Harvard ID Number: 80977816

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. Now suppose 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; however, your solution should be reasonable, and exceptional solutions will receive exceptional scores.

   (a) To generate a coin flip from a biased die, roll the die twice. If the number that appears is the same both times, roll the dice twice more. If the first number rolled is larger than the second number rolled, call that pair of rolls a heads (H). If the second number is larger than the first, call that pair a tails. This is a valid way to generate a coin flip because the probability of rolling an x and then a y where $x > y$ is the same as the probability of rolling y and then an x by symmetry and so the probability of generating a heads in this method is 0.5. The probability that we successfully generate a valid coin flip when we roll a die twice is 1 - P(failure) = 1 - $\sum_{i=1}^{n} p_i^2$ where p_i is the probability of rolling an i. Therefore the expected number of rolls we need to do to generate a valid coin flip is $2 \cdot \frac{1}{1-\sum_{i=1}^{n} p_i^2} = \frac{2}{1-\sum_{i=1}^{n} p_i^2}$ because the pairs of rolls follow a Geometric distribution with p = 1 - P(failure).

   (b) Before rolling the die, group the faces into three groups of two and assign each group a letter. So, $\{1,2\} \to x, \{3,4\} \to y$, and $\{5,6\} \to z$. Now roll the die three times. After we convert the die rolls to these letters, if the same letter appeared twice, roll the die another 3 times to generate a new die roll. If one number from each of these three groups appeared then we can match the order in which they appeared to some number from 1 to 6, because there are 3!=6 permutations of 3 distinct objects. Moreover, each of these permutations is equally likely and so by figuring out which permutation we have we can determine which number the three rolls generate. For example $xyz \to 1, xzy \to 2, yzx \to 3, yzx \to 4, zxy \to 5, zyx \to 6$. The probability that rolling the die three times successfully generates a random die roll is equal to $6(p_1+p_2)(p_3+p_4)(p_5+p_6)$ which p_i is the probability of rolling number i. Therefore the expected number of groups of 3 we need to roll before succeeding is $\frac{1}{6(p_1+p_2)(p_3+p_4)(p_5+p_6)}$ and so the expected total number of rolls is $\frac{3}{6(p_1+p_2)(p_3+p_4)(p_5+p_6)} = \frac{1}{2(p_1+p_2)(p_3+p_4)(p_5+p_6)}$

2. On a platform of your choice, implement the three different methods for computing the Fibonacci numbers (recursive, iterative, and matrix) discussed in lecture. Use integer variables. How fast does each method appear to be? Give precise timings if possible. (This is deliberately open-ended; give what you feel is a reasonable answer. You will need to figure out how to time processes on the system you are using, if you do not already know.) Can you determine the first Fibonacci number where you reach integer overflow? (If your platform does not have integer overflow – lucky you! – you might see how far each process gets after five minutes.)

   Since you should reach integer overflow with the faster methods quite quickly, modify your programs so that they return the Fibonacci numbers modulo $65536 = 2^{16}$. (In other words, make all of your

arithmetic modulo 2^{16} – this will avoid overflow! You must do this regardless of whether or not your system overflows.) For each method, what is the largest value of k such that you can compute the kth Fibonacci number (or the [kth Fibonacci number] modulo 65536) in one minute of machine time?

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

When programming in C with 32-bit signed integers, the largest Fibonacci number I can calculate without overflow is F_{47} which equals 1836311903. F_{48} causes overflow. My recursive implementation of the Fibonacci numbers, which does not memoize or store any results to speed up the calculation of the Fibonacci numbers, runs incredibly slowly. In one minute, it can only calculate up to the 46th Fibonacci when starting at the first Fibonacci number. The iterative implementation is even faster - in one minute it can compute the first 94737 Fibonacci numbers (mod 65536). The matrix multiplication method, using the faster exponentiation method discussed in class, is by far the fastest - in one minute my computer calculated the first 11210732 Fibonacci numbers (mod 65536).

The main function of the program will run the recursive, iterative, and matrix algorithms all for a minute each and will print out the results of each calculation. For the iterative and matrix methods only the results that are multiples of 10000 and 1000000, respectively, are printed. As noted in the comment for the matrix method, I used unsigned ints so I could take advantage of the full 32 bits because some of the numbers involved in multiplying come close to 65536 and when multiplied together are greater than 2^{31}.

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	Yes

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 $O(g)$.

(a) Let $f_1(n) = 3n$. So, $f_1(2n) = 6n$. Let $c = 3, N = 0$
So $\forall n \geq N$, $f_1(2n) \leq 3f_1(n)$ because $6n \leq 9n$ for $n \geq 0$. This implies that $f_1(2n) = O(f_1(n))$.

(b) Let $f_2(n) = 2^n$. $f_2(2n) = 2^{2n} = 4^n$. To show that $f_2(2n)$ is not $O(f_2(n))$ suppose it is and I'll show this leads to a contradiction.
If $f_2(2n) = O(f_2(n))$ then there is$c > 0, N \geq 0$ such that for all $n \geq N, f_2(2n) \leq c \cdot f_2(n)$
Therefore $2^{2n} \leq c2^n$ which implies that $2^n \leq c$ which is clearly a contradiction because for any fixed c, there is always a large enough n such that $2^n > c$.

(c) Because $f(n) = O(g(n))$, there is $c_1 > 0, N_1 \geq 0$ such that for all $n \geq N_1, f(n) \leq c_1 g(n)$ and that because $g(n) = O(h(n))$, there is $c_2 > 0, N_2 \geq 0$ such that forall $n \geq N_2, f(n) \leq c_2 g(n)$.
So $c_1 g(n) \leq c_1 c_2 h(n)$ for $n \geq N_2$ and since we know that $f(n) \leq c_1 g(n)$ for $n \geq N_1$, then $f_1(n) \leq c_1 c_2 h(n)$ for $n \geq \max(N_1, N_2)$.

(d) This statement if f is not $O(g)$, then g is $O(f)$ is false. Suppose $f(n) = n$ and $g(n) = n^{1+cos(0.5n\pi)}$. Then if n is an odd number $g(n)$ equals n but if n is a multiple of 4, then $g(n) = n^2$ but equals 1 if n is not odd and not a multiple of 4 while $f(n)$ always equal n no matter what. Therefore because f is always larger than g when n is not odd and not a multiple of four but always less if it is a multiple of 4, f cannot be $O(g)$ and g cannot be $O(f)$.

(e) The statement that if f is $o(g)$, then f is $O(g)$ is true.
Suppose that $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c - 1$ for some constant $c > 1$. So at some finite n^* we can say that $f(n) < c \cdot g(n)$ which implies that for any n greater than n^*, $f(n) \leq cg(n)$ which means that f is $O(g(n))$. The important takeaway here is that if the limit of the quotient of two functions is finite then the function in teh numerator is O of the funciton in the denominator.

By definition, if $f = o(g)$, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ so if we pick some $c > 0$ at some value n^*, for all $n \geq n^*$, $f(n) \leq cg(n)$ which implies that f is $O(g)$ by what was just shown above.

5. **Do not turn this in. This is a suggested exercise.**
InsertionSort is a simple sorting algorithm that works as follows on input $A[0], \ldots, A[n-1]$.

```
InsertionSort(A)
    for i = 1 to n − 1
      j = i
      while j > 0 and A[j − 1] > A[j]
        swap A[j] and A[j − 1]
        j = j − 1
```

Show that for any function $T = T(n)$ satisfying $T(n) = \Omega(n)$ and $T(n) = O(n^2)$ there is an infinite sequence of inputs $\{A_k\}_{k=1}^{\infty}$ such that A_k is an array of length k, and if $t(n)$ is the running time of InsertionSort on A_n, then the order of growth of $t(n)$ is $\Theta(T(n))$.

I collaborated with Manav Khandelwal and Lauren Kim on this assignment.