Problem Set 5, Part I

Problem 1: Using recursion to print an array

1-1)

```
public static void print(int[] arr, int start) {
      // parameter start keeps track of where you are in the array
      // always check for null references first

      if (arr == null || arr.length == 0) {
            throw new IllegalArgumentException();
      }
      if (start < 0 || start >= arr.length) {
            throw new IllegalArgumentException();
      }

      if (start == arr.length-1) {
            System.out.println(arr[start]);
            return;
      } else {
            System.out.println(arr[start]);
            print(arr, start+1);
      }

}
```

1-2)

```
public static void printReverse(int[] arr, int i) {
      /* for full credit: the first call should consider
       * the first element of the array first.
       * i - the index of the element that will be printed at last
       */

      if (arr == null || arr.length == 0) {
            throw new IllegalArgumentException();
      }
      if (i < 0 || i >= arr.length) {
            throw new IllegalArgumentException();
      }
      if (i == arr.length-1) {
            System.out.println(arr[i]);
            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(nlog(n))
d(n) = 4log(n) + 7	d(n) = O(log(n))
e(n) = 8 + n + 3n^2	e(n) = O(n^2)

4-2) The outer loop performs (2*n) repetitions; the inner loop performs (n-1) repetitions. Therefore, count() is called 2n(n-1) (i.e., 2n^2-2n) times, and the time efficiency is O(n^2).

4-3) The outest loop performs 5 repetitions; the inner loop performs n repetitions; the innest loop performs $\log_2(n)$ repetitions. Therefore, count is called $5n(\log_2(n))$ times, whose Big-O notation is thereby O(nlogn).

Problem 5: Comparing two algorithms

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

explanation: **Algorithm A employs a merge sort algorithm to identify the maximum value. The time efficiency of merge sort algorithm is O(nlogn) for the best case, average cases, and the worst case, since it always repeatedly divides tye array in half.**

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

explanation: **The worst case for algorithm B is that the array is fully sorted in increasing order, and then the variable largest is assigned to a new value after each comparison. In the worst case scenario, since the length of the array is n, the total number of comparisons will be n times, `largest = arr[i]` will be executed n times, and O(n) = n. This indicates that the time efficiency of algorithm B is linearly related to the size of the array.**