1 More Running Time

(a) Assume that `slam()` $\in \Theta(1)$ and returns a boolean.

```
1 public void comeon() {
2     int j = 0;
3     for (int i = 0; i < N; i += 1) {
4         for (; j < M; j += 1) {
5             if (slam(i, j))
6                 break;
7         }
8     }
9 }
```

(a) For `comeon()` the worst case is $\Theta(M+N)$ and the best case is $\Theta(N)$. To see this, note that `j` is declared outside of the for loops and so in the worst. In the best case, `bump(i, j)` always returns false. However, in the worst case where it's always false, `j` can only increment to at most M.

2 Recursive Running Time

This exercise targets understanding of how branching and input size affect the running time of analyzing recursive functions. I would explain each problem by writing the recurrence relation, draw the tree out, and ask (1) what's the branching factor, (2) how much work does each node contribute, (3) what's the height of the tree.

(a) Give the running time in terms of N.

```
public void andslam(int N) {
    if (n > 0) {
        for (int i = 0; i < N; i += 1) {
            System.out.println("datboi.jpg");
        }
    }
    andslam(n / 2);
}
```

(b) Give the running time for `andwelcome(arr, 0, N)` where N is the length of the input array `arr`.

```
public static void andwelcome(int[] arr, int low, int high) {
    System.out.print("[ ");
    for (int i = low; i < high; i += 1) {
         System.out.print("loyal  ");
    }
    System.out.println("]");
    if (high - low > 0) {
        double coin = Math.random();
        if (coin > 0.5) {
            andwelcome(arr, low, low + (high - low) / 2);
        } else {
            andwelcome(arr, low, low + (high - low) / 2);
            andwelcome(arr, low + (high - low) / 2, high);
        }
    }
}
```

(c) Give the running time in terms of N.

```
public int tothe(int N) {
    if (N <= 1) {
        return N;
    }
    return tothe(N - 1) + tothe(N - 1);
}
```

(d) Give the running time in terms of N

```
1 public static void spacejam(int n) {
2     if (n == 1) {
3         return;
4     }
5     for (int i = 0; i < n; i += 1) {
6         spacejam(n-1);
7     }
8 }
```

(a) `andslam(N)` runs in time $\Theta(N)$ worst and best case. The recurrence relation is $T(N) = T(\frac{N}{2}) + O(N)$.

(b) `andwelcome(arr, 0, N)` runs in time $\Theta(N \log N)$ worst case and $\Theta(N)$ best case. The recurrence relation is different for each case. In the worst case you always flip the wrong side of the coin resulting in a branching factor of 2. The recurence relation is $T(N) = 2T(\frac{N}{2}) + O(N)$. In the best case you always flip the right side of the coin giving a branching factor of 1. The recurrence relation is $T(N) = T(\frac{N}{2}) + O(N)$.

(c) For `tothe(N)` the worst and best case are $\Theta(2^N)$. The recurrence relation is $T(n) = 2T(n-1) + O(1)$. Draw the call tree out and for each node notice that there is constant work associated with it. The running time scales w.r.t. the number of nodes and we know that the number of nodes for a complete binary tree is $2^N - 1$.

(d) For `spacejam(N)` the worst and best case is $\Theta(N!)$. THe recurrence relation is $T(n) = nT(n-1) + O(1)$. To explain, I would again draw the tree out and make an argument based on the number of nodes.

3 Hey you watchu gon do

For each example below, there are two algorithms solving the same problem. Given the asymptotic runtimes for each, is one of the algorithms **guaranteed** to be faster? If so, which? And if neither is always faster, explain why. Assume the algorithms have very large input (so N is very large).

(a) Algorithm 1: $\Theta(N)$, Algorithm 2: $\Theta(N^2)$

(b) Algorithm 1: $\Omega(N)$, Algorithm 2: $\Omega(N^2)$

(c) Algorithm 1: $O(N)$, Algorithm 2: $O(N^2)$

(d) Algorithm 1: $\Theta(N^2)$, Algorithm 2: $O(\log N)$

(e) Algorithm 1: $O(N \log N)$, Algorithm 2: $\Omega(N \log N)$

(a) Algorithm 1: $\Theta(N)$ - straight forward, Θ gives tightest bounds

(b) Neither, something in $\Omega(N)$ could also be in $\Omega(N^2)$

(c) Neither, something in $O(N^2)$ could also be in $O(1)$

(d) Algorithm 2: $O(\log N)$ - Algorithm 2 cannot run SLOWER than $O(\log N)$ while Algorithm 1 is constrained on best and worst case by $\Theta(N^2)$.

(e) Neither, Algorithm 1 CAN be faster, but is not guaranteed - it is guaranteed to be "as fast as or faster" than Algorithm 2.

Would your answers above change if we did not assume that N was very large? Technically, no. But asymptotics are only applicable when considering behavior as N gets large. Consider this example: N^2 is asymptotically larger than $10000N$, yet when N is less than 10000, $10000N$ is larger than N^2.