{02})$ and c being a positive constant, we have proved $f(n)$ is $O(h(n))$ given that $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$. ■

- Give a proof or a counterexample: if f is not $O(g)$, then g is $O(f)$.
  Counterexample: $f(n) = n^2$
  g(n) = $\begin{cases} n & \text{if n mod } 2 = 0 \\ n^3 & \text{if n mod } 2 = 1 \end{cases}$
  When n is even, g(n) = n, and n or g(n) in this case is O(f(n)). But when n is odd, g(n) = n^3, and f(n) is O(n^3) or O(g(n)). Thus, f(n) is not O(g(n)) and g(n) is not O(f(n)).
- Give a proof or a counterexample: if f is $o(g)$, then f is $O(g)$.

Proof. Let us define a = O(b) as

$$\lim_{x \to \infty} \frac{a}{b} < \infty$$

Let us define a = o(b) as

$$\lim_{x \to \infty} \frac{a}{b} = 0$$

For f to be $o(g)$, then by definition of o,

$$\lim_{x \to \infty} \frac{f}{g} = 0$$

$$\lim_{x \to \infty} \frac{f}{g} = 0 < \infty$$

$$\lim_{x \to \infty} \frac{f}{g} < \infty$$

Because this fits the definition of O, we have proved that f is $O(g)$. ■

5. We found that a recurrence describing the number of comparison operations for a mergesort is $T(n) = 2T(n/2) + n - 1$ in the case where n is a power of 2. (Here $T(1) = 0$ and $T(2) = 1$.) We can generalize to when n is not a power of 2 with the recurrence

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1.$$

Exactly solve for this more general form of $T(n)$, and prove your solution is true by induct ion. Hint: plot the first several values of $T(n)$, graphically. What do you find? You might find the following concept useful in your solution: what is $2^{\lceil \log_2 n \rceil}$?

Proof. The solution's equation is:

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

We use proof by induction to prove the solution.
Our two base cases, b_1 and b_2 must yield T(1) = 0 and T(2) = 1.

$$b_1 = T(1) = 1 * 0 - 2^0 + 1 = -1 + 1 = 0$$

$$b_2 = T(2) = 2 * 1 - 2^1 + 1 = 2 - 2 + 1 = 1$$

Inductive hypothesis: Let us assume that the solution holds for all values of positive integers k in which k $\leq m$. Thus,

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

Case I: In the case that m is odd, we can represent m as 2q - 1 in which q is a positive integer less than m. To prove our solution holds, we will say that 2q = m + 1.

Case II: In the case that m is even, we can represent m as 2q - 2 in which q is a positive integer less than m. To prove our solution holds, we will say that 2q = m + 2.

Inductive step: We will prove that the solution holds for 2q which represents either m + 1 or m + 2. This is the solution that we are trying to prove:

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

We can begin with the equation that we have been given and know to be true:

$$T(2q) = T(\lceil \frac{2q}{2} \rceil) + T(\lfloor \frac{2q}{2} \rfloor) + 2q - 1$$

$$= T(\lceil q \rceil) + T(\lfloor q \rfloor) + 2q - 1$$

$$= T(q) + T(q) + 2q - 1$$

Because q $\leq$ m, the solution holds for T(q):

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

$$= 2q\lceil \log_2 q \rceil - 2 * 2^{\lceil \log_2 q \rceil} + 2q + 1$$

$$= 2q(\lceil \log_2 q \rceil + 1) - 2^{\lceil \log_2 q \rceil + 1} + 1 =$$

$$= 2q(\lceil \log_2 q \rceil + log_2 2) - 2^{\lceil \log_2 q \rceil + log_2 2} + 1$$

$$= 2q(\lceil \log_2 q \rceil + \lceil \log_2 2 \rceil) - 2^{\lceil \log_2 q \rceil + \lceil log_2 2 \rceil} + 1$$

We can apply the following ceiling function property in which m is a variable and c is an integer:

$$\lceil m \rceil + c = \lceil m + c \rceil$$

$$= 2q(\lceil \log_2 q + \log_2 2 \rceil) - 2^{\lceil \log_2 q + log_2 2 \rceil} + 1$$

$$= 2q(\lceil \log_2 2 * q \rceil) - 2^{\lceil \log_2 2*q \rceil} + 1$$

Thus, we have proved for 2q (which is equal to either m + 1 or m + 2:

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

Our final solution is

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

■

2) Submit your source code with your assignment. Please give a reasonable English explanation of your experience with your program(s).

The code can be run using three inputs: the fibonacci testing input, the 10 second input, and the overflow input. All fibonacci methods are run through a timer function and print the fibonacci term as well as the time in either seconds or microseconds. There is a helper file with methods for matrix multiplication and matrix repeated squaring.

For smaller fibonacci numbers, the iterative method is the fastest, followed by the matrix method, followed by the recursive method: Ex) for the 10th fibonacci number, the recursive method takes 1216 microseconds, the matrix method takes 272 microseconds, and the iterative method takes 37 microseconds.

For larger fibonacci numbers, the matrix multiplication method is fastest, followed by the iterative, followed by the recursive. Ex) for the 40,000 the fibonacci number, the iterative method takes 3151 microseconds and the matrix method takes 474 microseconds.

For the 10 second interval, the matrix multiplication can generate the largest number: it can generate a number nearly 10 factors of 10 higher than the iterative can in ten seconds. The iterative method is second fastest, and able to calculate a number in the hundreds of millions spot. The recursive method is the slowest by a large factor and can only calculate 44 in 10 seconds. When testing various fibonacci inputs, the recursive method had to frequently be commented out.

The overflow feature makes the code not perform mod 2^{16}. The last input before overflow is 1476. The input 1477 triggers the code to output "infinity".

```
class pset_1 {
    public static double Fib1_10 = 44.;
    public static double Fib2_10 = 970000000;
    public static float Fib3_10 = 999999999;
    public static double overflow_input = 1476.;
    public static boolean overflow = false;
    public static void main(final String[] args) {
        if ((args.length == 0 || args.length > 2) || ((!(isStringDouble(args[0])) &&
            System.out.println("Please follow the class name with overflow, 10 secs, 
        } else if (args.length == 2 && args[1].equals("secs")){
            timer(1, Fib1_10, true);
            timer(3, Fib3_10, true);
            timer(2, Fib2_10, true);
        } else if (args[0].equals("overflow")) {
            overflow = true;
            timer(2, overflow_input, false);
            timer(2, overflow_input + 1, false);
            timer(3, overflow_input, false);
            timer(3, overflow_input + 1, false);
        } else {
            timer(1, Double.parseDouble(args[0]), false);
            timer(2, Double.parseDouble(args[0]), false);
            timer(3, Double.parseDouble(args[0]), false);
        }

    }
```

```
public static boolean isStringDouble(String s)
{
    try
    {
        Double.parseDouble(s);
        return true;
    } catch (NumberFormatException ex)
    {
        return false;
    }
}


public static double Fib1_recursive(double n){
    if (n == 0){
        return 0;
    } else if (n == 1){
        return 1;
    } else {
        if (!overflow){
            return ((Fib1_recursive(n - 1) +
            Fib1_recursive(n - 2)) %
            pset_1_helper.base_a);
        } else {
            return ((Fib1_recursive(n - 1) +
            Fib1_recursive(n - 2)));
        }
    }
}


public static double Fib2_iterative(double n){
    double [] A = new double[3];
    A[0] = 0;
    A[1] = 1;
    for (int i = 2; i <= n; i++){
        if (!overflow){
            A[(i % 3)] = ((A[((i - 1) % 3)] + A [((i - 2) % 3)])
            % pset_1_helper.base_a);
        } else {
            A[i % 3] = (A[((i - 1)) % 3] + A [(i - 2) % 3]);
        }
    }
    if (!overflow){
        return A[(int)(n % 3)];
    } else {
        return A[(int)(n) % 3];
    }
```

```
    }


    public static double Fib3_matrix(double n_large){
        if (n_large == 0){
            return 0;
        }
        double n = n_large - 1;
        double[][] magic_matrix = {
            new double [] {0, 1},
            new double [] {1, 1}
        };
        String param = "0" + Long.toBinaryString((long)n);
        return (pset_1_helper.exp_matrices(magic_matrix, param)[1][1]);
    }

    public static void timer(int fun, double fib, boolean sec){
        long startTime = System.nanoTime();
        if (fun == 1){
            System.out.println(Fib1_recursive(fib));
        }
        else if (fun == 2){
            System.out.println(Fib2_iterative(fib));
        }
        else if (fun == 3){
            System.out.println(Fib3_matrix(fib));
        }
        long endTime = System.nanoTime();
        String duration_Fib;
        if (sec){
            duration_Fib = ((endTime - startTime) / 1000000000) + " seconds";

        } else {
            duration_Fib = ((endTime - startTime) / 1000) + " microseconds";
        }
        System.out.println("Fib" + fun + " took " + duration_Fib);
    }

}
class pset_1_helper {

    public static int base_a = (int)Math.pow(2, 16);
    public static double[][] multiplyMatrices(
        double[][] firstMatrix, double[][] secondMatrix) {
        double[][] result = new double[firstMatrix.length][secondMatrix[0].length];

        for (int row = 0; row < result.length; row++) {
            for (int col = 0; col < result[row].length; col++) {
```

```
                double cell = 0;
                for (int i = 0; i < secondMatrix.length; i++){
                    if (!pset_1.overflow){
                        cell += ((firstMatrix[row][i]) * (secondMatrix[i][col]));
                        cell = cell % base_a;
                    } else {
                        cell += ((firstMatrix[row][i] * secondMatrix[i][col]));
                    }
                }
                result[row][col] = cell;
            }
        }
        return result;
    }


    public static double[][] exp_matrices(double[][] m_matrix, String bin) {
        if (bin.equals("0")){
            double [][] ident = {{1, 0}, {0, 1}};
            return ident;
        }
        else if (bin.equals("1")){
            return m_matrix;
        }
        else if (bin.substring(bin.length() - 1).equals("0")){
            return exp_matrices(multiplyMatrices(m_matrix, m_matrix),
            bin.substring(0, bin.length() - 1));
        }
        else {
            return multiplyMatrices(m_matrix,
            exp_matrices(multiplyMatrices(m_matrix, m_matrix),
            bin.substring(0, bin.length() - 1)));
        }
    }
}
```