Problem Set 5, Part I

Problem 1: A method that makes multiple recursive calls

1-1)

```
                             1:foo(3,3)
                                  20
                              /       \
                             /         \
                            /           \
                       2:foo(4,1)    7:foo(1,4)
                          10            10
                                       /   \
                        /   \         /     \
                       /     \  11:foo(-1,5) 8:foo(2,2)
                      /       \       4          6
                                               /  \
                3:foo(5,-1) 4:foo(2,2)        /    \
                    4            6     9:foo(3,0)  10:foo(0,3)
                               /  \         3          3
                              /    \
                             /      \
                      5:foo(3,0)   6:foo(0,3)
                           3          3
```

1-2)

```
Call 3 (foo(5,-1)) returns 4
Call 5 (foo(3,0)) returns 3
Call 6 (foo(0,3)) returns 3
Call 4 (foo(2,2)) returns 6
Call 2 (foo(4,1)) returns 10
Call 9 (foo(3,0)) returns 3
Call 10 (foo(0,3)) returns 3
Call 8 (foo(2,2)) returns 6
Call 11 (foo(-1,5)) returns 4
Call 7 (foo(1,4)) returns 10
Call 1 (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

3-1) since the outer loop iterates n times and the middle loop iterates n times and the inner loop iterates n because it produces count if k<j. The big-O notation would be n*n*n = O(n^3).

3-2) the outer loop iterates log(n) times, the middle loop iterates n times, and the O(1000). The big-O notation is log(n)*n*1000 = O(nlog(n))

3-3) the outer loop iterates n times, the middle loop iterates 2n times, and the inner loop iterates log(n) times. The big-O notation would be n*2n*log(n) = O(n^2 log(n))

3-4) the outer loop iterates O(3) times, the middle loop iterates n times, and the inner loop iterates n times. The big-O notation would be 3*n*n = O(n^2)

3-5) The outer loop iterates n times and the inner loop n times as well. The big-O notation would be n*n = O(n^2)

Problem 4: Comparing two algorithms

4-1) The worst-case time efficiency for Algorithm A in terms of the length n of the array is O(n^2) because it goes through the outer loop for the length of the array minus 1 times, which is n times, and it goes through the inner loop the length of the array times, which is also n. n*n = O(n^2)

4-2) The worst-case time efficiency for Algorithm B is O(n logn) because mergesort has a worst-case efficiency of O(n logn) and the for loop iterates n times. n + nlogn = O(n logn)

4-3)

```
public static int numDuplicatesC(int[] arr) {
        int numDups = 0;
        int copy = 0;
        for (int i = 0; i < arr.length; i++) {
                if (arr[i] == arr[copy]) {
                        numDups +=1;
                }
        }
        return numDups;
}
```

The worst case efficiency is O(n) since the algorithm does not use any sorting and just runs the first and only loop which is O(n).

Problem 5: Sum generator

5-1) 2n*(n(n+1))/2

5-2) The efficiency of this algorithm is O(n^3) because the simplification is n^3+n^2 and the slowest part of the algorithm is n^3.

5-3)

```
public static String generateSums(int n) {
        String result = "";
        int sum = 0;
        String otherResult = "";

        for (int i = 1; i <= n; i++) {
            sum += i;

            if (n == 0) {
                return result;
            }

            if (i == 1) {
                otherResult += i;
                result += otherResult + "\n";
            } else if (i <= n) {
                otherResult += " + " + i;
                result += otherResult;
                if (i != n) {
                    result += "\n";
                }
            }
        }
            result += " = " + sum;
            return result;
    }
```

5-4) The time efficiency is O(n) because the for loop iterates n times.

Problem 6: Basic Sorting Algorithms

6-1) The array looks like this after the second pass:
{3,4,18,24,33,40,8,10,12}

6-2) The array looks like this after the 4th iteration:
{4,10,18,24,33,40,8,3,12}

6-3) They array looks like this after the 3rd pass:
{4,10,18,8,3,12,24,33,40}

Problem 7: Comparing two algorithms

7-1) For algorithm A, the best case efficiency is O(n) if there is only one value, but the average and worst case is O(n) since it goes through all the elements n times.

7-2) For Algorithm B, The best case would be O(n^2) since it would already be sorted but the worst and average case would be O(n^2) since it goes through a nested for loop that has to iterate through.

7-3) Algorithm A would be more efficient since it has a faster run-time of O(n).