Problem Set 4, Part I

Problem 1: Sorting practice

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

1-2) {7, 13, 14, 24, 27, 20, 10, 33}

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

1-4) {10, 7, 13, 27, 24, 20, 14, 33}

1-5) {7, 10, 13, 27, 24, 20, 14, 33}

1-6) {7, 13, 14, 27, 24, 20, 10, 33}

Problem 2: Practice with big-O

2-1)

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)

2-2) The outer loop will be iterated 2n times, while the inner loop will be iterated n - 1 times. Since the inner loop is independent of the outer loop, count() will be called 2n * (n - 1) = 2n^2 - 2n times, and the big-O expression is given as O(n^2).

2-3) The first (outer-most) loop will run exactly 5 times, independent of the value of n. The first inner loop will run n times, while the second inner (inner-most) loop will run $\log_2 n$ times. Since the outer loop and inner loops are independent of each other, count() will be called a(n)*b(n)*c(n) times (where a,b,c signify the 3 different loops of the nested loop). Hence, the count() method is called 5 * n * $\log_2 n$ = 5$\log_2$(n) times, and the big-O expression is given as O(nlog(n)).

Problem 3: Comparing two algorithms

worst-case time efficiency of algorithm A: O(nlog(n))
Explanation: the built in mergesort runs consistently regardless of whether the array is sorted or not and like the coded version of merge sort, it will halve the array and do the comparisons and create the new arrays and merge them back. Returning the last (largest) element of the array takes a constant time O(1) which is insignificant compared to the divide and conquer, merge sort algorithm which has a big-O notation of O(n) = nlog(n).

worst-case time efficiency of algorithm B: O(n)
Explanation: The worst case scenario is where the largest value is located at the end of the array (last index). Hence the for-loop passes through the entire array before finding the largest array. The initialization and returning of the largest value takes a constant time O(1) which is insignificant compared to the for-loop which scales proportionately with n (O(n) = n).

Problem 4: Counting unique values

4-1) The worst case scenario occurs when there are n distinct elements.

4-2) n^2/2 - n/2

4-3) The outer loop runs n times and the inner loop runs n^2/2-n/2 times. The boolean assignment of `appearsAfter = false` will run at O(1) n times because it is embedded within the outer loop. The comparison `arr[j] == arr[i]` will similarly run at O(1) n^2/2-n/2 times, but the `break` will not execute because each element is distinct and thus the inner loop will keep running. Finally, the conditional `if (!appearsAfter)` will run at O(1). Thus, the worst case time efficiency will be O(n + n^2/2 - n/2) = O(n^2).

4-4) The best case scenario occurs when there are n NON-DISTINCT elements.

4-5) The outer loop runs n times regardless. However, the inner loop will only compare once each time and `break` executes after the comparison `arr[j] == arr[i]`, which is always true in this case. Hence, the overall best case time efficiency will be O(n).

Problem 5: Improving the efficiency of an algorithm

5-1)

```
public static int numUnique(int[] arr) {
      Sort.mergesort(arr);
      int count = 1;
      for (int i = 1; i < arr.length; i++) {
            if (arr[i] != arr[i -1]) {
                  count ++;
            }
      }
      return count;
}
```

5-2) The worst case time efficiency is now O(nlog(n)) because the comparison and sorting are independent of each other so the function can be written as C(n) + S(n). Since the comparison is O(n) and merge sort is O(nlog(n)), the dominant term is O(nlog(n)).

5-3) The new method now has a best case time efficiency has a dominant term of O(nlog(n)). This is because merge sort has a time efficiency of O(nlog(n)) regardless of the order of the elements in the array. Thus, the time efficiency is fixed for both worst case and best case.

Problem 6: Practice with references

6-1)

Expression	**Address**	**Value**
n	0x128	0x800
n.ch	0x128	‘e’
n.next	0x800	0x240
n.prev.next	0x180	0x800
n.next.prev	0x240	0x800
n.next.prev.prev	0x800	0x180

6-2)

```
public static void main(String[] args) {
      m.next = n.next;
      n.next = m;
      m.prev = m.next.prev;
      m.next.prev = m;
}
```

6-3)

```
public static void addNexts(DNode last) {
      while (last.prev != null) {
            last.prev.next = last;
            last = last.prev;
      }
}
```

Problem 7: Printing the odd values in a list of integers

7-1)

```
public static void printOddsRecur(IntNode first) {
      if (first == null) {
            return;
      }

      if (first.val % 2 == 1) {
            System.out.println(first.val);
      }

      printOddsRecur(Intnode first.next);
}
```

7-2)

```
public static void printOddsIter(IntNode first) {
      while (first != null) {
            if (first.val % 2 == 1) {
                  System.out.println(first.val);
            }
            first = first.next;
      }
}
```