1. Asymptotics is Fun!

(a) Using the function $g$ defined below, what is the runtime of the following function calls? Write each answer in terms of $N$.

```c
void g(int N, int x) {
    if (N == 0) {
        return;
    }
    for (int i = 1; i <= x; i++) {
        g(N - 1, i);
    }
}
```

$g(N, 1)$: $\Theta(N)$

Explanation: When $x$ is 1, the loop gets executed once and makes a single recursive call to $g(N - 1)$. The recursion goes $g(N)$, $g(N - 1)$, $g(N - 2)$, and so on. This is a total of $N$ recursive calls, each doing constant work.

$g(N, 2)$: $\Theta(N^2)$

Explanation: When $x$ is 2, the loop gets executed twice. This means a call to $g(N)$ makes 2 recursive calls to $g(N - 1, 1)$ and $g(N - 1, 2)$. The recursion tree looks like this:

```
g(N, 2)
  /   \
g(N - 1, 1)   g(N - 1, 2)
  /       \
 g(N - 2, 1)   g(N - 2, 2)
```

From the first part, we know $g(\ldots, 1)$ does linear work. Thus, this is a recursion tree with $N$ levels, and the total work is $(N - 1) + (N - 2) + \ldots + 1 = \Theta(N^2)$ work.

(b) Suppose we change line 6 to $g(N - 1, x)$ and change the stopping condition in the for loop to $i <= f(x)$ where $f$ returns a random number between 1 and $x$, inclusive. For the following function calls, find the tightest $\Omega$ and big $O$ bounds.
```c
void g(int N, int x) {
    if (N == 0) {
        return;
    }
    for (int i = 1; i <= f(x); i++) {
        g(N - 1, x);
    }
}
```

Solution:

- \( g(N, 2) \): \( \Omega(N), O(2^N) \)
- \( g(N, N) \): \( \Omega(N), O(N^N) \)

Explanation: Suppose \( f(x) \) always returns 1. Then, this is the same as case 1 from (a), resulting in a linear runtime.

On the other hand, suppose \( f(x) \) always returns \( x \). Then \( g(N, x) \) makes \( x \) recursive calls to \( g(N - 1, x) \), each of which makes \( x \) recursive calls to \( g(N - 2, x) \), and so on, so the recursion tree has 1, \( x \), \( x^2 \) ... nodes per level. Outside of the recursion, the function \( g \) does \( x \) work per node. Thus, the overall work is \( x \times 1 + x \times x + x \times x^2 + \ldots + x \times x^{N-1} = x(1 + x + x^2 + \ldots + x^{N-1}) \).

Plug in \( x = 2 \) to get \( 2(1 + 2 + 2^2 + \ldots + 2^{N-1}) = O(2^N) \) for our first upper bound. Plug in \( x = N \) to get \( N(1 + N + N^2 + \ldots + N^{N-1}) = O(N^N) \) (ignoring lower-order terms).
2 Flip Flop

Suppose we have the flip function as defined below. Assume the method unknown returns a random integer between 1 and N, exclusive, and runs in constant time. For each definition of the flop method below, give the best and worst case runtime of flip in \( \Theta(\cdot) \) notation as a function of N.

```java
public static void flip(int N) {
    if (N <= 100) {
        return;
    }
    int stop = unknown(N);
    for (int i = 1; i < N; i++) {
        if (i == stop) {
            flop(i, N);
            return;
        }
    }
}
```

(a) ```java
public static void flop(int i, int N) {
    flip(N - i);
}
```  

Best Case: \( \Theta(\cdot) \), Worst Case: \( \Theta(\cdot) \)

**Solution:**  
Best Case: \( \Theta(N) \), Worst Case: \( \Theta(N) \)

**Explanation:** Consider some arbitrary value of stop. When stop = x, we do x work inside of flip (the for loop) and recursively call flip(N - x) through flop. This results in a total of \( N / x \) calls before reaching our base case, and x work per call, for a total of \( \Theta(N) \) work. Note that this holds for any value of x, so our best and worst case are the same.

(b) ```java
public static void flop(int i, int N) {
    int minimum = Math.min(i, N - i);
    flip(minimum);
    flip(minimum);
}
```  

Best Case: \( \Theta(\cdot) \), Worst Case: \( \Theta(\cdot) \)

**Solution:**  
Best Case: \( \Theta(1) \), Worst Case: \( \Theta(N \log N) \)

**Explanation:** In the best case, stop = 1. This hits the base case immediately, so we make 2 calls to flip then stop for \( \Theta(1) \) work.

In the worst case, stop = \( N / 2 \). This results in flip making 2 recursive calls to itself with the argument \( N / 2 \). Note the similarity of this recurrence and mergesort; the runtime is the same \( \Theta(N \log N) \).
(c) public static void flop(int i, int N) {
    flip(i);
    flip(N - i);
}

Best Case: $\Theta(\ )$, Worst Case: $\Theta(\ )$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$

Explanation: In the best case, suppose $\text{stop} = 1$. Then $\text{flip}(N)$ makes recursive calls to $\text{flip}(1)$ and $\text{flip}(N - 1)$, the first of which terminates immediately in the base case. $\text{flip}(N - 1)$ then calls $\text{flip}(1)$ and $\text{flip}(N - 2)$. The pattern is a linear recursion: constant work per call, $N$ calls total for $\Theta(N)$ work.

In the worst case, suppose $\text{stop} = N - 1$. Note that this case is symmetrical to the best case in terms of recursive calls; however we do work proportional to $N$ inside of $\text{flip}$ each time because of the for loop. The overall work is $(N - 1) + (N - 2) + (N - 3) + \ldots + 2 + 1 = \Theta(N^2)$. 
3  Prime Factors

Determine the best and worst case runtime of `prime_factors` in $\Theta(\cdot)$ notation as a function of $N$.

```java
int prime_factors(int N) {
    int factor = 2;
    int count = 0;
    while (factor * factor <= N) {
        while (N % factor == 0) {
            System.out.println(factor);
            count += 1;
            N = N / factor;
        }
        factor += 1;
    }
    return count;
}
```

Best Case: $\Theta(\cdot)$, Worst Case: $\Theta(\cdot)$

**Solution:**
Best Case: $\Theta(log(N))$, Worst Case: $\Theta(\sqrt{N})$

**Explanation:** In the best case, $N$ is some power of 2. Then the inner while loop will halve $N$ each time until it becomes 1. At this point, both the inner and outer while loop conditions will be false and the function will return. Halving $N$ each time results in a $\Theta(log N)$ runtime.

In the worst case, $N$ will not be divisible by any value of `factor`. This means we increment `factor` by 1 each time until `factor * factor > N`. This is at most $\sqrt{N}$ loops.