Problem Set 5, Part I

Problem 1: A method that makes multiple recursive calls

1-1)

```
                             1:foo(3,3)
                              /       \
                             /         \
                            /           \
                   2:foo(4,1)        7:foo(1,4)
                    /     \              /     \
                  /       \        /       \
                 /         \          /         \
       3:foo(5,-1)   4:foo(2,2)  11:foo(-1,5)   8:foo(2,2)
            4         /     \           4         /     \
                     /       \                   /       \
                    /         \                 /         \
           5:foo(3,0)     6:foo(0,3)      9:foo(3,0)   10:foo(0,3)
                3              3              3             3
```

1-2)

```
// results of the left branch from the first call
Call 5 to foo(3,0) returns 3.
Call 6 to foo(0,3) returns 3.
Call 4 to foo(2,2) returns 6.
Call 3 to foo(5,-1) returns 4.
Call 2 to foo(4,1) returns 10.

// results of the right branch from the first call
Call 9 to foo(3,0) returns 3.
Call 10 to foo(0,3) returns 3.
Call 8 to foo(2,2) returns 6.
Call 11 to foo(-1,5) returns 4.
Call 7 to foo(1,4) returns 10.

// final return value.
Call 1 to foo(3,3) returns 20.
```

Problem 2: Expressing Big-O

1. a(n) = O(n)
2. b(n) = O(n^2)
3. c(n) = O(n^3)
4. d(n) = O(nlog(n))
5. e(n) = O(n^2)
6. f(n) = O(n^2)
7. g(n) = O(2^n)

Problem 3: Computing Big-O

// We are assuming that the time complexity of count() is constant.

// Initializing variables to be 1 instead of 0 in summations for the sake of range simplicity.

3-1) The time complexity of Code Fragment 1 is **O(n^3)** when its three nested summations of the loops are multiplied together, because the outermost loop runs n times, the second inner loop runs n times, and the third innermost loop runs n times for each iteration of the second inner loop.

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 1 = n * (1 + 2 + 3 + \dots + (n-1)) = O(n^3)$$

3-2) The time complexity of Code Fragment 2 is **O(n(log(n)))** when its three nested summations of the loops are multiplied together, because the outermost loop runs log(n) times, the second inner loop runs n times, and the third innermost loop runs 1000 times (this a constant and is omitted in the big-O notation.

$$\sum_{i=n}^{log(n)}\sum_{j=1}^{n}\sum_{k=1}^{1000n} 1 = log(n) * n * 1000 = O(nlog(n))$$

3-3) The time complexity of Code Fragment 3 is **O(n^2(log(n)))** when its three nested summations of the loops are multiplied together, because the innermost third loop runs log(n) times, the second loop runs n times, and the outermost loop runs n times.

$$\sum_{i=1}^{n}\sum_{j=1}^{2n}\sum_{k=1}^{log(n)} 1 = n * 2n * log(n) = O(n^2 log(n))$$

3-4) The time complexity of Code Fragment 4 is **O(n^2)** when its three nested summations of the loops are multiplied together, because the internal loop will run n times for each iteration for each second inner loop's iteration, and the outermost loop will run 3 times (a constant).

$$\sum_{i=1}^{3}\sum_{j=1}^{n}\sum_{k=1}^{n} 1 = 3\sum_{i=1}^{n}\sum_{k=i}^{n} 1 = 3(1 + 2 + 3 + \dots + (n-1)) = O(n^2)$$

3-5) the time complexity of Code Fragment 5 is **O(n^2)**, because the internal loop will run n times for each iteration for each external loop's iteration (from 1 to n).

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = 1 + 2 + 3 + ... + (n - 1) + n = O(n^2)$$

Problem 4: Comparing two algorithms

4-1) The worst case efficiency of algorithm A in terms of the length n of the array would be O(n^2), because the algorithm operates through a nested loop, in which the second loop is dependent on the first loop and produces the geometric sum as shown below.

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = 1 + 2 + 3 + ... + (n - 1) = O(n^2)$$

4-2) The worst case efficiency of algorithm B in terms of the length n of the array would be O(nlog(n)), because the algorithm produces the equation of nlog(n) + n, where nlog(n) is the dominant term. The algorithm has only a single for loop that checks only for as long as the length of the array even in a case where all elements may essentially be duplicates of one another.

4-3) Algorithm C's worst-case time efficiency is O(n), because, as shown below, it only runs as long as the length of the array through a single for loop.

```
public static int numDuplicates(int[] arr) {
      int n = arr.length;
      int duplicates = 0;
      boolean[] passedVals = new boolean[n];
      for (int i = 0; i < n; i++) {
            if (passedVals[arr[i]] == true) {
            duplicates++;
      } passedVals[arr[i]] = true;
      } return duplicates;
}
```

Problem 5: Sum generator

5-1) The sum = sum+j line is executed (n(n+1))/2 times.

5-2) The time efficiency of the algorithm is O(n^2) as it utilizes a nested loop that as its internal loop will run n times for each iteration for each external loop's iteration (from 1 to n) as shown in the summation below.

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = 1 + 2 + 3 + ... + (n - 1) = O(n^2)$$

5-3)

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

5-4) The time efficiency of this alternative implementation would be O(n), because the algorithm only utilizes a single for loop.

$$\sum_{i=1}^{n} 1 = n = O(n)$$

Problem 6: Basic Sorting Algorithms

6-1) {3, 4, 18, 24, 33, 40, 8, 10, 12}

6-2) {4, 10, 18, 24, 33, 40, 8, 3, 12}

6-3) {4, 10, 18, 8, 3, 12, 24, 33, 40}

Problem 7: Comparing two algorithms

7-1) The worst, best, and average time efficiency of Algorithm A is O(n), because the algorithm searches the entire array sequentially through a single for loop to keep track of the largest element, which can be represented as the following summation:

$$\sum_{i=1}^{n} 1 = n = O(n)$$

7-2) The worst, best, and average time efficiency of Algorithm B is O(n^2), because the non-optimized bubblesort algorithm executes adjacent value comparison regardless of if the array is sorted or not.

7-3) Algorithm A has more efficient run-time complexity than Algorithm B, because O(n) is faster in all cases than O(n^2).