1 Basic Algorithmic Analysis

For each of the following function pairs f and g, list out the Θ, Ω, O relationships between f and g, if any such relationship exists. The log function here denotes the natural logarithm.

For all the problems below, you should be able to eye the asymptotic relations without thinking about the limits which rigorously define them.

1. $f(x) = x^2$, $g(x) = x^2 + x$

   $f(x) \in \Theta(g(x))$: When comparing polynomials the only thing that matters is the degree

2. $f(x) = 50000x^3$, $g(x) = x^5$

   $f(x) \in O(g(x))$: Same as above, and $5 > 3$

3. $f(x) = \log(x)$, $g(x) = 5x$

   $f(x) \in O(g(x))$: Polynomials always grow faster than logarithms

4. $f(x) = e^x$, $g(x) = x^5$ (hint: $5 > e$)

   $f(x) \in \Omega(g(x))$: The hint was a red herring (sorry!), exponential growth is always faster than polynomial growth.

5. $f(x) = \log(5^x)$, $g(x) = x$

   $f(x) \in \Theta(g(x))$: It is a useful fact to remember that $\log_b(a)$ and $\log_c(a)$ differ by a constant multiple for any pair (b, c). In particular, $\log(5^x) = \alpha \log_5(5^x) = \alpha x$ for some α (remind yourself of the change-of-base formula if you're curious what α is!)

2 Practice with Runtime

For each of the following functions, find the Big-Theta expression for the runtime of the function in terms of the input variable n.

1. For this problem, you may assume that the static method *constant* runs in $\Theta(1)$ time.

   The outer nested loop runs n times, and the inner nested loop asymptotically runs n times (the actual number of times varies, but linearly with n), that means the first double-for loop runs in $\Theta(n^2)$ time (since the activity per inner loop - a print statement - runs in time independant of n). The second loop runs in $\Theta(n)$ time for the same reason, so we have $\Theta(n^2+n)$, but this is the same as $\Theta(n^2)$, which is the answer.

```
public static void thisIsANestedLoop(int n) {
    for (int i = 0; i < n; i += 1) {
        for (int j = 0; j < i; j += 1) {
            System.out.println(i + j);
        }
    }

    for (int k = 0; k < n; k += 1) {
        constant(k);
    }
}
```

2. This one is trickier! Note that in the final iteration of the outer loop, the inner loop will run n times. The iteration before, the inner loop will run $\frac{n}{2}$ times, and before that $\frac{n}{4}$, and so on. Abstracting, you can see that the number of times the inner loop will run is $n+\frac{n}{2}+\frac{n}{4}+...$. To figure out how many times the outer loop runs, we need to know how many times (starting with 1) you can double before reaching n: $\log_2(n)$. Therefore that sum is $\sum_{i=0}^{\log(n)} \frac{n}{2^i}$, which you might see is bounded by $2n$. Thus the total number of times the (constant-time) print statement is in $O(2n)$, and the overall runtime is $\Theta(n)$.

```
public static void thisIsMoreConfusing(int n) {
    for (int i = 1; i <= n; i *= 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("moo");
        }
    }
}
```

   That was tough! In general you won't be expected to be able to derive complicated runtimes on the fly in this class, but you do need to be aware of certain common paradigms. Among them, it is important to be aware that nested for loops do not always just increase the degree of the runtime.

3 A Bit with some Bits

Complete the following method such that it does what it is intended to do: given a list of integers, it returns an integer such that the i-th bit of the return value is 1 if and only if more than half of the integers in the list have 1 in the ith bit. Keep in mind that Java **int**s are 32 bits long!

Note: the solution to this question isn't very complicated, but it's not short! Try breaking it down into components, and ask your neighbors for help!

I'm sure there are multiple solutions to this problem. In general, the easiest thing to do is to keep track of some variable (toR here). At each i-stage, generate a number (j here) that represents a number with all 0s except for a 1 at the correct ith spot. You then AND through all the numbers in your list and see if the result is equal to the number j. If so, you increment some counter. If, after going through your list, your counter is big enough, you flip the bit in toR at the ith spot. Using bit-OR is the easiest way to increment the right place in toR.

```
public static int bitVote(int[] bitList) {
        int toR = 0;
        for (int i = 0; i < 32; i++) {
                int j = 1 << i;
                int count = 0;
                for (int k : bitList) {
                        if ((k & j) == j) {
                                count += 1;
                        }
                }
                if (count > bitList.length/2) {
                        toR = toR | j;
                }
        }
        return toR;
}
```