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.

   Solution.
   Given a six-sided die, label each side one through six. Consider a result of sides 1-3 to be 'A', and a result of sides 4-6 to be 'B.' Then, the question can be approached by Von Neumann's algorithm. Roll the dice twice. Consider AB to be a result of heads in the simulated fair coin, and BA to be a result of tails. If a result of AA or BB is rolled, disregard and roll again. Roll until either AB or BA is achieved, as those two results represent a valid fair coin flip. Similar to the Von Neumann algorithm, the expected number of die rolls until a coin flip is generated is $1/pq$; consider the probability of sides 1-3 on the die as p, and the probability of sides 4-6 as $q = 1 - p$. Then, the probability of generating a fair coin flip in a round is $2pq$, and we multiply by 2 to account for the two rolls in one round. Considering the rolls as a set of independent Bernoulli trials, the expected value is $1/pq$.

   To generate unbiased die rolls, we must create six events of equal probability, instead of two in the first algorithm. So, we consider all permutations of three events, creating six groups of equal probability, and we can achieve an unbiased die roll in three rolls of any biased die. We roll the die three times, and then consider the relative values of the rolls, calling the smallest roll a, the middle roll b, and the largest roll c. For example, the roll combination 1-2-4 would be abc, while the roll 6-1-2 would be cab. If any result is rolled more than once during the three rolls, disregard the set and roll again. Assign the result abc to correspond to a value of 1 in the unbiased die, acb to 2, bac to 3, etc. for all six permutations. Each permutation has an equal probability of being rolled, and by disregarding the incompatible sets of three rolls, we can simulate an unbiased die using any biased die. To find the expected number of biased die rolls until an unbiased roll is generated, consider the probability of success, which will be one minus the scenarios when the same number is rolled three times in a set of three rolls, or the same number is rolled two out of three times. Assigning p_1 as the probability of rolling a 1, p_2 as the probability of rolling a 2, etc., this comes out to be $1-(p_1^3+p_2^3+p_3^3+p_4^3+p_5^3+p_6^3+p_1^2(1-p_1)+p_2^2(1-p_2)+p_3^2(1-p_3)+p_4^2(1-p_4)+p_5^2(1-p_5)+p_6^2(1-p_6))$. Again treating rolls as a set of independent Bernoulli trials, the expected value will be one divided by this expression, and then multiplied by 3 to account for the 3 die rolls within one round; thus, the expected number of rolls is $3/(1-(p_1^3+p_2^3+p_3^3+p_4^3+p_5^3+p_6^3+p_1^2(1-p_1)+p_2^2(1-p_2)+p_3^2(1-p_3)+p_4^2(1-p_4)+p_5^2(1-p_5)+p_6^2(1-p_6)))$.

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).

Solution.
See file *benegasps*1.*c* for all three methods implemented in C. The recursive method was quite slow, taking around 1.4 seconds to return the 40th Fibonacci number, while the iterative and matrix methods were both quite fast at that level, taking around 0.00002 seconds. With higher inputs, the matrix methods proved fastest. For each method, the first Fibonacci method to reach integer overflow was 49. Once the modulo was applied to each method, the 49th Fibonacci number was the largest number the recursive algorithm could compute in one minute of machine time, while the iterative and matrix methods could compute everything; given $MAXINT$, they both computed in under a minute.

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 $O(g)$.

Solution.

- We will prove this by choosing an example and using the definition of big-O. Let $f_1(n) = n$. Then, $f_1(2) = 2n$. In order for $f_1(2n)$ to be $O(f_1(n))$, the inequality $f_1(2n) \leq c * f_1(n)$ must hold for some positive constants c, N where $n \geq N$. Simplifying the inequality, we get that $2n \leq cn$, which is true for all $c \geq 2$. Therefore, for any c under this condition, $f_1(2n)$ is $O(f_1(n))$.
- Let $f_2(n) = n!$. Then $f_2(2n) = (2n)!$. In order for $f_2(2n)$ to not be $O(f_2(n))$, the inequality $f_2(2n) > c * f_2(n)$ must hold for some constants c, N such that $n \geq N$. Plugging in our function, we get $(2n)! > c * (n)!$. Simplifying the inequality, we can expand to $2n(2n-1)(2n-2)...(1) >$

$cn(n-1)(n-2)...(1)$, and we can then cancel out all of the terms with n on the right side, ending up with $2n(2n-1)...(n+1) > c$. Clearly, for any large n, the left side will always be larger than any constant c, so the inequality is satisfied and thus $f_2(n)$ is not $O(f_2(n))$.

- Following from the big-O definition, $f(n)$ is $O(g(n))$ if there exist constants c, N_0 such that for all $n > N_0$ then $f(n) \leq c * g(n)$, and $g(n)$ is $O(h(n))$ if there exist constants d, N_1 such that for all $n > N_1$ then $g(n) \leq d * h(n)$. If we divide by c in the first inequality, we are left with $f(n)/c \leq g(n)$, and we can then substitute in for $g(n)$ in the second inequality to get $f(n)/c \leq d * h(n)$. Multiply by c to get $f(n) \leq c * d * h(n)$, which is the definition of $f(n) = O(h(n))$, as the constants c and d become a single constant. Thus, the transitive property holds.
- This statement is false. Consider $f(n)$ to be a piecewise function where odd inputs correspond to the function 2^n and even inputs correspond to the function $log(n)$. Let $g(n) = n$. We can see that f is not $O(g)$, because there is no constant N that exists for all constants c and when $n > N$ such that $f(n) \leq c * g(n)$, due to the fact that one half of f grows much faster than the other half. The same argument can be made that g is not $O(f)$; because there is no constant N that exists for all constants c and when $n > N$ such that $g(n) \leq c * f(n)$ due to the very different growth rates, so there is no constant where we can guarantee that the inequality will be satisfied. Thus, this is a counterexample.
- This statement is true. By the definition of little-o, f is $o(g)$ if for all c there exists some N such that for all $n > N$, $f(n) < c * g(n)$. Consider the case of $c = 1$, and it is easy to see that there will be an N such that for all $n > N$, $f(n) \leq c * g(n)$, which is the definition for $f = O(g)$.

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

```
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 on this problem set with Nathan Lee, Jeremy Welborn, and Kate Schember.