**Problem Set 5, Part I**

**Problem 1: Using recursion to print an array**

**1-1)**

```
public static void print(int[] arr, int start) {
    if (arr == null || start < 0) {
        throw new IllegalArgumentException();
    }
    if (start >= arr.length) {
        return;
    } else {
        System.out.println(arr[start]);
        print(arr, start+1);
    }
}
```

**1-2)**

```
public static void printReverse(int[] arr, int i) {
    if (arr == null || i < 0 || i > arr.length) {
        throw new IllegalArgumentException();
    }
    if (i == arr.length) {
        return;
    } else {
        printReverse(arr, i+1);
        System.out.println(arr[i]);
    }
}
```

**1-3)** ***initial call:*** `printReverse(arr, 0);`

**Problem 2: A method that makes multiple recursive calls**

**2-1)**

```
                                 1: foo(4, 7)
                             /                 \
                            /                   \
                           /                     \
                   2: foo(3, 5)               7. foo(5, 6)
                     /        \                  /       \
                    /          \                /         \
                   /            \              /           \
              3. foo(2, 3)   6. foo(4, 4)  8. foo(4, 4)   9. foo(6, 5)
                /       \          6             6              7
               /         \
              /           \
         4. foo(1, 1)    5. foo(3, 2)
              2               4
```

**2-2)**

```
call 4 (foo(1, 1)) returns 2
call 5 (foo(3, 2)) returns 4
call 3 (foo(2, 3)) returns 6
call 6 (foo(4, 4)) returns 6
call 2 (foo(3, 5)) returns 12
call 8 (foo(4, 4)) returns 6
call 9 (foo(6, 5)) returns 7
call 7 (foo(5, 6)) returns 13
call 1 (foo(4, 7)) returns 25
```

## Problem 3: Sorting practice

**3-1)**
{7, 10, 13, 27, 24, 20, 14, 33}

**3-2)**
{7, 13, 14, 24, 27, 20, 10, 33}

**3-3)**
{7, 13, 14, 20, 10, 24, 27, 33}

**3-4)**
{10, 7, 13, 27, 24, 20, 14, 33}

**3-5)**
{7, 10, 13, 27, 24, 20, 14, 33}

**3-6)**
{7, 13, 14, 27, 24, 20, 10, 33}

## Problem 4: Practice with big-O

**4-1)**

| **function** | **big-O expression** |
|---|---|
| a(n) = 5n + 1 | a(n) = O(n) |
| b(n) = 2n^3 + 3n^2 + nlog(n) | b(n) = O(n^3) |
| c(n) = 10 + 5nlog(n) + 10n | c(n) = O(nlogn) |
| d(n) = 4log(n) + 7 | d(n) = O(logn) |
| e(n) = 8 + n + 3n^2 | e(n) = O(n^2) |

**4-2)**
O(n^2)
The outer loop will be executed for 2*n=O(n) times. As the outer loop executes once, the inner loop will be executed for n-1=O(n) times. Therefore, the count() method is called for 2n(n-1)=O(n^2) times.

**4-3)**
O(nlogn)
The i loop will be executed for 5 times. As the i loop executes once, the j loop will be executed for n=O(n) times. As the j loop executes once, the k loop will be executed for log(n)=O(logn) times. Therefore, the count() method is called for 5*n*logn=O(nlogn) times.

**Problem 5: Comparing two algorithms**

*worst-case time efficiency of algorithm A:* O(nlogn)

*Explanation:* The worst case is the array in order from the largest to the smallest. The algorithm uses Mergesort, and there are log(n) levels in the call tree. Merging two halves of an array of size n requires 2n moves, so the number of moves in each level is 2n. The total number of moves is 2n*log(n) = O(nlogn). The number of comparisons is also O(nlogn). Thus, the running time is O(nlogn).

*worst-case time efficiency of algorithm B:* O(n)

*Explanation:* The worst case is the array in order from the smallest to the largest. The i in the for loop will increment from 0 to n-1. The number of comparison is O(n). The number of moves is also O(n). Thus, the running time is O(n).