Problem Set 5, Part I

Problem 1: Sorting practice

1-1)
{3, 4, 18, 24, 33, 40, 8, 10, 12}
1-2)
{4, 10, 18, 24, 33, 30, 8, 3, 12}
1-3)
{4, 10, 18, 8, 3, 12, 24, 33, 40}
1-4)
{10, 18, 4, 24, 12, 3, 8, 40, 33}
1-5)
{10, 18, 4, 8, 12, 3, 24, 40, 33}
1-6)
{4, 10, 18, 24, 33, 40, 8, 3, 12}

Problem 2: Practice with big-O

2-1)

function	**big-O expression**
a(n) = 5n + 1	a(n) = O(n)
b(n) = 5 - 10n - n^2	b(n) = O(n^2)
c(n) = 4n + 2log(n)	c(n) = O(n)
d(n) = 6nlog(n) + n^2	d(n) = O(n^2)
e(n) = 2n^2 + 3n^3 - 7n	e(n) = O(n^3)

2-2)
O(n) = (3(n)(n+1))/2 = O(n^2)
The outer for loop will run 3 times, the second for loop will run ‘n’ times, and the third for loop will run ‘j’ times, where ‘j’ is the iteration of the second for loop. Furthermore, the second and third for loop together is an arithmetic sum sequence, which can be written as ((n)(n+1)/2). Multiplying everything together, count() is called (3(n)(n+1))/2 times, or in Big-O notation: O(n^2) times.

2-3)
O(n) = n(log2(n)) = n(log(n))
The first for loop will run ‘n’ times. The second for loop runs a total of log (base 2) of ‘n’ times. This second for loop is logarithmic because it decrements by a multiple of 2 every time rather than at a constant linear rate. You are dividing j by 2 each time. Thus, count() is called O(n*log(n)) times.

Problem 3: Comparing two algorithms
worst-case time efficiency of algorithm A: O(n^2)
explanation: The worst case for this duplicate counting algorithm is if there was an array of n elements that had no duplicates. Therefore, the method would have to iterate through all n elements of the array for the outer for loop. Furthermore, for each iteration in the outer loop, the method must also iterate through the inner for loop n-1 times. We don't check the last value because there can't be any duplicates beyond the last value. The time complexity of this method is n(n-1), or O(n^2) quadratic time, which is not very efficient.

worst-case time efficiency of algorithm B: O(n log n)
Explanation: An example of the worst-case time efficiency of algorithm B would be an array that is an alternative array between a small number, a number larger than the previous, and then a number smaller than the second, and so on. For example, {1, 10, 3, 7, 2, 5} would be an example of the worst-case time efficiency because we use mergesort. Regardless, mergesort's best and worst case is always O(n log n). Since we use apply mergesort into the algorithm, the array is always in order afterward. Even in the worst case, we would simply go through the entire array n times. Since the loop is separate from the mergesort algorithm, we add n log n to n. Thus, Big-O is O(n log n).

Problem 4: Practice with references

4-1)

Expression	**Address**	**Value**
n	0x100	0x712
n.ch	0x712	‘n’
n.prev	0x718	0x064
n.prev.prev	0x070	0x360
n.prev.next.next	0x714	null
n.prev.prev.next	0x362	0x064

4-2)

```
x.next = n;
x.prev = n.prev;
n.prev.next = x;
n.prev = x;
```

4-3)

```
public static void initPrevs(DNode first) {
    first.prev = null;
    DNode trav = first.next;
    DNode behind = first;
    while (trav != null) {
        trav.prev = behind;
        trav = trav.next;
        behind = behind.next;
    }
}
```